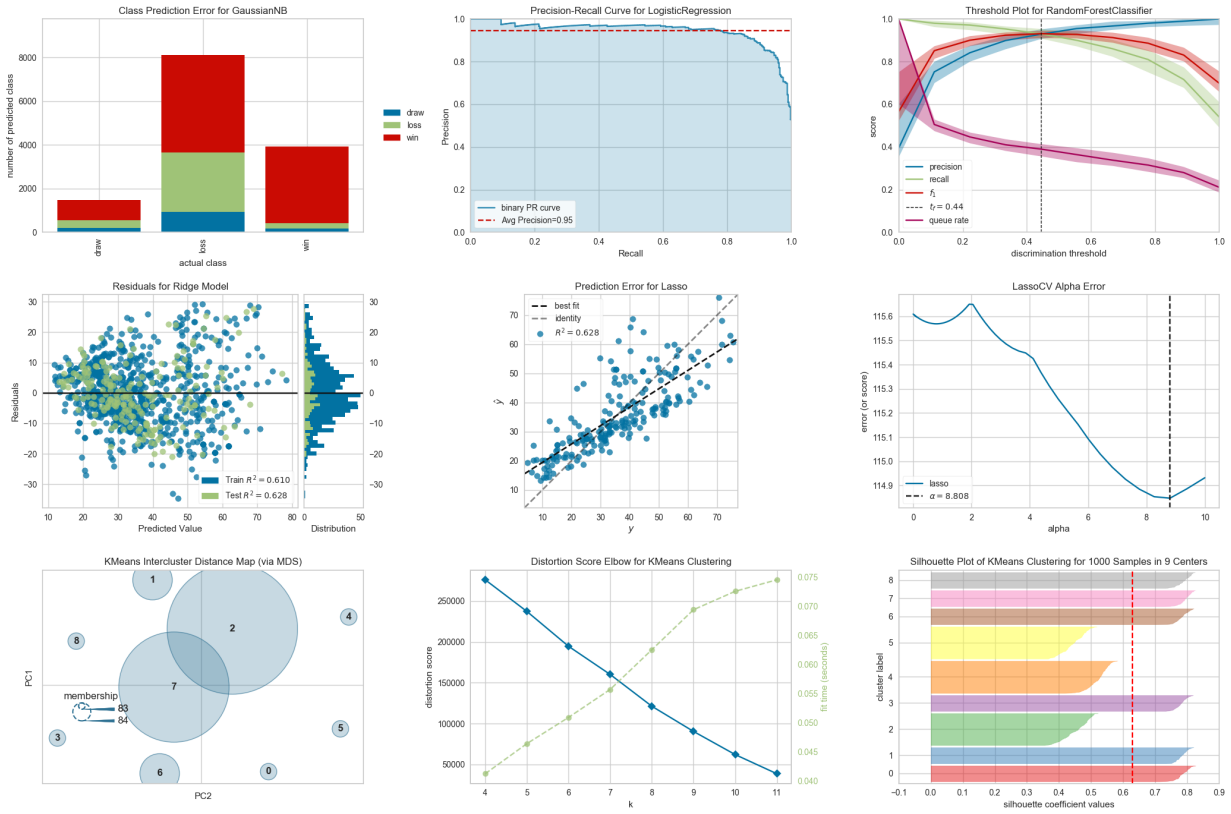
Yellowbrick Documentation

Release v1.5

The scikit-yb developers

Aug 21, 2022

CONTENTS



Yellowbrick extends the Scikit-Learn API to make model selection and hyperparameter tuning easier. Under the hood, it's using Matplotlib.

CHAPTER

ONE

RECOMMENDED LEARNING PATH

1. Check out the *Quick Start*, try the *Model Selection Tutorial*, and check out the *Oneliners*.
2. Use Yellowbrick in your work, referencing the *Visualizers and API* for assistance with specific visualizers and detailed information on optional parameters and customization options.
3. Star us on GitHub and follow us on Twitter (@scikit_yb) so that you'll hear about new visualizers as soon as they're added.

CHAPTER

TWO

CONTRIBUTING

Interested in contributing to Yellowbrick? Yellowbrick is a welcoming, inclusive project and we would love to have you. We follow the Python Software Foundation Code of Conduct.

No matter your level of technical skill, you can be helpful. We appreciate bug reports, user testing, feature requests, bug fixes, product enhancements, and documentation improvements.

Check out the *Contributing* guide!

If you've signed up to do user testing, head over to the *User Testing Instructions*.

Please consider joining the Google Groups Listserv listserve so you can respond to questions.

Thank you for your contributions!

CHAPTER

THREE

CONCEPTS & API

CHAPTER

FOUR

VISUALIZERS

The primary goal of Yellowbrick is to create a sensical API similar to Scikit-Learn.

Visualizers are the core objects in Yellowbrick. They are similar to transformers in Scikit-Learn. Visualizers can wrap a model estimator - similar to how the "ModelCV" (e.g. RidgeCV, LassoCV) methods work.

Some of our most popular visualizers include:

4.1 Feature Visualization

- *Rank Features*: pairwise ranking of features to detect relationships
- *Parallel Coordinates*: horizontal visualization of instances
- *Radial Visualization*: separation of instances around a circular plot
- *PCA Projection*: projection of instances based on principal components
- *Manifold Visualization*: high dimensional visualization with manifold learning
- *Joint Plots*: direct data visualization with feature selection

4.2 Classification Visualization

- *Class Prediction Error*: shows error and support in classification
- *Classification Report*: visual representation of precision, recall, and F1
- *ROC/AUC Curves*: receiver operator characteristics and area under the curve
- *Precision-Recall Curves*: precision vs recall for different probability thresholds
- *Confusion Matrices*: visual description of class decision making
- *Discrimination Threshold*: find a threshold that best separates binary classes

4.3 Regression Visualization

- *Prediction Error Plot*: find model breakdowns along the domain of the target
- *Residuals Plot*: show the difference in residuals of training and test data
- *Alpha Selection*: show how the choice of alpha influences regularization
- *Cook's Distance*: show the influence of instances on linear regression

4.4 Clustering Visualization

- *K-Elbow Plot*: select k using the elbow method and various metrics
- *Silhouette Plot*: select k by visualizing silhouette coefficient values
- *Intercluster Distance Maps*: show relative distance and size/importance of clusters

4.5 Model Selection Visualization

- *Validation Curve*: tune a model with respect to a single hyperparameter
- *Learning Curve*: show if a model might benefit from more data or less complexity
- *Feature Importances*: rank features by importance or linear coefficients for a specific model
- *Recursive Feature Elimination*: find the best subset of features based on importance

4.6 Target Visualization

- *Balanced Binning Reference*: generate a histogram with vertical lines showing the recommended value point to bin the data into evenly distributed bins
- *Class Balance*: see how the distribution of classes affects the model
- *Feature Correlation*: display the correlation between features and dependent variables

4.7 Text Visualization

- *Term Frequency*: visualize the frequency distribution of terms in the corpus
- *t-SNE Corpus Visualization*: use stochastic neighbor embedding to project documents
- *Dispersion Plot*: visualize how key terms are dispersed throughout a corpus
- *UMAP Corpus Visualization*: plot similar documents closer together to discover clusters
- *PosTag Visualization*: plot the counts of different parts-of-speech throughout a tagged corpus

… and more! Visualizers are being added all the time. Check the examples (or even the develop branch). Feel free to contribute your ideas for new Visualizers!

CHAPTER

FIVE

GETTING HELP

Can't get someting to work? Here are places you can find help.

1. The docs (you're here!).
2. Stack Overflow. If you ask a question, please tag it with "yellowbrick".
3. The Yellowbrick Google Groups Listserv.
4. You can also Tweet or direct message us on Twitter @scikit_yb.

CHAPTER

SIX

FIND A BUG?

Check if there's already an open issue on the topic. If needed, file an issue.

CHAPTER

SEVEN

OPEN SOURCE

The Yellowbrick license is an open source Apache 2.0 license. Yellowbrick enjoys a very active developer community; please consider *Contributing*!

Yellowbrick is hosted on GitHub. The issues and pull requests are tracked there.

CHAPTER

EIGHT

TABLE OF CONTENTS

8.1 Quick Start

If you're new to Yellowbrick, this guide will get you started and help you include visualizers in your machine learning workflow. Before we begin, however, there are several notes about development environments that you should consider.

Yellowbrick has two primary dependencies: scikit-learn and matplotlib. If you do not have these Python packages, they will be installed alongside Yellowbrick. Note that Yellowbrick works best with scikit-learn version 0.20 or later and matplotlib version 3.0.1 or later. Both of these packages require some C code to be compiled, which can be difficult on some systems, like Windows. If you're having trouble, try using a distribution of Python that includes these packages like Anaconda.

Yellowbrick is also commonly used inside of a Jupyter Notebook alongside Pandas data frames. Notebooks make it especially easy to coordinate code and visualizations; however, you can also use Yellowbrick inside of regular Python scripts, either saving figures to disk or showing figures in a GUI window. If you're having trouble with this, please consult matplotlib's backends documentation.

Note: Jupyter, Pandas, and other ancillary libraries like the Natural Language Toolkit (NLTK) for text visualizers are not installed with Yellowbrick and must be installed separately.

8.1.1 Installation

Yellowbrick is a Python 3 package and works well with 3.4 or later. The simplest way to install Yellowbrick is from PyPI with pip, Python's preferred package installer.

```
$ pip install yellowbrick
```

Note that Yellowbrick is an active project and routinely publishes new releases with more visualizers and updates. In order to upgrade Yellowbrick to the latest version, use `pip` as follows.

```
$ pip install -U yellowbrick
```

You can also use the `-U` flag to update scikit-learn, matplotlib, or any other third party utilities that work well with Yellowbrick to their latest versions.

If you're using Anaconda, you can take advantage of the conda utility to install the Anaconda Yellowbrick package:

```
conda install -c districtdatalabs yellowbrick
```

If you're having trouble with installation, please let us know on GitHub.

Once installed, you should be able to import Yellowbrick without an error, both in Python and inside of Jupyter notebooks. Note that because of matplotlib, Yellowbrick does not work inside of a virtual environment on macOS without jumping through some hoops.

8.1.2 Using Yellowbrick

The Yellowbrick API is specifically designed to play nicely with scikit-learn. The primary interface is therefore a `Visualizer` – an object that learns from data to produce a visualization. Visualizers are scikit-learn Estimator objects and have a similar interface along with methods for drawing. In order to use visualizers, you simply use the same workflow as with a scikit-learn model, import the visualizer, instantiate it, call the visualizer's `fit()` method, then in order to render the visualization, call the visualizer's `show()` method.

For example, there are several visualizers that act as transformers, used to perform feature analysis prior to fitting a model. The following example visualizes a high-dimensional data set with parallel coordinates:

```
from yellowbrick.features import ParallelCoordinates

visualizer = ParallelCoordinates()
visualizer.fit_transform(X, y)
visualizer.show()
```

As you can see, the workflow is very similar to using a scikit-learn transformer, and visualizers are intended to be integrated along with scikit-learn utilities. Arguments that change how the visualization is drawn can be passed into the visualizer upon instantiation, similarly to how hyperparameters are included with scikit-learn models.

The `show()` method finalizes the drawing (adding titles, axes labels, etc) and then renders the image on your behalf. If you're in a Jupyter notebook, the image should just appear in the notebook output. If you're in a Python script, a GUI window should open with the visualization in interactive form. However, you can also save the image to disk by passing in a file path as follows:

```
visualizer.show(outpath="pcoords.png")
```

The extension of the filename will determine how the image is rendered. In addition to the `.png` extension, `.pdf` is also commonly used for high-quality publication ready images.

Note: Data input to Yellowbrick is identical to that of scikit-learn. Datasets are usually described with a variable `X` (sometimes referred to simply as data) and an optional variable `y` (usually referred to as the target). The required data `X` is a table that contains instances (or samples) which are described by features. `X` is therefore a *two-dimensional matrix* with a shape of `(n, m)` where `n` is the number of instances (rows) and `m` is the number of features (columns). `X` can be a Pandas DataFrame, a NumPy array, or even a Python lists of lists.

The optional target data, `y`, is used to specify the ground truth in supervised machine learning. `y` is a vector (a one-dimensional array) that must have length `n` – the same number of elements as rows in `X`. `y` can be a Pandas Series, a Numpy array, or a Python list.

Visualizers can also wrap scikit-learn models for evaluation, hyperparameter tuning and algorithm selection. For example, to produce a visual heatmap of a classification report, displaying the precision, recall, F1 score, and support for each class in a classifier, wrap the estimator in a visualizer as follows:

```
from yellowbrick.classifier import ClassificationReport
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
```

(continues on next page)

(continued from previous page)

```
visualizer = ClassificationReport(model)

visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.show()
```

Only two additional lines of code are required to add visual evaluation of the classifier model, the instantiation of a `ClassificationReport` visualizer that wraps the classification estimator and a call to its `show()` method. In this way, Visualizers *enhance* the machine learning workflow without interrupting it.

The class-based API is meant to integrate with scikit-learn directly, however on occasion there are times when you just need a quick visualization. Yellowbrick supports quick functions for taking advantage of this directly. For example, the two visual diagnostics could have been instead implemented as follows:

```
from sklearn.linear_model import LogisticRegression

from yellowbrick.features import parallel_coordinates
from yellowbrick.classifier import classification_report

# Displays parallel coordinates
g = parallel_coordinates(X, y)

# Displays classification report
g = classification_report(LogisticRegression(), X, y)
```

These quick functions give you slightly less control over the machine learning workflow, but quickly get you diagnostics on demand and are very useful in exploratory processes.

8.1.3 Walkthrough

Let's consider a regression analysis as a simple example of the use of visualizers in the machine learning workflow. Using a bike sharing dataset based upon the one uploaded to the UCI Machine Learning Repository, we would like to predict the number of bikes rented in a given hour based on features like the season, weather, or if it's a holiday.

Note: We have updated the dataset from the UCI ML repository to make it a bit easier to load into Pandas; make sure you download the Yellowbrick version of the dataset using the `load_bikeshare` method below. Please also note that Pandas is required to follow the supplied code. Pandas can be installed using `pip install pandas` if you haven't already installed it.

We can load our data using the `yellowbrick.datasets` module as follows:

```
import pandas as pd
from yellowbrick.datasets import load_bikeshare

X, y = load_bikeshare()
print(X.head())
```

This prints out the first couple lines of our dataset which looks like:

```
   season  year  month  hour  holiday  weekday  workingday  weather  temp  \
0       1     0      1     0        0        6           0        1  0.24
```

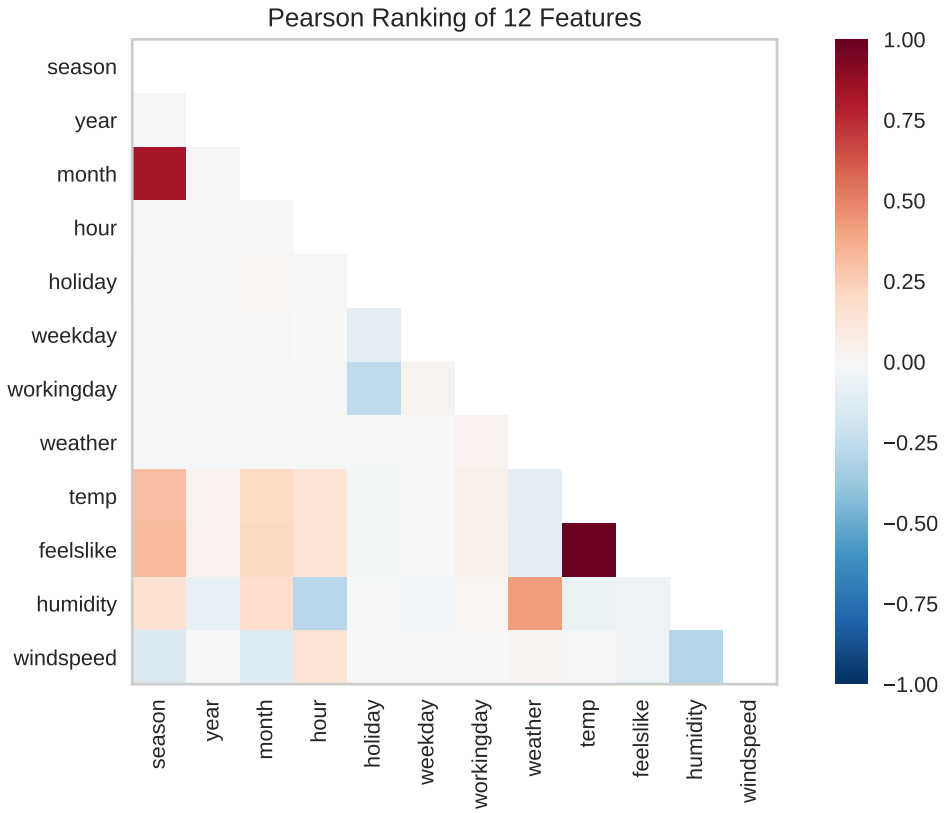
(continues on next page)

(continued from previous page)

```
1       1     0       1     1         0         6           0        1  0.22
2       1     0       1     2         0         6           0        1  0.22
3       1     0       1     3         0         6           0        1  0.24
4       1     0       1     4         0         6           0        1  0.24

   feelslike  humidity  windspeed
0     0.2879      0.81        0.0
1     0.2727      0.80        0.0
2     0.2727      0.80        0.0
3     0.2879      0.75        0.0
4     0.2879      0.75        0.0
```

The machine learning workflow is the art of creating *model selection triples*, a combination of features, algorithm, and hyperparameters that uniquely identifies a model fitted on a specific data set. As part of our feature selection, we want to identify features that have a linear relationship with each other, potentially introducing covariance into our model and breaking OLS (guiding us toward removing features or using regularization). We can use the *Rank Features* visualizer to compute Pearson correlations between all pairs of features as follows:

```
from yellowbrick.features import Rank2D

visualizer = Rank2D(algorithm="pearson")
visualizer.fit_transform(X)
visualizer.show()
```

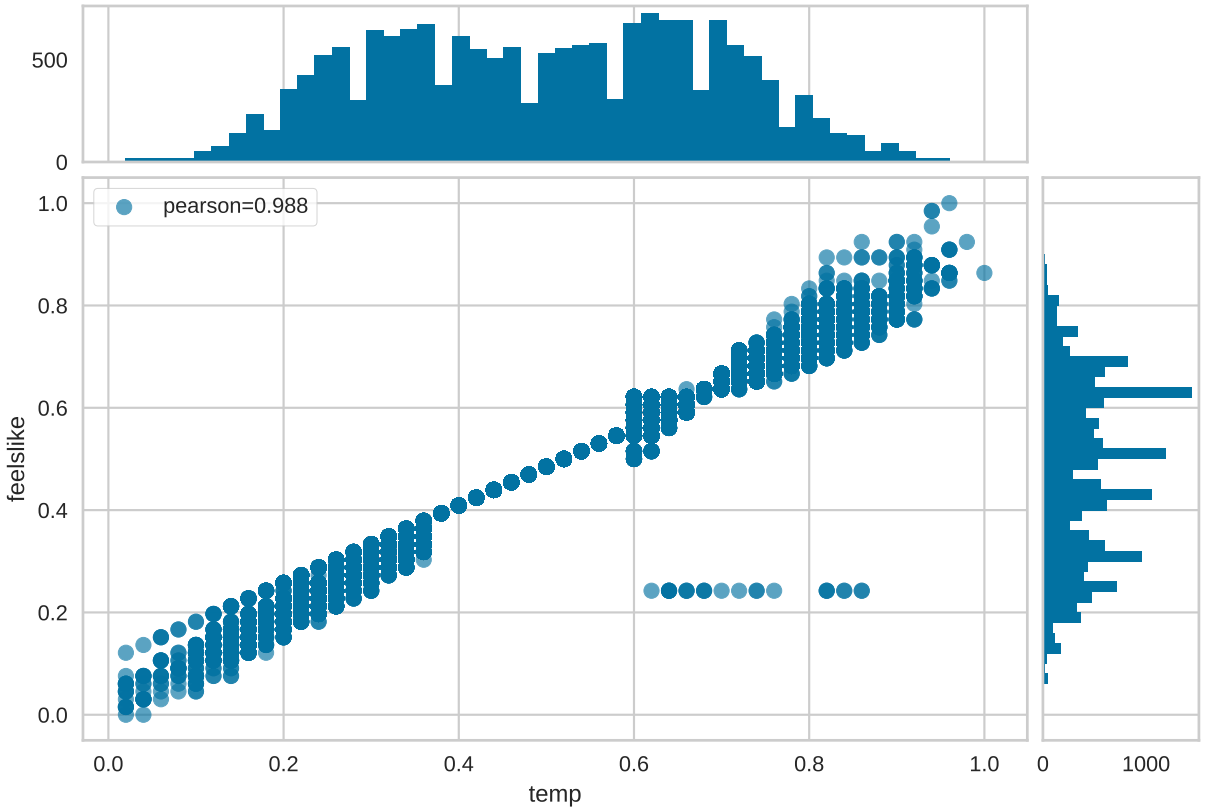


This figure shows us the Pearson correlation between pairs of features such that each cell in the grid represents two

features identified in order on the x and y axes and whose color displays the magnitude of the correlation. A Pearson correlation of 1.0 means that there is a strong positive, linear relationship between the pairs of variables and a value of -1.0 indicates a strong negative, linear relationship (a value of zero indicates no relationship). Therefore we are looking for dark red and dark blue boxes to identify further.

In this chart, we see that the features `temp` and `feelslike` have a strong correlation and also that the feature `season` has a strong correlation with the feature `month`. This seems to make sense; the apparent temperature we feel outside depends on the actual temperature and other airquality factors, and the season of the year is described by the month! To dive in deeper, we can use the *Direct Data Visualization* (`JointPlotVisualizer`) to inspect those relationships.

```
from yellowbrick.features import JointPlotVisualizer

visualizer = JointPlotVisualizer(columns=['temp', 'feelslike'])
visualizer.fit_transform(X, y)
visualizer.show()
```



This visualizer plots a scatter diagram of the apparent temperature on the y axis and the actual measured temperature on the x axis and draws a line of best fit using a simple linear regression. Additionally, univariate distributions are shown as histograms above the x axis for temp and next to the y axis for feelslike. The `JointPlotVisualizer` gives an at-a-glance view of the very strong positive correlation of the features, as well as the range and distribution of each feature. Note that the axes are normalized to the space between zero and one, a common technique in machine learning to reduce the impact of one feature over another.

This plot is very interesting because there appear to be some outliers in the dataset. These instances may need to be manually removed in order to improve the quality of the final model because they may represent data input errors, and potentially train the model on a skewed dataset which would return unreliable model predictions. The first instance of outliers occurs in the temp data where the `feelslike` value is approximately equal to 0.25 - showing a horizontal line

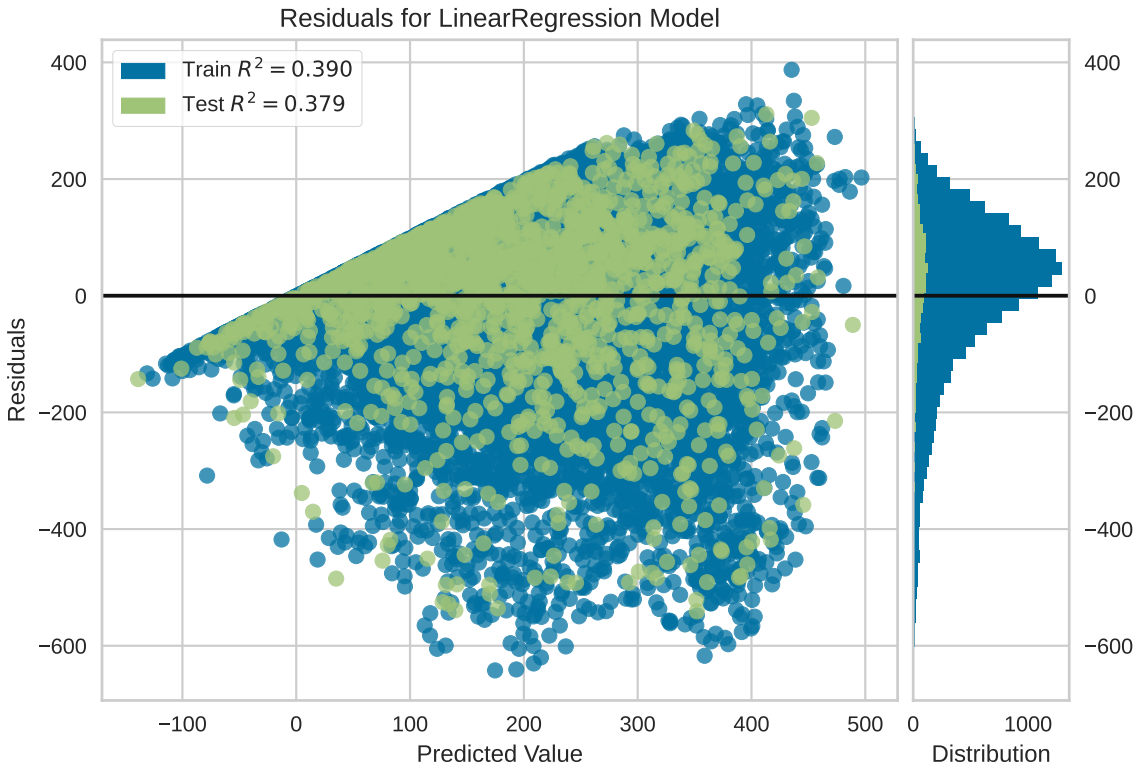
of data, likely created by input error.

We can also see that more extreme temperatures create an exaggerated effect in perceived temperature; the colder it is, the colder people are likely to believe it to be, and the warmer it is, the warmer it is perceived to be, with moderate temperatures generally having little effect on individual perception of comfort. This gives us a clue that `feelslike` may be a better feature than `temp` - promising a more stable dataset, with less risk of running into outliers or errors.

We can ultimately confirm the assumption by training our model on either value, and scoring the results. If the `temp` value is indeed less reliable, we should remove the `temp` variable in favor of `feelslike` . In the meantime, we will use the `feelslike` value due to the absence of outliers and input error.

At this point, we can train our model; let's fit a linear regression to our model and plot the residuals.

```
from yellowbrick.regressor import ResidualsPlot
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Create training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.1
)

visualizer = ResidualsPlot(LinearRegression())
visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.show()
```



The residuals plot shows the error against the predicted value (the number of riders), and allows us to look for het-

eroskedasticity in the model; e.g. regions in the target where the error is greatest. The shape of the residuals can strongly inform us where OLS (ordinary least squares) is being most strongly affected by the components of our model (the features). In this case, we can see that the lower predicted number of riders results in lower model error, and conversely that the the higher predicted number of riders results in higher model error. This indicates that our model has more noise in certain regions of the target or that two variables are colinear, meaning that they are injecting error as the noise in their relationship changes.

The residuals plot also shows how the model is injecting error, the bold horizontal line at `residuals = 0` is no error, and any point above or below that line indicates the magnitude of error. For example, most of the residuals are negative, and since the score is computed as `actual - expected`, this means that the expected value is bigger than the actual value most of the time; e.g. that our model is primarily guessing more than the actual number of riders. Moreover, there is a very interesting boundary along the top right of the residuals graph, indicating an interesting effect in model space; possibly that some feature is strongly weighted in the region of that model.

Finally the residuals are colored by training and test set. This helps us identify errors in creating train and test splits. If the test error doesn't match the train error then our model is either overfit or underfit. Otherwise it could be an error in shuffling the dataset before creating the splits.

Along with generating the residuals plot, we also measured the performance by "scoring" our model on the test data, e.g. the code snippet `visualizer.score(X_test, y_test)`. Because we used a linear regression model, the scoring consists of finding the R-squared value of the data, which is a statistical measure of how close the data are to the fitted regression line. The R-squared value of any model may vary slightly between prediction/test runs, however it should generally be comparable. In our case, the R-squared value for this model was only 0.328, suggesting that linear correlation may not be the most appropriate to use for fitting this data. Let's see if we can fit a better model using *regularization*, and explore another visualizer at the same time.

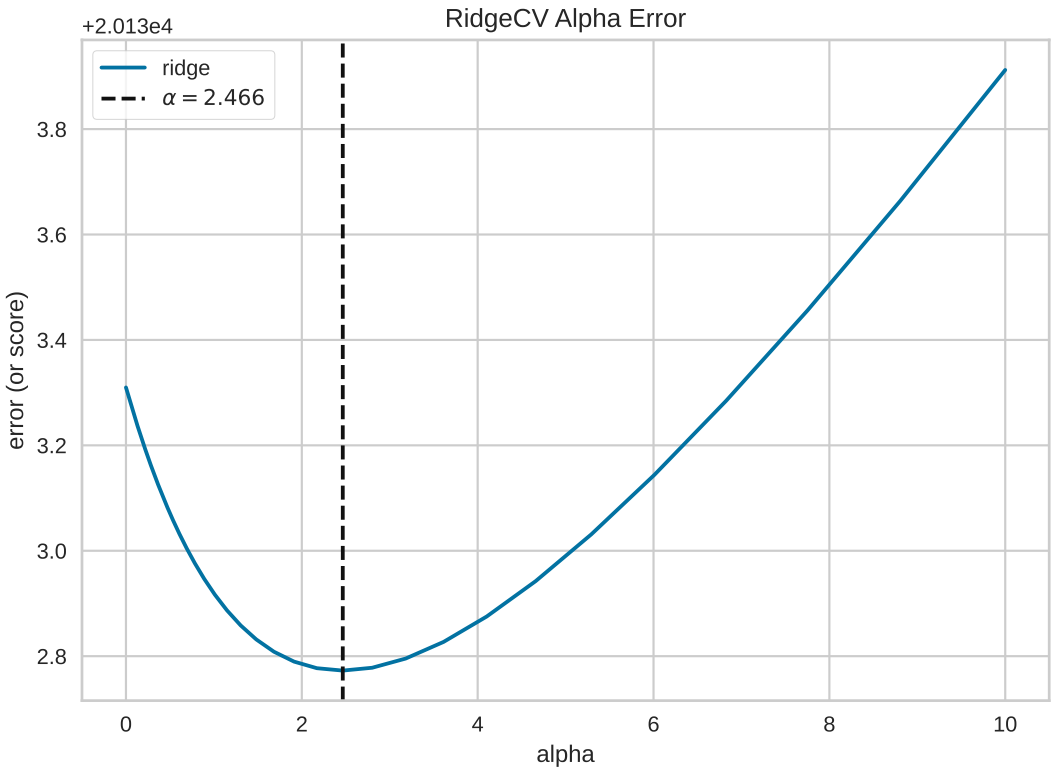
```
import numpy as np

from sklearn.linear_model import RidgeCV
from yellowbrick.regressor import AlphaSelection

alphas = np.logspace(-10, 1, 200)
visualizer = AlphaSelection(RidgeCV(alphas=alphas))
visualizer.fit(X, y)
visualizer.show()
```

When exploring model families, the primary thing to consider is how the model becomes more *complex*. As the model increases in complexity, the error due to variance increases because the model is becoming more overfit and cannot generalize to unseen data. However, the simpler the model is the more error there is likely to be due to bias; the model is underfit and therefore misses its target more frequently. The goal therefore of most machine learning is to create a model that is *just complex enough*, finding a middle ground between bias and variance.

For a linear model, complexity comes from the features themselves and their assigned weight according to the model. Linear models therefore expect the *least number of features* that achieves an explanatory result. One technique to achieve this is *regularization*, the introduction of a parameter called alpha that normalizes the weights of the coefficients with each other and penalizes complexity. Alpha and complexity have an inverse relationship, the higher the alpha, the lower the complexity of the model and vice versa.

The question therefore becomes how you choose alpha. One technique is to fit a number of models using cross-validation and selecting the alpha that has the lowest error. The `AlphaSelection` visualizer allows you to do just that, with a visual representation that shows the behavior of the regularization. As you can see in the figure above, the error decreases as the value of alpha increases up until our chosen value (in this case, 3.181) where the error starts to increase. This allows us to target the bias/variance trade-off and to explore the relationship of regularization methods (for example Ridge vs. Lasso).

We can now train our final model and visualize it with the `PredictionError` visualizer:

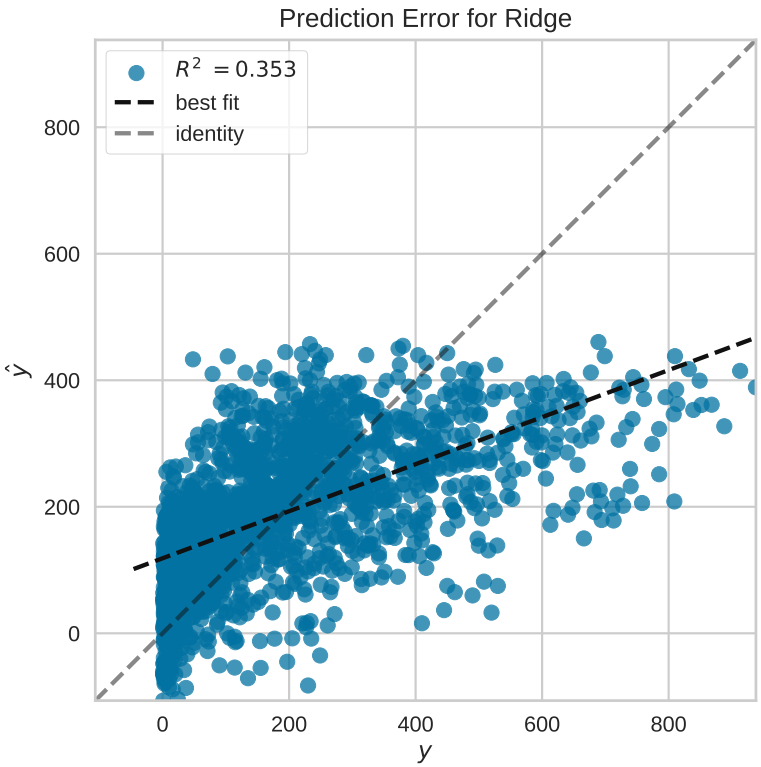



```
from sklearn.linear_model import Ridge
from yellowbrick.regressor import PredictionError

visualizer = PredictionError(Ridge(alpha=3.181))
visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.show()
```

The prediction error visualizer plots the actual (measured) vs. expected (predicted) values against each other. The dotted black line is the 45 degree line that indicates zero error. Like the residuals plot, this allows us to see where error is occurring and in what magnitude.

In this plot, we can see that most of the instance density is less than 200 riders. We may want to try orthogonal matching pursuit or splines to fit a regression that takes into account more regionality. We can also note that that weird topology from the residuals plot seems to be fixed using the Ridge regression, and that there is a bit more balance in our model between large and small values. Potentially the Ridge regularization cured a covariance issue we had between two features. As we move forward in our analysis using other model forms, we can continue to utilize visualizers to quickly compare and see our results.

Hopefully this workflow gives you an idea of how to integrate Visualizers into machine learning with scikit-learn and inspires you to use them in your work and write your own! For additional information on getting started with Yellowbrick, check out the *Model Selection Tutorial*. After that you can get up to speed on specific visualizers detailed in the *Visualizers and API*.



8.2 Model Selection Tutorial

In this tutorial, we are going to look at scores for a variety of Scikit-Learn models and compare them using visual diagnostic tools from Yellowbrick in order to select the best model for our data.

8.2.1 The Model Selection Triple

Discussions of machine learning are frequently characterized by a singular focus on model selection. Be it logistic regression, random forests, Bayesian methods, or artificial neural networks, machine learning practitioners are often quick to express their preference. The reason for this is mostly historical. Though modern third-party machine learning libraries have made the deployment of multiple models appear nearly trivial, traditionally the application and tuning of even one of these algorithms required many years of study. As a result, machine learning practitioners tended to have strong preferences for particular (and likely more familiar) models over others.

However, model selection is a bit more nuanced than simply picking the "right" or "wrong" algorithm. In practice, the workflow includes:

1. selecting and/or engineering the smallest and most predictive feature set
2. choosing a set of algorithms from a model family, and
3. tuning the algorithm hyperparameters to optimize performance.

The **model selection triple** was first described in a 2015 SIGMOD paper by Kumar et al. In their paper, which concerns the development of next-generation database systems built to anticipate predictive modeling, the authors cogently express that such systems are badly needed due to the highly experimental nature of machine learning in practice. "Model

selection," they explain, "is iterative and exploratory because the space of [model selection triples] is usually infinite, and it is generally impossible for analysts to know a priori which [combination] will yield satisfactory accuracy and/or insights."

Recently, much of this workflow has been automated through grid search methods, standardized APIs, and GUI-based applications. In practice, however, human intuition and guidance can more effectively hone in on quality models than exhaustive search. By visualizing the model selection process, data scientists can steer towards final, explainable models and avoid pitfalls and traps.

The Yellowbrick library is a diagnostic visualization platform for machine learning that allows data scientists to steer the model selection process. Yellowbrick extends the Scikit-Learn API with a new core object: the Visualizer. Visualizers allow visual models to be fit and transformed as part of the Scikit-Learn Pipeline process, providing visual diagnostics throughout the transformation of high dimensional data.

8.2.2 About the Data

This tutorial uses the mushrooms data from the Yellowbrick *Example Datasets* module. Our objective is to predict if a mushroom is poisonous or edible based on its characteristics.

Note: The YB version of the mushrooms data differs from the mushroom dataset from the UCI Machine Learning Repository. The Yellowbrick version has been deliberately modified to make modeling a bit more of a challenge.

The data include descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each species was identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended (this latter class was combined with the poisonous one).

Our data contains information for 3 nominally valued attributes and a target value from 8124 instances of mushrooms (4208 edible, 3916 poisonous).

Let's load the data:

```
from yellowbrick.datasets import load_mushroom

X, y = load_mushroom()
print(X[:5]) # inspect the first five rows
```

```
        shape  surface     color
0      convex   smooth    yellow
1        bell   smooth     white
2      convex    scaly     white
3      convex   smooth      gray
4      convex    scaly    yellow
```

8.2.3 Feature Extraction

Our data, including the target, is categorical. We will need to change these values to numeric ones for machine learning. In order to extract this from the dataset, we'll have to use scikit-learn transformers to transform our input dataset into something that can be fit to a model. Luckily, scikit-learn does provide transformers for converting categorical labels into numeric integers: sklearn.preprocessing.LabelEncoder and sklearn.preprocessing.OneHotEncoder.

We'll use a combination of scikit-learn's `Pipeline` object (here's a great post on using pipelines by Zac Stewart), `OneHotEncoder`, and `LabelEncoder`:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, LabelEncoder

# Label-encode targets before modeling
y = LabelEncoder().fit_transform(y)

# One-hot encode columns before modeling
model = Pipeline([
 ('one_hot_encoder', OneHotEncoder()),
 ('estimator', estimator)
])
```

8.2.4 Modeling and Evaluation

Common metrics for evaluating classifiers

Precision is the number of correct positive results divided by the number of all positive results (e.g. *How many of the mushrooms we predicted would be edible actually were?*).

Recall is the number of correct positive results divided by the number of positive results that should have been returned (e.g. *How many of the mushrooms that were poisonous did we accurately predict were poisonous?*).

The **F1 score** is a measure of a test's accuracy. It considers both the precision and the recall of the test to compute the score. The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst at 0.

```
precision = true positives / (true positives + false positives)

recall = true positives / (false negatives + true positives)

F1 score = 2 * ((precision * recall) / (precision + recall))
```

Now we're ready to make some predictions!

Let's build a way to evaluate multiple estimators – first using traditional numeric scores (which we'll later compare to some visual diagnostics from the Yellowbrick library).

```
from sklearn.metrics import f1_score
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC, NuSVC, SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.linear_model import LogisticRegressionCV, LogisticRegression, SGDClassifier
from sklearn.ensemble import BaggingClassifier, ExtraTreesClassifier,␣
↪RandomForestClassifier


models = [
    SVC(gamma='auto'), NuSVC(gamma='auto'), LinearSVC(),
    SGDClassifier(max_iter=100, tol=1e-3), KNeighborsClassifier(),
    LogisticRegression(solver='lbfgs'), LogisticRegressionCV(cv=3),
    BaggingClassifier(), ExtraTreesClassifier(n_estimators=300),
    RandomForestClassifier(n_estimators=300)
```

(continues on next page)

(continued from previous page)

```
]


def score_model(X, y, estimator, **kwargs):
    """
    Test various estimators.
    """
    y = LabelEncoder().fit_transform(y)
    model = Pipeline([
        ('one_hot_encoder', OneHotEncoder()),
        ('estimator', estimator)
    ])

    # Instantiate the classification model and visualizer
    model.fit(X, y, **kwargs)

    expected  = y
    predicted = model.predict(X)

    # Compute and return F1 (harmonic mean of precision and recall)
    print("{}: {}".format(estimator.__class__.__name__, f1_score(expected, predicted)))

for model in models:
    score_model(X, y, model)
```

```
SVC: 0.6624286455630514
NuSVC: 0.6726016476215785
LinearSVC: 0.6583804143126177
SGDClassifier: 0.5582697992842696
KNeighborsClassifier: 0.6581185045215279
LogisticRegression: 0.6580434509606933
LogisticRegressionCV: 0.6583804143126177
BaggingClassifier: 0.6879633373770051
ExtraTreesClassifier: 0.6871364804544838
RandomForestClassifier: 0.687643484132343
```

Preliminary Model Evaluation

Based on the results from the F1 scores above, which model is performing the best?

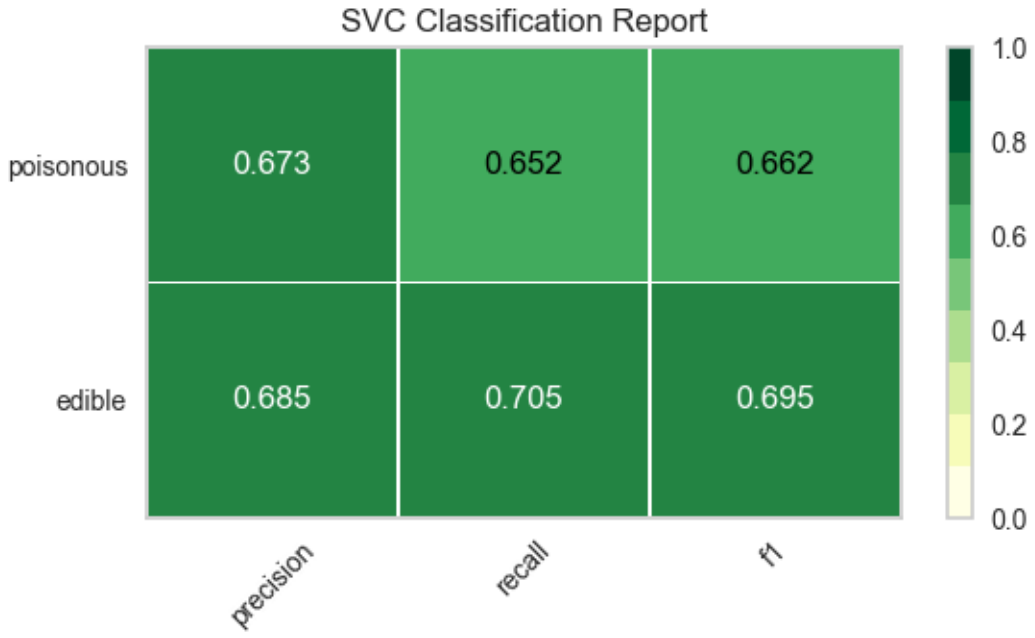
8.2.5 Visual Model Evaluation

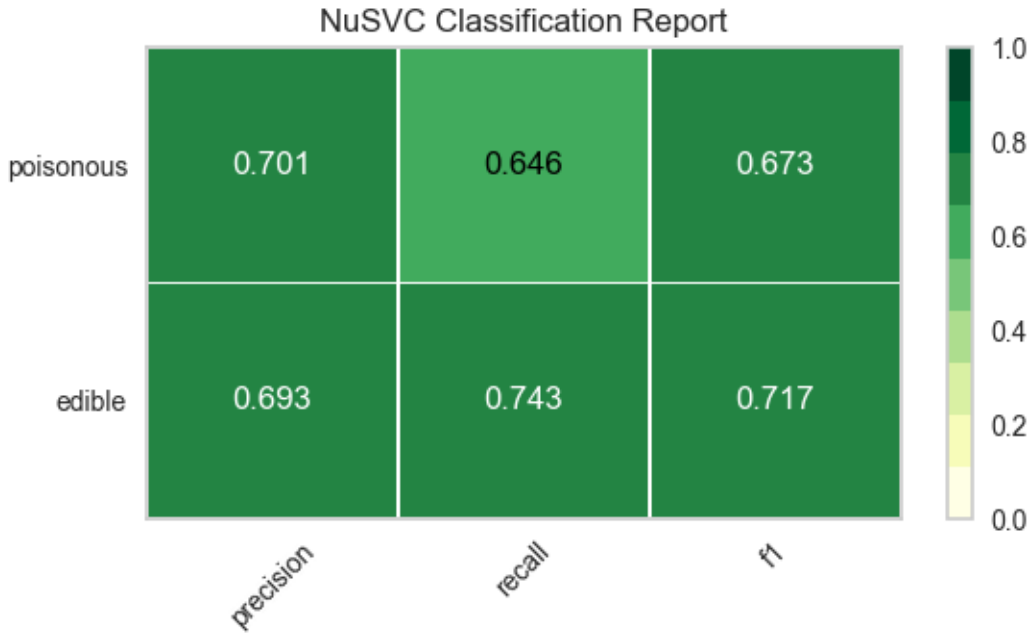
Now let's refactor our model evaluation function to use Yellowbrick's `ClassificationReport` class, a model visualizer that displays the precision, recall, and F1 scores. This visual model analysis tool integrates numerical scores as well as color-coded heatmaps in order to support easy interpretation and detection, particularly the nuances of Type I and Type II error, which are very relevant (lifesaving, even) to our use case!

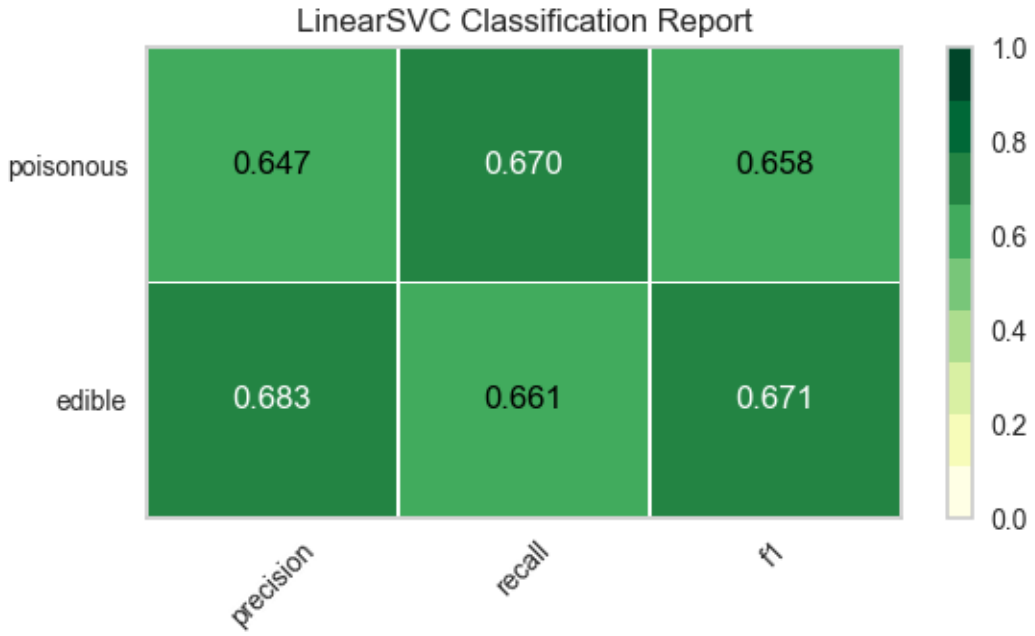
Type I error (or a **"false positive"**) is detecting an effect that is not present (e.g. determining a mushroom is poisonous when it is in fact edible).

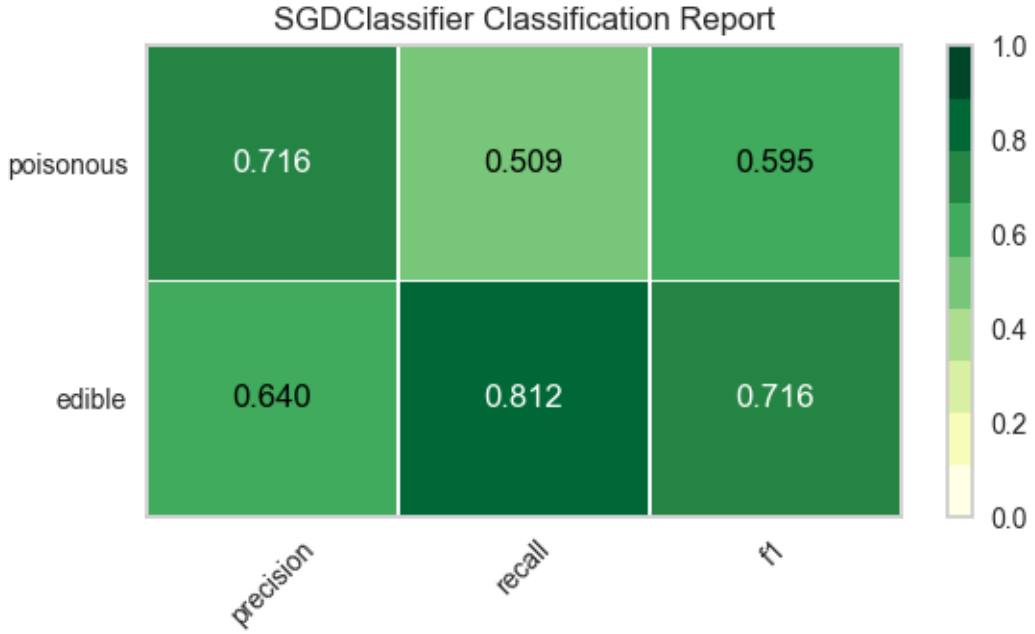
Type II error (or a **"false negative"**) is failing to detect an effect that is present (e.g. believing a mushroom is edible when it is in fact poisonous).

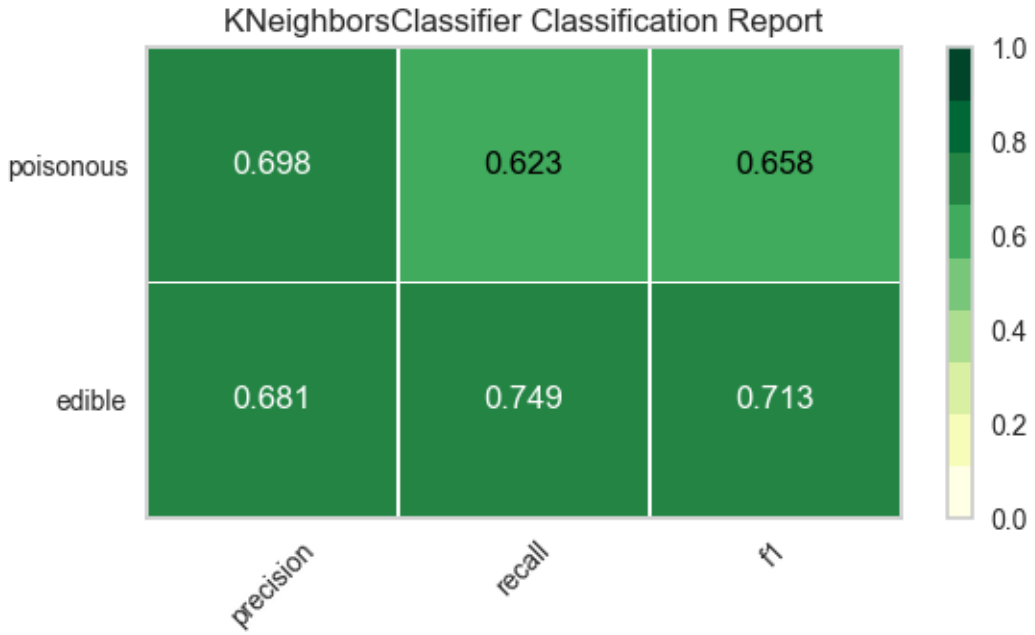
```
from sklearn.pipeline import Pipeline
from yellowbrick.classifier import ClassificationReport


def visualize_model(X, y, estimator, **kwargs):
    """
    Test various estimators.
    """
    y = LabelEncoder().fit_transform(y)
    model = Pipeline([
        ('one_hot_encoder', OneHotEncoder()),
        ('estimator', estimator)
    ])

    # Instantiate the classification model and visualizer
    visualizer = ClassificationReport(
        model, classes=['edible', 'poisonous'],
        cmap="YlGn", size=(600, 360), **kwargs
    )
    visualizer.fit(X, y)
    visualizer.score(X, y)
    visualizer.show()

for model in models:
    visualize_model(X, y, model)
```



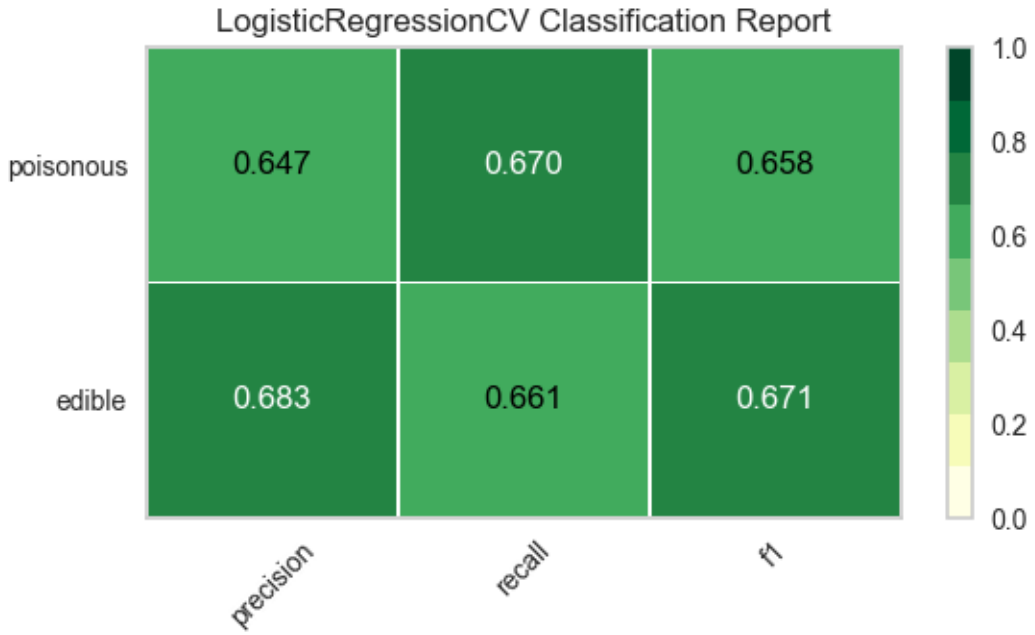

NuSVC Classification Report
poisonous
edible
0.701
0.646
0.673
0.693
0.743
0.717
precision
recall
f1
1.0
0.8
0.6
0.4
0.2
0.0

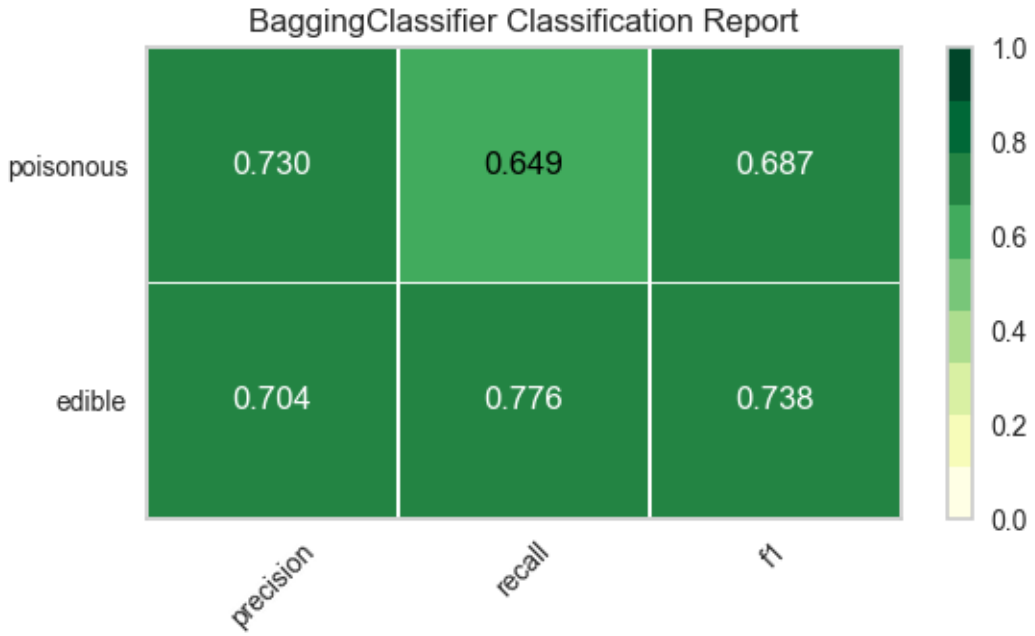

LinearSVC Classification Report
poisonous
edible
0.647
0.670
0.658
0.683
0.661
0.671
precision
recall
f1
1.0
0.8
0.6
0.4
0.2
0.0


SGDClassifier Classification Report
poisonous
edible
0.716
0.509
0.595
0.640
0.812
0.716
precision
recall
f1
1.0
0.8
0.6
0.4
0.2
0.0

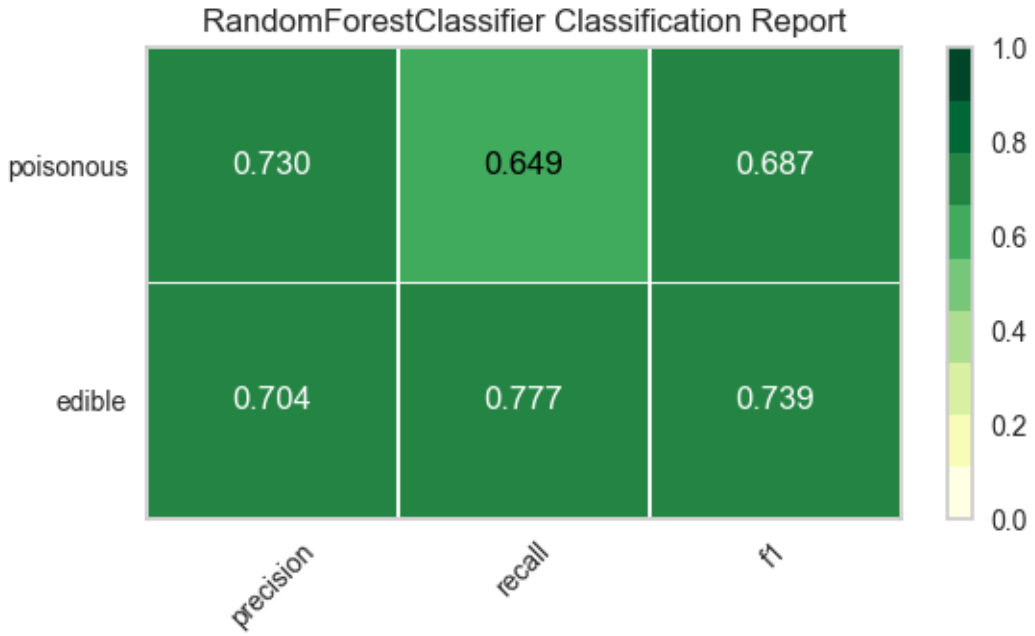

KNeighborsClassifier Classification Report
poisonous
edible
0.698
0.623
0.658
0.681
0.749
0.713
precision
recall
f1
1.0
0.8
0.6
0.4
0.2
0.0


LogisticRegression Classification Report
poisonous
edible
0.647
0.669
0.658
0.682
0.661
0.671
precision
recall
f1
1.0
0.8
0.6
0.4
0.2
0.0


LogisticRegressionCV Classification Report
poisonous
edible
0.647
0.670
0.658
0.683
0.661
0.671
precision
recall
f1
1.0
0.8
0.6
0.4
0.2
0.0


BaggingClassifier Classification Report
poisonous
edible
0.730
0.649
0.687
0.704
0.776
0.738
precision
recall
f1
1.0
0.8
0.6
0.4
0.2
0.0


ExtraTreesClassifier Classification Report
poisonous
edible
0.730
0.649
0.687
0.704
0.777
0.739
precision
recall
f1
1.0
0.8
0.6
0.4
0.2
0.0



8.2.6 Reflection

1. Which model seems best now? Why?
2. Which is most likely to save your life?
3. How is the visual model evaluation experience different from numeric model evaluation?

8.3 Visualizers and API

Welcome to the API documentation for Yellowbrick! This section contains a complete listing of the currently available, production-ready visualizers along with code examples of how to use them. You may use the following links to navigate to the reference material for each visualization.

8.3.1 Example Datasets

Yellowbrick hosts several datasets wrangled from the UCI Machine Learning Repository to present the examples used throughout this documentation. These datasets are hosted in our CDN and must be downloaded for use. Typically, when a user calls one of the data loader functions, e.g. `load_bikeshare()` the data is automatically downloaded if it's not already on the user's computer. However, for development and testing, or if you know you will be working without internet access, it might be easier to simply download all the data at once.

The data downloader script can be run as follows:

```
$ python -m yellowbrick.download
```

This will download all of the data to the `fixtures` directory inside of the Yellowbrick site packages. You can specify the location of the download either as an argument to the downloader script (use `--help` for more details) or by setting

the `$YELLOWBRICK_DATA` environment variable. This is the preferred mechanism because this will also influence how data is loaded in Yellowbrick.

Note: Developers who have downloaded data from Yellowbrick versions earlier than v1.0 may experience some problems with the older data format. If this occurs, you can clear out your data cache by running `python -m yellowbrick.download --cleanup`. This will remove old datasets and download the new ones. You can also use the `--no-download` flag to simply clear the cache without re-downloading data. Users who are having difficulty with datasets can also use this or they can uninstall and reinstall Yellowbrick using `pip`.

Once you have downloaded the example datasets, you can load and use them as follows:

```
from yellowbrick.datasets import load_bikeshare

X, y = load_bikeshare() # returns features and targets for the bikeshare dataset
```

Each dataset has a `README.md` with detailed information about the data source, attributes, and target as well as other metadata. To get access to the metadata or to more precisely control your data access you can return the dataset directly from the loader as follows:

```
dataset = load_bikeshare(return_dataset=True)
print(dataset.README)

df = dataset.to_dataframe()
df.head()
```

Datasets

Unless otherwise specified, most of the documentation examples currently use one or more of the listed datasets. Here is a complete listing of all datasets in Yellowbrick and the analytical tasks with which they are most commonly associated:

- *Bikeshare*: suitable for regression
- *Concrete*: suitable for regression
- *Credit*: suitable for classification/clustering
- *Energy*: suitable for regression
- *Game*: suitable for multi-class classification
- *Hobbies*: suitable for text analysis/classification
- *Mushroom*: suitable for classification/clustering
- *Occupancy*: suitable for classification
- *Spam*: suitable for binary classification
- *Walking*: suitable for time series analysis/clustering
- *NFL*: suitable for clustering

Bikeshare

This dataset contains the hourly and daily count of rental bikes between years 2011 and 2012 in Capital bikeshare system with the corresponding weather and seasonal information.

Samples total	17379
Dimensionality	12
Features	real, positive
Targets	ints, 1-977
Task(s)	regression

Description

Bike sharing systems are new generation of traditional bike rentals where whole process from membership, rental and return back has become automatic. Through these systems, user is able to easily rent a bike from a particular position and return back at another position. Currently, there are about over 500 bike-sharing programs around the world which is composed of over 500 thousands bicycles. Today, there exists great interest in these systems due to their important role in traffic, environmental and health issues.

Apart from interesting real world applications of bike sharing systems, the characteristics of data being generated by these systems make them attractive for the research. Opposed to other transport services such as bus or subway, the duration of travel, departure and arrival position is explicitly recorded in these systems. This feature turns bike sharing system into a virtual sensor network that can be used for sensing mobility in the city. Hence, it is expected that most of important events in the city could be detected via monitoring these data.

Citation

Downloaded from the UCI Machine Learning Repository on May 4, 2017.

Fanaee-T, Hadi, and Gama, Joao, 'Event labeling combining ensemble detectors and background knowledge', Progress in Artificial Intelligence (2013): pp. 1-15, Springer Berlin Heidelberg

Loader

`yellowbrick.datasets.loaders.`**`load_bikeshare`**(*data_home=None*, *return_dataset=False*)

Loads the bike sharing univariate dataset that is well suited to regression tasks. The dataset contains 17379 instances with 12 integer and real valued attributes and a continuous target.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

return_dataset
[bool, default=False] Return the raw dataset object instead of X and y numpy arrays to get access to alternative targets, extra features, content and meta.

Returns

X
[array-like with shape (n_instances, n_features) if return_dataset=False] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,) if return_dataset=False] A pandas Series or numpy array describing the target vector.

dataset
[Dataset instance if return_dataset=True] The Yellowbrick Dataset object provides an interface to accessing the data in a variety of formats as well as associated metadata and content.

Concrete

Concrete is the most important material in civil engineering. The concrete compressive strength is a highly nonlinear function of age and ingredients.

Samples total	1030
Dimensionality	9
Features	real
Targets	float, 2.3-82.6
Task(s)	regression

Description

Given are the variable name, variable type, the measurement unit and a brief description. The concrete compressive strength is the regression problem. The order of this listing corresponds to the order of numerals along the rows of the database.

Citation

Downloaded from the UCI Machine Learning Repository on October 13, 2016.

Yeh, I-C. "Modeling of strength of high-performance concrete using artificial neural networks." Cement and Concrete research 28.12 (1998): 1797-1808.

Loader

`yellowbrick.datasets.loaders.`**`load_concrete`**(*data_home=None*, *return_dataset=False*)

Loads the concrete multivariate dataset that is well suited to regression tasks. The dataset contains 1030 instances and 8 real valued attributes with a continuous target.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

return_dataset
[bool, default=False] Return the raw dataset object instead of X and y numpy arrays to get access to alternative targets, extra features, content and meta.

Returns

X
[array-like with shape (n_instances, n_features) if return_dataset=False] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,) if return_dataset=False] A pandas Series or numpy array describing the target vector.

dataset
[Dataset instance if return_dataset=True] The Yellowbrick Dataset object provides an interface to accessing the data in a variety of formats as well as associated metadata and content.

Credit

This research aimed at the case of customers' default payments in Taiwan and compares the predictive accuracy of probability of default among six data mining methods.

Samples total	30000
Dimensionality	24
Features	real, int
Targets	int, 0 or 1
Task(s)	classification

Description

This research aimed at the case of customers' default payments in Taiwan and compares the predictive accuracy of probability of default among six data mining methods. From the perspective of risk management, the result of predictive accuracy of the estimated probability of default will be more valuable than the binary result of classification - credible or not credible clients. Because the real probability of default is unknown, this study presented the novel "Sorting Smoothing Method" to estimate the real probability of default. With the real probability of default as the response variable (Y), and the predictive probability of default as the independent variable (X), the simple linear regression result (Y = A + BX) shows that the forecasting model produced by artificial neural network has the highest coefficient of determination; its regression intercept (A) is close to zero, and regression coefficient (B) to one. Therefore, among the six data mining techniques, artificial neural network is the only one that can accurately estimate the real probability of default.

Citation

Downloaded from the UCI Machine Learning Repository on October 13, 2016.

Yeh, I. C., & Lien, C. H. (2009). The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. Expert Systems with Applications, 36(2), 2473-2480.

Loader

`yellowbrick.datasets.loaders.load_credit`(*data_home=None*, *return_dataset=False*)

Loads the credit multivariate dataset that is well suited to binary classification tasks. The dataset contains 30000 instances and 23 integer and real value attributes with a discrete target.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

return_dataset
[bool, default=False] Return the raw dataset object instead of X and y numpy arrays to get access to alternative targets, extra features, content and meta.

Returns

X
[array-like with shape (n_instances, n_features) if return_dataset=False] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,) if return_dataset=False] A pandas Series or numpy array describing the target vector.

dataset
[Dataset instance if return_dataset=True] The Yellowbrick Dataset object provides an interface to accessing the data in a variety of formats as well as associated metadata and content.

Energy

The dataset was created by Angeliki Xifara (angxifara '@' gmail.com, Civil/Structural Engineer) and was processed by Athanasios Tsanas (tsanasthanasis '@' gmail.com, Oxford Centre for Industrial and Applied Mathematics, University of Oxford, UK).

Samples total	768
Dimensionality	8
Features	real, int
Targets	float, 6.01-43.1
Task(s)	regression, classification

Description

We perform energy analysis using 12 different building shapes simulated in Ecotect. The buildings differ with respect to the glazing area, the glazing area distribution, and the orientation, amongst other parameters. We simulate various settings as functions of the afore-mentioned characteristics to obtain 768 building shapes. The dataset comprises 768 samples and 8 features, aiming to predict two real valued responses. It can also be used as a multi-class classification problem if the response is rounded to the nearest integer.

Example

The energy dataset contains a multi-target supervised dataset for both the heating and the cooling load of buildings. By default only the heating load is returned for most examples. To perform a multi-target regression, simply access the dataframe and select both the heating and cooling load columns as follows:

```
from yellowbrick.datasets import load_energy
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split as tts

features = [
   "relative compactness",
   "surface area",
   "wall area",
   "roof area",
   "overall height",
   "orientation",
   "glazing area",
   "glazing area distribution",
]
target = ["heating load", "cooling load"]

df = load_energy(return_dataset=True).to_dataframe()
X, y = df[features], df[target]

X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2)

model = RandomForestRegressor().fit(X_train, y_train)
model.score(X_test, y_test)
```

Note that not all regressors support multi-target regression, one simple strategy in this case is to use a `sklearn.multioutput.MultiOutputRegressor`, which fits an estimator for each target.

Citation

Downloaded from the UCI Machine Learning Repository March 23, 2015.

A. Tsanas, A. Xifara: 'Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools', Energy and Buildings, Vol. 49, pp. 560-567, 2012

For further details on the data analysis methodology:

A. Tsanas, 'Accurate telemonitoring of Parkinson's disease symptom severity using nonlinear speech signal processing and statistical machine learning', D.Phil. thesis, University of Oxford, 2012

Loader

yellowbrick.datasets.loaders.**load_energy**(*data_home=None*, *return_dataset=False*)

Loads the energy multivariate dataset that is well suited to multi-output regression and classification tasks. The dataset contains 768 instances and 8 real valued attributes with two continous targets.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

return_dataset
[bool, default=False] Return the raw dataset object instead of X and y numpy arrays to get access to alternative targets, extra features, content and meta.

Returns

X
[array-like with shape (n_instances, n_features) if return_dataset=False] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,) if return_dataset=False] A pandas Series or numpy array describing the target vector.

dataset
[Dataset instance if return_dataset=True] The Yellowbrick Dataset object provides an interface to accessing the data in a variety of formats as well as associated metadata and content.

Game

The dataset was created and donated to the UCI ML Repository by John Tromp (tromp '@' cwi.nl).

Samples total	67557
Dimensionality	42
Features	categorical
Targets	str: {"win", "loss", "draw"}
Task(s)	classification

Description

This database contains all legal 8-ply positions in the game of connect-4 in which neither player has won yet, and in which the next move is not forced.

The symbol x represents the first player; o the second. The dataset contains the state of the game by representing each position in a 6x7 grid board. The outcome class is the game theoretical value for the first player.

Example

Note that to use the game dataset the categorical data in the features array must be encoded numerically. There are a number of numeric encoding mechanisms such as the `sklearn.preprocessing.OrdinalEncoder` or the `sklearn.preprocessing.OneHotEncoder` that may be used as follows:

```
from sklearn.preprocessing import OneHotEncoder
from yellowbrick.datasets import load_game

X, y = load_game()
X = OneHotEncoder().fit_transform(X)
```

Citation

Downloaded from the UCI Machine Learning Repository on May 4, 2017.

Loader

`yellowbrick.datasets.loaders.`**load_game**(*data_home=None*, *return_dataset=False*)

Load the Connect-4 game multivariate and spatial dataset that is well suited to multiclass classification tasks. The dataset contains 67557 instances with 42 categorical attributes and a discrete target.

Note that the game data is stored with categorical features that need to be numerically encoded before use with scikit-learn estimators. We recommend the use of the `sklearn.preprocessing.OneHotEncoder` for this task and to develop a `Pipeline` using this dataset.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

return_dataset
[bool, default=False] Return the raw dataset object instead of X and y numpy arrays to get access to alternative targets, extra features, content and meta.

Returns

X
[array-like with shape (n_instances, n_features) if return_dataset=False] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,) if return_dataset=False] A pandas Series or numpy array describing the target vector.

dataset
[Dataset instance if return_dataset=True] The Yellowbrick Dataset object provides an interface to accessing the data in a variety of formats as well as associated metadata and content.

Hobbies

The Baleen hobbies corpus contains 448 files in 5 categories.

Samples total	448
Dimensionality	23738
Features	strings (tokens)
Targets	str: {"books", "cinema", "cooking", "gaming", "sports"}
Task(s)	classification, clustering

Description

The hobbies corpus is a text corpus wrangled from the Baleen RSS Corpus in order to enable students and readers to practice different techniques in Natural Language Processing. For more information see Applied Text Analysis with Python: Enabling Language-Aware Data Products with Machine Learning and the associated code repository. It is structured as:

Documents and File Size

- books: 72 docs (4.1MiB)
- cinema: 100 docs (9.2MiB)
- cooking: 30 docs (3.0MiB)
- gaming: 128 docs (8.8MiB)
- sports: 118 docs (15.9MiB)

Document Structure

Overall:

- 7,420 paragraphs (16.562 mean paragraphs per file)
- 14,251 sentences (1.921 mean sentences per paragraph).

By Category:

- books: 844 paragraphs and 2,030 sentences
- cinema: 1,475 paragraphs and 3,047 sentences

- cooking: 1,190 paragraphs and 2,425 sentences
- gaming: 1,802 paragraphs and 3,373 sentences
- sports: 2,109 paragraphs and 3,376 sentences

Words and Vocabulary

Word count of 288,520 with a vocabulary of 23,738 (12.154 lexical diversity).

- books: 41,851 words with a vocabulary size of 7,838
- cinema: 69,153 words with a vocabulary size of 10,274
- cooking: 37,854 words with a vocabulary size of 5,038
- gaming: 70,778 words with a vocabulary size of 9,120
- sports: 68,884 words with a vocabulary size of 8,028

Example

The hobbies corpus loader returns a `Corpus` object with the raw text associated with the data set. This must be vectorized into a numeric form for use with scikit-learn. For example, you could use the `sklearn.feature_extraction.text.TfidfVectorizer` as follows:

```
from yellowbrick.datasets import load_hobbies

from sklearn.naive_bayes import MultinomialNB
from sklearn.preprocessing import LabelEncoder
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split as tts

corpus = load_hobbies()
X = TfidfVectorizer().fit_transform(corpus.data)
y = LabelEncoder().fit_transform(corpus.target)

X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2)

model = MultinomialNB().fit(X_train, y_train)
model.score(X_test, y_test)
```

For more detail on text analytics and machine learning with scikit-learn, please refer to "Working with Text Data" in the scikit-learn documentation.

Citation

Exported from S3 on: Jan 21, 2017 at 06:42.

Bengfort, Benjamin, Rebecca Bilbro, and Tony Ojeda. Applied Text Analysis with Python: Enabling Language-aware Data Products with Machine Learning. " O'Reilly Media, Inc.", 2018.

Loader

yellowbrick.datasets.loaders.**load_hobbies**(*data_home=None*)

Loads the hobbies text corpus that is well suited to classification, clustering, and text analysis tasks. The dataset contains 448 documents in 5 categories with 7420 paragraphs, 14251 sentences, 288520 words, and a vocabulary of 23738.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

Returns

dataset
[Corpus] The Yellowbrick Corpus object provides an interface to accessing the text documents and metadata associated with the corpus.

Mushroom

From Audobon Society Field Guide; mushrooms described in terms of physical characteristics; classification: poisonous or edible.

Samples total	8124
Dimensionality	4 (reduced from 22)
Features	categorical
Targets	str: {"edible", "poisonous"}
Task(s)	classification

Description

This data set includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family (pp. 500-525). Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom; no rule like "leaflets three, let it be" for Poisonous Oak and Ivy.

Citation

Downloaded from the UCI Machine Learning Repository on February 28, 2017.

Schlimmer, Jeffrey Curtis. "Concept acquisition through representational adjustment." (1987).

Langley, Pat. "Trading off simplicity and coverage in incremental concept learning." Machine Learning Proceedings 1988 (2014): 73.

Duch, Włodzisław, Rafał Adamczak, and Krzysztof Grabczewski. "Extraction of logical rules from training data using backpropagation networks." The 1st Online Workshop on Soft Computing. 1996.

Duch, Wlodzislaw, Rafal Adamczak, and Krzysztof Grabczewski. "Extraction of crisp logical rules using constrained backpropagation networks." (1997).

Loader

`yellowbrick.datasets.loaders.`**`load_mushroom`**(*data_home=None*, *return_dataset=False*)

Loads the mushroom multivariate dataset that is well suited to binary classification tasks. The dataset contains 8123 instances with 3 categorical attributes and a discrete target.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

return_dataset
[bool, default=False] Return the raw dataset object instead of X and y numpy arrays to get access to alternative targets, extra features, content and meta.

Returns

X
[array-like with shape (n_instances, n_features) if return_dataset=False] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,) if return_dataset=False] A pandas Series or numpy array describing the target vector.

dataset
[Dataset instance if return_dataset=True] The Yellowbrick Dataset object provides an interface to accessing the data in a variety of formats as well as associated metadata and content.

Occupancy

Experimental data used for binary classification (room occupancy) from Temperature, Humidity, Light and CO2. Ground-truth occupancy was obtained from time stamped pictures that were taken every minute.

Samples total	20560
Dimensionality	6
Features	real, positive
Targets	int: {1 for occupied, 0 for not occupied}
Task(s)	classification
Samples per class	imbalanced

Description

Three data sets are submitted, for training and testing. Ground-truth occupancy was obtained from time stamped pictures that were taken every minute. For the journal publication, the processing R scripts can be found on GitHub.

Citation

Downloaded from the UCI Machine Learning Repository on October 13, 2016.

Candanedo, Luis M., and Véronique Feldheim. "Accurate occupancy detection of an office room from light, temperature, humidity and CO 2 measurements using statistical learning models." Energy and Buildings 112 (2016): 28-39.

`yellowbrick.datasets.loaders.load_occupancy`(*data_home=None*, *return_dataset=False*)

Loads the occupancy multivariate, time-series dataset that is well suited to binary classification tasks. The dataset contains 20560 instances with 5 real valued attributes and a discrete target.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

return_dataset
[bool, default=False] Return the raw dataset object instead of X and y numpy arrays to get access to alternative targets, extra features, content and meta.

Returns

X
[array-like with shape (n_instances, n_features) if return_dataset=False] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,) if return_dataset=False] A pandas Series or numpy array describing the target vector.

dataset
[Dataset instance if return_dataset=True] The Yellowbrick Dataset object provides an interface to accessing the data in a variety of formats as well as associated metadata and content.

Spam

Classifying Email as Spam or Non-Spam.

Samples total	4601
Dimensionality	57
Features	real, integer
Targets	int: {1 for spam, 0 for not spam}
Task(s)	classification

Description

The "spam" concept is diverse: advertisements for products/web sites, make money fast schemes, chain letters, pornography...

Our collection of spam e-mails came from our postmaster and individuals who had filed spam. Our collection of non-spam e-mails came from filed work and personal e-mails, and hence the word 'george' and the area code '650' are indicators of non-spam. These are useful when constructing a personalized spam filter. One would either have to blind such non-spam indicators or get a very wide collection of non-spam to generate a general purpose spam filter.

Determine whether a given email is spam or not.

~7% misclassification error. False positives (marking good mail as spam) are very undesirable.If we insist on zero false positives in the training/testing set, 20-25% of the spam passed through the filter.

Citation

Downloaded from the UCI Machine Learning Repository on March 23, 2018.

Cranor, Lorrie Faith, and Brian A. LaMacchia. "Spam!." Communications of the ACM 41.8 (1998): 74-83.

Loader

`yellowbrick.datasets.loaders.load_spam`(*data_home=None*, *return_dataset=False*)

Loads the email spam dataset that is weill suited to binary classification and threshold tasks. The dataset contains 4600 instances with 57 integer and real valued attributes and a discrete target.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

return_dataset
[bool, default=False] Return the raw dataset object instead of X and y numpy arrays to get access to alternative targets, extra features, content and meta.

Returns

X
[array-like with shape (n_instances, n_features) if return_dataset=False] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,) if return_dataset=False] A pandas Series or numpy array describing the target vector.

dataset
[Dataset instance if return_dataset=True] The Yellowbrick Dataset object provides an interface to accessing the data in a variety of formats as well as associated metadata and content.

Walking

The dataset collects data from an Android smartphone positioned in the chest pocket. Accelerometer Data are collected from 22 participants walking in the wild over a predefined path. The dataset is intended for Activity Recognition research purposes. It provides challenges for identification and authentication of people using motion patterns. Sampling frequency of the accelerometer: DELAY_FASTEST with network connections disabled.

Samples total	149331
Dimensionality	4
Features	real
Targets	int, 1-22
Task(s)	classification, clustering

Description

In this article, a novel technique for user's authentication and verification using gait as a biometric unobtrusive pattern is proposed. The method is based on a two stages pipeline. First, a general activity recognition classifier is personalized for an specific user using a small sample of her/his walking pattern. As a result, the system is much more selective with respect to the new walking pattern. A second stage verifies whether the user is an authorized one or not. This stage is defined as a one-class classification problem. In order to solve this problem, a four-layer architecture is built around the geometric concept of convex hull. This architecture allows to improve robustness to outliers, modeling non-convex shapes, and to take into account temporal coherence information. Two different scenarios are proposed as validation with two different wearable systems. First, a custom high-performance wearable system is built and used in a free environment. A second dataset is acquired from an Android-based commercial device in a 'wild' scenario with rough terrains, adversarial conditions, crowded places and obstacles. Results on both systems and datasets are very promising, reducing the verification error rates by an order of magnitude with respect to the state-of-the-art technologies.

Citation

Downloaded from the UCI Machine Learning Repository on August 23, 2018.

Casale, Pierluigi, Oriol Pujol, and Petia Radeva. "Personalization and user verification in wearable systems using biometric walking patterns." Personal and Ubiquitous Computing 16.5 (2012): 563-580.

Loader

`yellowbrick.datasets.loaders.load_walking`(*data_home=None*, *return_dataset=False*)

Loads the walking activity dataset that is weill suited to clustering and multi-label classification tasks. The dataset contains multi-variate time series data with 149,332 real valued measurements across 22 unique walkers.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

return_dataset
[bool, default=False] Return the raw dataset object instead of X and y numpy arrays to get access to alternative targets, extra features, content and meta.

Returns

X
[array-like with shape (n_instances, n_features) if return_dataset=False] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,) if return_dataset=False] A pandas Series or numpy array describing the target vector.

dataset
[Dataset instance if return_dataset=True] The Yellowbrick Dataset object provides an interface to accessing the data in a variety of formats as well as associated metadata and content.

NFL

This dataset is comprised of statistics on all eligible receivers from the 2018 NFL regular season.

Samples total	494
Dimensionality	20
Features	str, int
Targets	N/A
Task(s)	clustering

Description

The dataset consists of an aggregate of all relevant statistics for eligible receivers that played in at least 1 game and had at least 1 target throughout the season. This is not limited to players specifically designated as wide-receivers, but may include other positions such as running-backs and tight-ends.

Citation

Redistributed with the permission of Sports Reference LLC on June 11, 2019 via email.

Sports Reference LLC, "2018 NFL Receiving," Pro-Football-Reference.com - Pro Football Statistics and History. [Online]. Available here. [Accessed: 18-Jun-2019]

Loader

`yellowbrick.datasets.loaders.load_nfl`(*data_home=None*, *return_dataset=False*)

Loads the football receivers dataset that is well suited to clustering tasks. The dataset contains 494 instances with 28 integer, real valued, and categorical attributes and a discrete target.

The Yellowbrick datasets are hosted online and when requested, the dataset is downloaded to your local computer for use. Note that if the dataset hasn't been downloaded before, an Internet connection is required. However, if the data is cached locally, no data will be downloaded. Yellowbrick checks the known signature of the dataset with the data downloaded to ensure the download completes successfully.

Datasets are stored alongside the code, but the location can be specified with the `data_home` parameter or the `$YELLOWBRICK_DATA` envvar.

Parameters

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from `$YELLOWBRICK_DATA` or the default returned by `get_data_home`.

return_dataset
[bool, default=False] Return the raw dataset object instead of X and y numpy arrays to get access to alternative targets, extra features, content and meta.

Returns

X
[array-like with shape (n_instances, n_features) if return_dataset=False] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,) if return_dataset=False] A pandas Series or numpy array describing the target vector.

dataset
[Dataset instance if return_dataset=True] The Yellowbrick Dataset object provides an interface to accessing the data in a variety of formats as well as associated metadata and content.

Yellowbrick has included these datasets in our package for demonstration purposes only. The datasets have been repackaged with the permission of the authors or in accordance with the terms of use of the source material. If you use a Yellowbrick wrangled dataset, please be sure to cite the original author.

API Reference

By default, the dataset loaders return a features table, `X`, and a target vector `y` when called. If the user has Pandas installed, the data types will be a `pd.DataFrame` and `pd.Series` respectively, otherwise the data will be returned as numpy arrays. This functionality ensures that the primary use of the datasets, to follow along with the documentation examples, is as simple as possible. However, advanced users may note that there does exist an underlying object with advanced functionality that can be accessed as follows:

```
dataset = load_occupancy(return_dataset=True)
```

There are two basic types of dataset, the `Dataset` which is used for *tabular data* loaded from a CSV and the `Corpus`, used to load *text corpora* from disk. Both types of dataset give access to a readme file, a citation in BibTex format, json metadata that describe the fields and target, and different data types associated with the underlying datasset. Both objects are also responsible for locating the dataset on disk and downloading it safely if it doesn't exist yet. For more on how Yellowbrick downloads and stores data, please see *Local Storage*.

Tabular Data

Most example datasets are returned as tabular data structures loaded either from a .csv file (using Pandas) or from dtype encoded .npz file to ensure correct numpy arrays are returned. The `Dataset` object loads the data from these stored files, preferring to use Pandas if it is installed. It then uses metadata to slice the DataFrame into a feature matrix and target array. Using the dataset directly provides extra functionality, and can be retrieved as follows:

```
from yellowbrick.datasets import load_concrete
dataset = load_concrete(return_dataset=True)
```

For example if you wish to get the raw data frame you can do so as follows:

```
df = dataset.to_dataframe()
df.head()
```

There may be additional columns in the DataFrame that were part of the original dataset but were excluded from the featureset. For example, the *energy dataset* contains two targets, the heating and the cooling load, but only the heating load is returned by default. The api documentation that follows describes in details the metadata properties and other functionality associated with the `Dataset`:

class yellowbrick.datasets.base.**Dataset**(*name*, *url=None*, *signature=None*, *data_home=None*)

Bases: `BaseDataset`

Datasets contain a reference to data on disk and provide utilities for quickly loading files and objects into a variety of formats. The most common use of the Dataset object is to load example datasets provided by Yellowbrick to run the examples in the documentation.

The dataset by default will return the data as a numpy array, however if Pandas is installed, it is possible to access the data as a DataFrame and Series object. In either case, the data is represented by a features table, X and a target vector, y.

Parameters

name
[str] The name of the dataset; should either be a folder in data home or specified in the yellowbrick.datasets.DATASETS variable. This name is used to perform all lookups and identify the dataset externally.

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from YELLOWBRICK_DATA or the default returned by `get_data_home`.

url
[str, optional] The web location where the archive file of the dataset can be downloaded from.

signature
[str, optional] The signature of the data archive file, used to verify that the latest version of the data has been downloaded and that the download hasn't been corrupted or modified in anyway.

`property README`

Returns the contents of the README.md file that describes the dataset in detail and contains attribution information.

`property citation`

Returns the contents of the citation.bib file that describes the source and provenance of the dataset or to cite for academic work.

`contents()`

Contents returns a list of the files in the data directory.

`download`(*replace=False*)

Download the dataset from the hosted Yellowbrick data store and save it to the location specified by `get_data_home`. The downloader verifies the download completed successfully and safely by comparing the expected signature with the SHA 256 signature of the downloaded archive file.

Parameters

replace
[bool, default: False] If the data archive already exists, replace the dataset. If this is False and the dataset exists, an exception is raised.

`property meta`

Returns the contents of the meta.json file that describes important attributes about the dataset and modifies the behavior of the loader.

`to_data()`

Returns the data contained in the dataset as X and y where X is the features matrix and y is the target vector. If pandas is installed, the data will be returned as DataFrame and Series objects. Otherwise, the data will be returned as two numpy arrays.

Returns

X
[array-like with shape (n_instances, n_features)] A pandas DataFrame or numpy array describing the instance features.

y
[array-like with shape (n_instances,)] A pandas Series or numpy array describing the target vector.

`to_dataframe()`

Returns the entire dataset as a single pandas DataFrame.

Returns

df
[DataFrame with shape (n_instances, n_columns)] A pandas DataFrame containing the complete original data table including all targets (specified by the meta data) and all features (including those that might have been filtered out).

`to_numpy()`

Returns the dataset as two numpy arrays: X and y.

Returns

X
[array-like with shape (n_instances, n_features)] A numpy array describing the instance features.

y
[array-like with shape (n_instances,)] A numpy array describing the target vector.

`to_pandas()`

Returns the dataset as two pandas objects: X and y.

Returns

X
[DataFrame with shape (n_instances, n_features)] A pandas DataFrame containing feature data and named columns.

y
[Series with shape (n_instances,)] A pandas Series containing target data and an index that matches the feature DataFrame index.

Text Corpora

Yellowbrick supports many text-specific machine learning visualizations in the *yellowbrick.text* module. To facilitate these examples and show an end-to-end visual diagnostics workflow that includes text preprocessing, Yellowbrick supports a `Corpus` dataset loader that provides access to raw text data from individual documents. Most notably used with the *hobbies corpus*, a collection of blog posts from different topics that can be used for text classification tasks.

A text corpus is composed of individual documents that are stored on disk in a directory structure that also identifies document relationships. The file name of each document is a unique file ID (e.g. the MD5 hash of its contents). For example, the hobbies corpus is structured as follows:

```
data/hobbies
├── README.md
└── books
|   ├── 56d62a53c1808113ffb87f1f.txt
|   └── 5745a9c7c180810be6efd70b.txt
└── cinema
|   ├── 56d629b5c1808113ffb87d8f.txt
|   └── 57408e5fc180810be6e574c8.txt
└── cooking
|   ├── 56d62b25c1808113ffb8813b.txt
|   └── 573f0728c180810be6e2575c.txt
└── gaming
|   ├── 56d62654c1808113ffb87938.txt
|   └── 574585d7c180810be6ef7ffc.txt
└── sports
    ├── 56d62adec1808113ffb88054.txt
    └── 56d70f17c180810560aec345.txt
```

Unlike the `Dataset`, corpus dataset loaders do not return `X` and `y` specially prepared for machine learning. Instead, these loaders return a `Corpus` object, which can be used to get a more detailed view of the dataset. For example, to list the unique categories in the corpus, you would access the `labels` property as follows:

```
from yellowbrick.datasets import load_hobbies

corpus = load_hobbies()
corpus.labels
```

Addtionally, you can access the list of the absolute paths of each file, which allows you to read individual documents or to use scikit-learn utilties that read the documents streaming one at a time rather than loading them into memory all at once.

```
with open(corpus.files[8], 'r') as f:
    print(f.read())
```

To get the raw text data and target labels, use the `data` and `target` properties.

```
X, y = corpus.data, corpus.target
```

For more details on the other metadata properties associated with the `Corpus`, please refer to the API reference below. For more detail on text analytics and machine learning with scikit-learn, please refer to "Working with Text Data" in the scikit-learn documentation.

class yellowbrick.datasets.base.**Corpus**(*name*, *url=None*, *signature=None*, *data_home=None*)

Bases: `BaseDataset`

Corpus datasets contain a reference to documents on disk and provide utilities for quickly loading text data for use in machine learning workflows. The most common use of the corpus is to load the text analysis examples from the Yellowbrick documentation.

Parameters

name
[str] The name of the corpus; should either be a folder in data home or specified in the yellowbrick.datasets.DATASETS variable. This name is used to perform all lookups and identify the corpus externally.

data_home
[str, optional] The path on disk where data is stored. If not passed in, it is looked up from YELLOWBRICK_DATA or the default returned by `get_data_home`.

url
[str, optional] The web location where the archive file of the corpus can be downloaded from.

signature
[str, optional] The signature of the data archive file, used to verify that the latest version of the data has been downloaded and that the download hasn't been corrupted or modified in anyway.

property README

Returns the contents of the README.md file that describes the dataset in detail and contains attribution information.

property citation

Returns the contents of the citation.bib file that describes the source and provenance of the dataset or to cite for academic work.

contents()

Contents returns a list of the files in the data directory.

property data

Read all of the documents from disk into an in-memory list.

download(*replace=False*)

Download the dataset from the hosted Yellowbrick data store and save it to the location specified by `get_data_home`. The downloader verifies the download completed successfully and safely by comparing the expected signature with the SHA 256 signature of the downloaded archive file.

Parameters

replace
[bool, default: False] If the data archive already exists, replace the dataset. If this is False and the dataset exists, an exception is raised.

property files

Returns the list of file names for all documents.

property labels

Return the unique labels assigned to the documents.

property meta

Returns the contents of the meta.json file that describes important attributes about the dataset and modifies the behavior of the loader.

property root

Discovers and caches the root directory of the corpus.

property target

Returns the label associated with each item in data.

Local Storage

Yellowbrick datasets are stored in a compressed format in the cloud to ensure that the install process is as streamlined and lightweight as possible. When you request a dataset via the loader module, Yellowbrick checks to see if it has been downloaded already, and if not, it downloads it to your local disk.

By default the dataset is stored, uncompressed, in the `site-packages` folder of your Python installation alongside the Yellowbrick code. This means that if you install Yellowbrick in multiple virtual environments, the datasets will be downloaded multiple times in each environment.

To cleanup downloaded datasets, you may use the download module as a command line tool. Note, however, that this will only cleanup the datasets in the yellowbrick package that is on the `$PYTHON_PATH` of the environment you're currently in.

```
$ python -m yellowbrick.download --cleanup --no-download
```

Alternatively, because the data is stored in the same directory as the code, you can simply `pip uninstall yellowbrick` to cleanup the data.

A better option may be to use a single dataset directory across all virtual environments. To specify this directory, you must set the `$YELLOWBRICK_DATA` environment variable, usually by adding it to your bash profile so it is exported every time you open a terminal window. This will ensure that you have only downloaded the data once.

```
$ export YELLOWBRICK_DATA="~/.yellowbrick"
$ python -m yellowbrick.download -f
$ ls $YELLOWBRICK_DATA
```

To identify the location that the Yellowbrick datasets are stored for your installation of Python/Yellowbrick, you can use the `get_data_home` function:

`yellowbrick.datasets.path.`**`get_data_home`**(*path=None*)

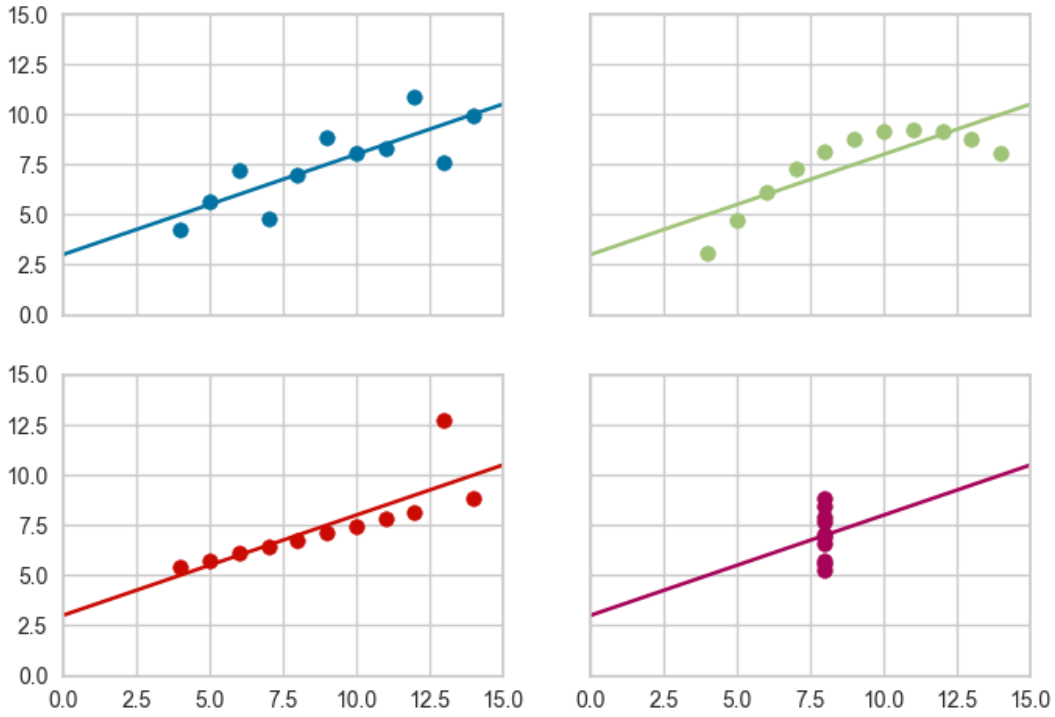
Return the path of the Yellowbrick data directory. This folder is used by dataset loaders to avoid downloading data several times.

By default, this folder is colocated with the code in the install directory so that data shipped with the package can be easily located. Alternatively it can be set by the `$YELLOWBRICK_DATA` environment variable, or programmatically by giving a folder path. Note that the `'~'` symbol is expanded to the user home directory, and environment variables are also expanded when resolving the path.

8.3.2 Anscombe's Quartet

Yellowbrick has learned Anscombe's lesson—which is why we believe that visual diagnostics are vital to machine learning.

```
import yellowbrick as yb
import matplotlib.pyplot as plt

g = yb.anscombe()
plt.show()
```



API Reference

Plots Anscombe's Quartet as an illustration of the importance of visualization.

yellowbrick.anscombe.**anscombe**()

Creates 2x2 grid plot of the 4 anscombe datasets for illustration.

8.3.3 Feature Analysis Visualizers

Feature analysis visualizers are designed to visualize instances in data space in order to detect features or targets that might impact downstream fitting. Because ML operates on high-dimensional data sets (usually at least 35), the visualizers focus on aggregation, optimization, and other techniques to give overviews of the data. It is our intent that the steering process will allow the data scientist to zoom and filter and explore the relationships between their instances and between dimensions.

At the moment we have the following feature analysis visualizers implemented:

- *Rank Features*: rank single and pairs of features to detect covariance
- *RadViz Visualizer*: plot data points along axes ordered around a circle to detect separability
- *Parallel Coordinates*: plot instances as lines along vertical axes to detect classes or clusters
- *PCA Projection*: project higher dimensions into a visual space using PCA
- *Manifold Visualization*: visualize high dimensional data using manifold learning
- *Direct Data Visualization*: (aka Jointplots) plot 2D correlation between features and target

Feature analysis visualizers implement the `Transformer` API from scikit-learn, meaning they can be used as intermediate transform steps in a `Pipeline` (particularly a `VisualPipeline`). They are instantiated in the same way, and then fit and transform are called on them, which draws the instances correctly. Finally `show` is called which finalizes and displays the image.

```
# Feature Analysis Imports
# NOTE that all these are available for import directly from the ``yellowbrick.features``␣
↪module
from yellowbrick.features.rankd import Rank1D, Rank2D
from yellowbrick.features.radviz import RadViz
from yellowbrick.features.pcoords import ParallelCoordinates
from yellowbrick.features.jointplot import JointPlotVisualizer
from yellowbrick.features.pca import PCADecomposition
from yellowbrick.features.manifold import Manifold
```

RadViz Visualizer

`RadViz` is a multivariate data visualization algorithm that plots each feature dimension uniformly around the circumference of a circle then plots points on the interior of the circle such that the point normalizes its values on the axes from the center to each arc. This mechanism allows as many dimensions as will easily fit on a circle, greatly expanding the dimensionality of the visualization.

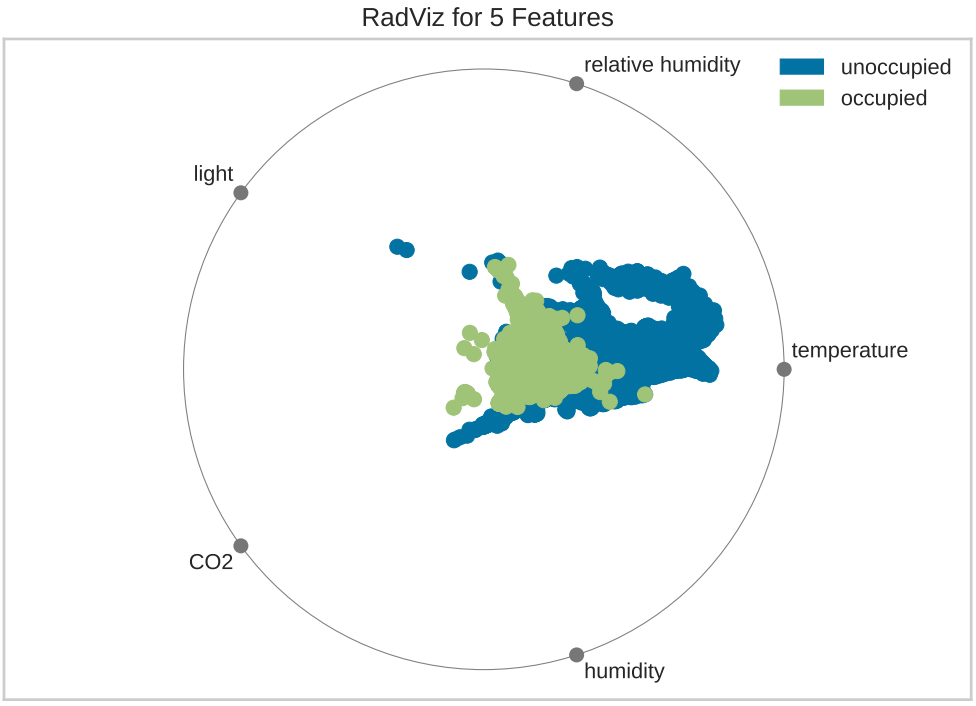
Data scientists use this method to detect separability between classes. E.g. is there an opportunity to learn from the feature set or is there just too much noise?

If your data contains rows with missing values (`numpy.nan`), those missing values will not be plotted. In other words, you may not get the entire picture of your data. `RadViz` will raise a `DataWarning` to inform you of the percent missing.

If you do receive this warning, you may want to look at imputation strategies. A good starting place is the scikit-learn Imputer.

Visualizer	*RadialVisualizer*
Quick Method	*radviz()*
Models	Classification, Regression
Workflow	Feature Engineering

```
from yellowbrick.datasets import load_occupancy
from yellowbrick.features import RadViz

# Load the classification dataset
X, y = load_occupancy()

# Specify the target classes
classes = ["unoccupied", "occupied"]

# Instantiate the visualizer
visualizer = RadViz(classes=classes)

visualizer.fit(X, y)           # Fit the data to the visualizer
visualizer.transform(X)        # Transform the data
visualizer.show()              # Finalize and render the figure
```



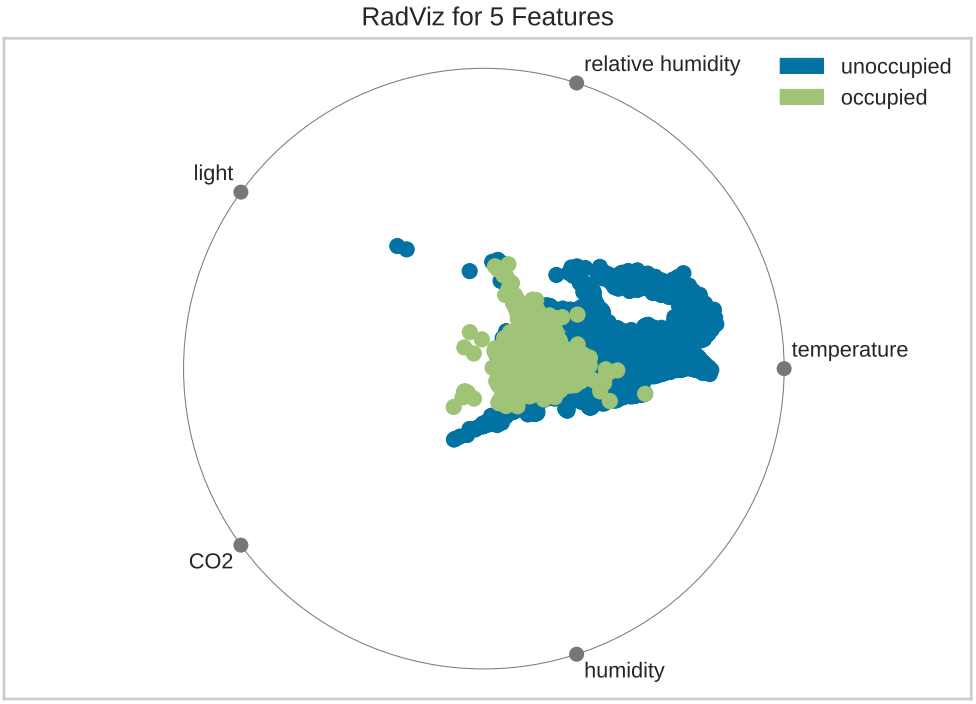
For regression, the RadViz visualizer should use a color sequence to display the target information, as opposed to

discrete colors.

Quick Method

The same functionality above can be achieved with the associated quick method `radviz`. This method will build the `RadViz` object with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from yellowbrick.features.radviz import radviz
from yellowbrick.datasets import load_occupancy

#Load the classification dataset
X, y = load_occupancy()

# Specify the target classes
classes = ["unoccupied", "occupied"]

# Instantiate the visualizer
radviz(X, y, classes=classes)
```



API Reference

Implements radviz for feature analysis.

yellowbrick.features.radviz.**RadViz**

alias of *RadialVisualizer*

class yellowbrick.features.radviz.**RadialVisualizer**(*ax=None*, *features=None*, *classes=None*, *colors=None*, *colormap=None*, *alpha=1.0*, ***kwargs*)

Bases: DataVisualizer

RadViz is a multivariate data visualization algorithm that plots each axis uniformely around the circumference of a circle then plots points on the interior of the circle such that the point normalizes its values on the axes from the center to each arc.

Parameters

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

features
[list, default: None] a list of feature names to use The names of the features specified by the columns of the input dataset. This length of this list must match the number of columns in X, otherwise an exception will be raised on `fit()`.

classes
[list, default: None] a list of class names for the legend The class labels for each class in y, ordered by sorted class index. These names act as a label encoder for the legend, identifying integer classes or renaming string labels. If omitted, the class labels will be taken from the unique values in y.

Note that the length of this list must match the number of unique values in y, otherwise an exception is raised. This parameter is only used in the discrete target type case and is ignored otherwise.

colors
[list or tuple, default: None] optional list or tuple of colors to colorize lines A single color to plot all instances as or a list of colors to color each instance according to its class. If not enough colors per class are specified then the colors are treated as a cycle.

colormap
[string or cmap, default: None] optional string or matplotlib cmap to colorize lines The colormap used to create the individual colors. If classes are specified the colormap is used to evenly space colors across each class.

alpha
[float, default: 1.0] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> visualizer = RadViz()
>>> visualizer.fit(X, y)
>>> visualizer.transform(X)
>>> visualizer.show()
```

Attributes

features_
[ndarray, shape (n_features,)] The names of the features discovered or used in the visualizer that can be used as an index to access or modify data in X. If a user passes feature names in, those features are used. Otherwise the columns of a DataFrame are used or just simply the indices of the data array.

classes_
[ndarray, shape (n_classes,)] The class labels that define the discrete values in the target. Only available if the target type is discrete. This is guaranteed to be strings even if the classes are a different type.

draw(*X*, *y*, ***kwargs*)

Called from the fit method, this method creates the radviz canvas and draws each instance as a class or target colored point, whose location is determined by the feature data set.

finalize(***kwargs*)

Sets the title and adds a legend. Removes the ticks from the graph to make a cleaner visualization.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

fit(*X*, *y=None*, ***kwargs*)

The fit method is the primary drawing input for the visualization since it has both the X and y data required for the viz and the transform method does not.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

kwargs
[dict] Pass generic arguments to the drawing method

Returns

self
[instance] Returns the instance of the transformer/visualizer

static normalize(*X*)

MinMax normalization to fit a matrix in the space [0,1] by column.

yellowbrick.features.radviz.**radviz**(*X*, *y=None*, *ax=None*, *features=None*, *classes=None*, *colors=None*, *colormap=None*, *alpha=1.0*, *show=True*, ***kwargs*)

Displays each feature as an axis around a circle surrounding a scatter plot whose points are each individual instance.

This helper function is a quick wrapper to utilize the RadialVisualizer (Transformer) for one-off analysis.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n, default:None] An array or series of target or class values

ax
[matplotlib Axes, default: None] The axes to plot the figure on.

features
[list of strings, default: None] The names of the features or columns

classes
[list of strings, default: None] The names of the classes in the target

colors
[list or tuple of colors, default: None] Specify the colors for each individual class

colormap
[string or matplotlib cmap, default: None] Sequential colormap for continuous target

alpha
[float, default: 1.0] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Returns

viz
[RadViz] Returns the fitted, finalized visualizer

Rank Features

`Rank1D` and `Rank2D` evaluate single features or pairs of features using a variety of metrics that score the features on the scale [-1, 1] or [0, 1] allowing them to be ranked. A similar concept to SPLOMs, the scores are visualized on a lower-left triangle heatmap so that patterns between pairs of features can be easily discerned for downstream analysis.

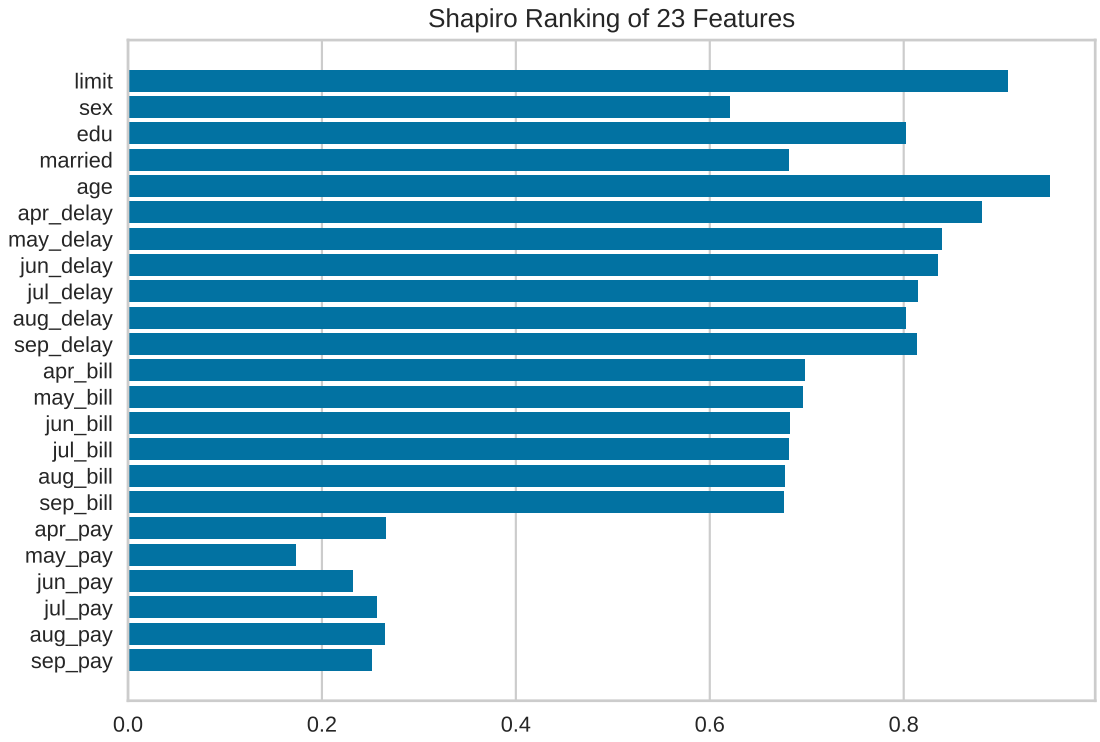
In this example, we'll use the credit default data set from the UCI Machine Learning repository to rank features. The code below creates our instance matrix and target vector.

Visualizers	*Rank1D*, *Rank2D*
Quick Methods	*rank1d()*, *rank2d()*
Models	General Linear Models
Workflow	Feature engineering and model selection

Rank 1D

A one-dimensional ranking of features utilizes a ranking algorithm that takes into account only a single feature at a time (e.g. histogram analysis). By default we utilize the Shapiro-Wilk algorithm to assess the normality of the distribution of instances with respect to the feature. A barplot is then drawn showing the relative ranks of each feature.

```
from yellowbrick.datasets import load_credit
from yellowbrick.features import Rank1D

# Load the credit dataset
X, y = load_credit()

# Instantiate the 1D visualizer with the Shapiro ranking algorithm
visualizer = Rank1D(algorithm='shapiro')

visualizer.fit(X, y)           # Fit the data to the visualizer
visualizer.transform(X)        # Transform the data
visualizer.show()              # Finalize and render the figure
```

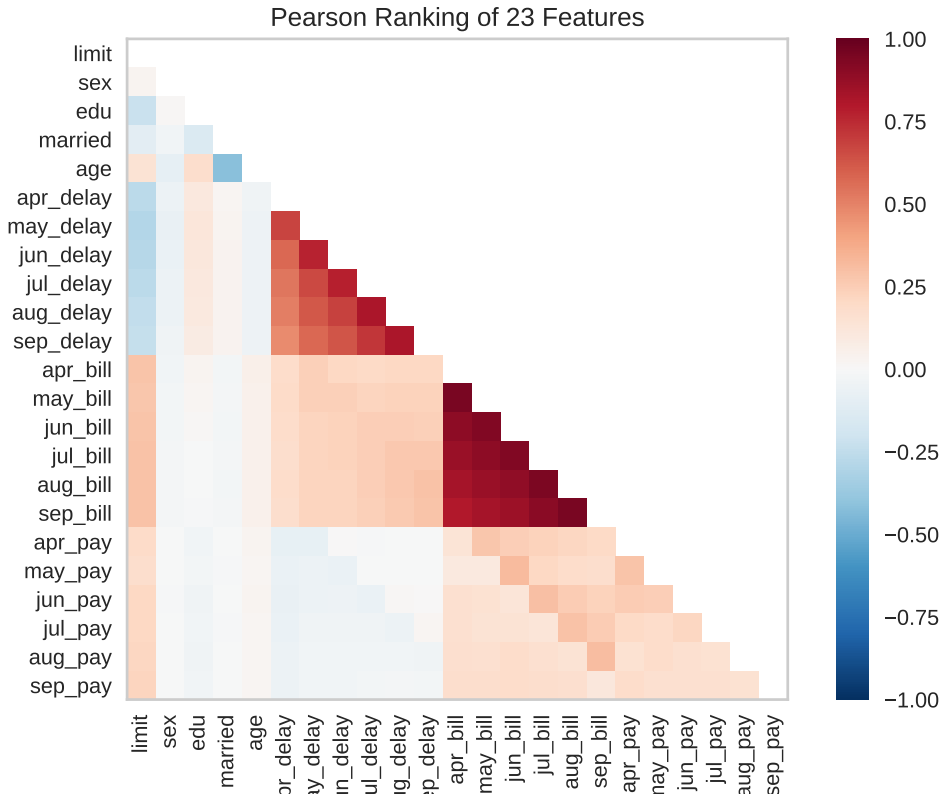


Rank 2D

A two-dimensional ranking of features utilizes a ranking algorithm that takes into account pairs of features at a time (e.g. joint plot analysis). The pairs of features are then ranked by score and visualized using the lower left triangle of a feature co-occurence matrix.

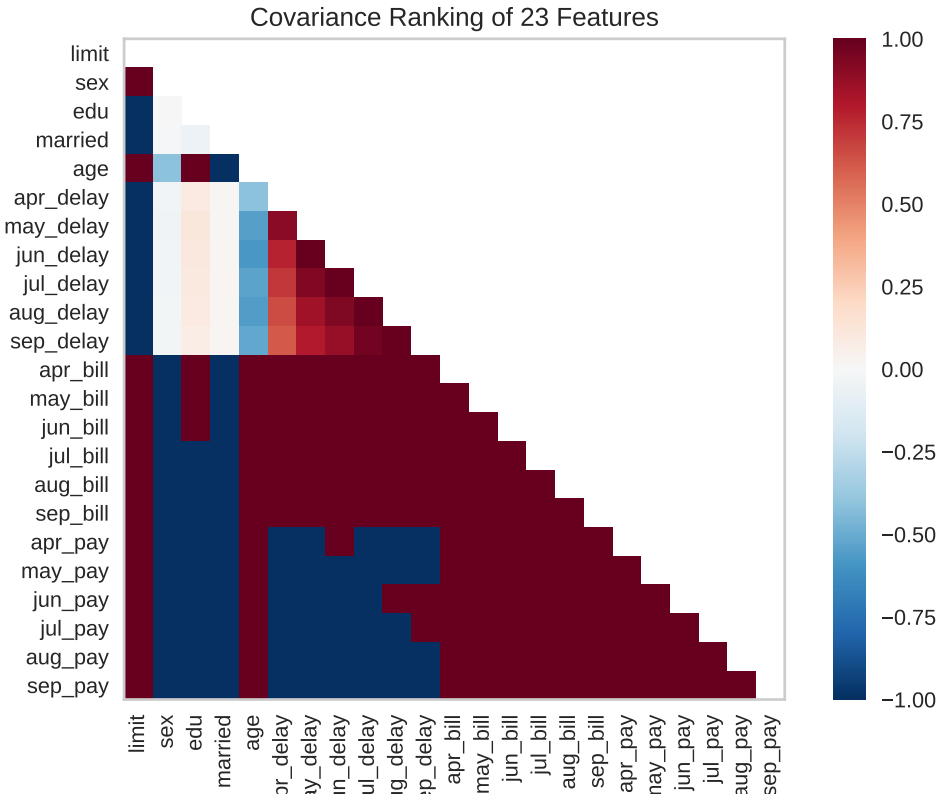
By default, the `Rank2D` visualizer utilizes the Pearson correlation score to detect colinear relationships.

```
from yellowbrick.datasets import load_credit
from yellowbrick.features import Rank2D

# Load the credit dataset
X, y = load_credit()

# Instantiate the visualizer with the Pearson ranking algorithm
visualizer = Rank2D(algorithm='pearson')

visualizer.fit(X, y)           # Fit the data to the visualizer
visualizer.transform(X)        # Transform the data
visualizer.show()              # Finalize and render the figure
```



Alternatively, we can utilize the covariance ranking algorithm, which attempts to compute the mean value of the product of deviations of variates from their respective means. Covariance loosely attempts to detect a colinear relationship between features. Compare the output from Pearson above to the covariance ranking below.

```
from yellowbrick.datasets import load_credit
from yellowbrick.features import Rank2D

# Load the credit dataset
X, y = load_credit()

# Instantiate the visualizer with the covariance ranking algorithm
visualizer = Rank2D(algorithm='covariance')

visualizer.fit(X, y)           # Fit the data to the visualizer
visualizer.transform(X)        # Transform the data
visualizer.show()              # Finalize and render the figure
```



Quick Methods

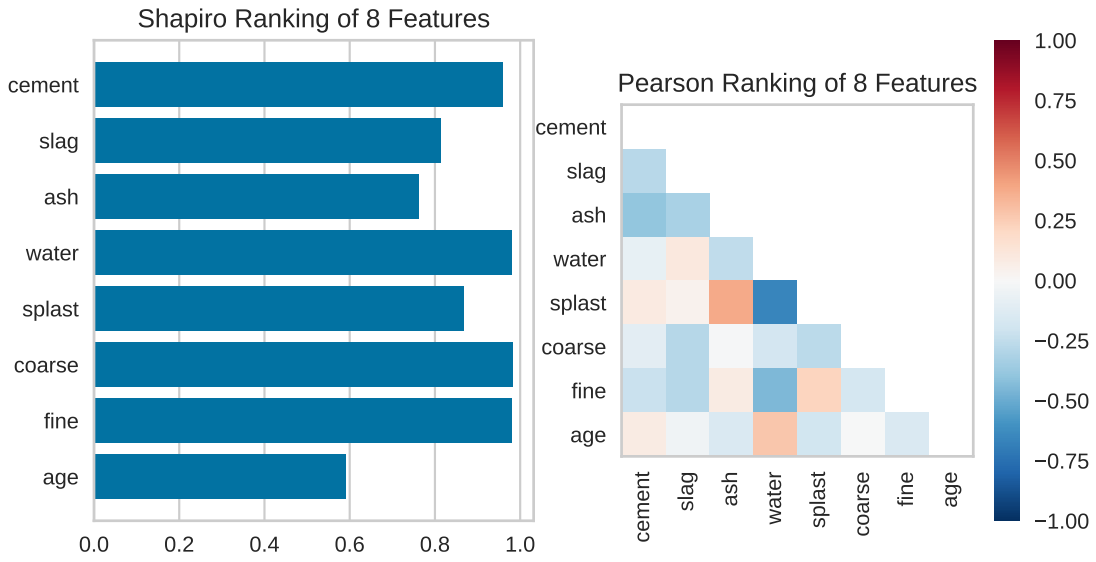
Similar functionality can be achieved using the one line quick methods, `rank1d` and `rank2d`. These functions instantiate and fit their respective visualizer on the data and immediately show it without having to use the class-based API.

```
from yellowbrick.datasets import load_concrete
from yellowbrick.features import rank1d, rank2d

# Load the concrete dataset
```

(continues on next page)

(continued from previous page)

```
X, _ = load_concrete()

_, axes = plt.subplots(ncols=2, figsize=(8,4))

rank1d(X, ax=axes[0], show=False)
rank2d(X, ax=axes[1], show=False)
plt.show()
```



API Reference

Implements 1D (histograms) and 2D (joint plot) feature rankings.

class yellowbrick.features.rankd.**Rank1D**(*ax=None*, *algorithm='shapiro'*, *features=None*, *orient='h'*, *show_feature_names=True*, *color=None*, ***kwargs*)

Bases: RankDBase

Rank1D computes a score for each feature in the data set with a specific metric or algorithm (e.g. Shapiro-Wilk) then returns the features ranked as a bar plot.

Parameters

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

algorithm
[one of {'shapiro', }, default: 'shapiro'] The ranking algorithm to use, default is 'Shapiro-Wilk.

features
[list] A list of feature names to use. If a DataFrame is passed to fit and features is None, feature names are selected as the columns of the DataFrame.

orient
['h' or 'v', default='h'] Specifies a horizontal or vertical bar chart.

show_feature_names
[boolean, default: True] If True, the feature names are used to label the x and y ticks in the plot.

color: string
Specify color for barchart

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> visualizer = Rank1D()
>>> visualizer.fit(X, y)
>>> visualizer.transform(X)
>>> visualizer.show()
```

Attributes

ranks_
[ndarray] An array of rank scores with shape (n,), where n is the number of features. It is computed during *fit*.

draw(***kwargs*)

Draws the bar plot of the ranking array of features.

ranking_methods = {'shapiro': <function Rank1D.<lambda>>}

class yellowbrick.features.rankd.**Rank2D**(*ax=None*, *algorithm='pearson'*, *features=None*, *colormap='RdBu_r'*, *show_feature_names=True*, ***kwargs*)

Bases: RankDBase

Rank2D performs pairwise comparisons of each feature in the data set with a specific metric or algorithm (e.g. Pearson correlation) then returns them ranked as a lower left triangle diagram.

Parameters

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

algorithm
[str, default: 'pearson'] The ranking algorithm to use, one of: 'pearson', 'covariance', 'spearman', or 'kendalltau'.

features
[list] A list of feature names to use. If a DataFrame is passed to fit and features is None, feature names are selected as the columns of the DataFrame.

colormap
[string or cmap, default: 'RdBu_r'] optional string or matplotlib cmap to colorize lines Use either color to colorize the lines on a per class basis or colormap to color them on a continuous scale.

show_feature_names
[boolean, default: True] If True, the feature names are used to label the axis ticks in the plot.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

These parameters can be influenced later on in the visualization process, but can and should be set as early as possible.

Examples

```
>>> visualizer = Rank2D()
>>> visualizer.fit(X, y)
>>> visualizer.transform(X)
>>> visualizer.show()
```

Attributes

ranks_
[ndarray] An array of rank scores with shape (n,n), where n is the number of features. It is computed during *fit*.

draw(***kwargs*)

Draws the heatmap of the ranking matrix of variables.

ranking_methods = {'covariance': <function Rank2D.<lambda>>, 'kendalltau': <function Rank2D.<lambda>>, 'pearson': <function Rank2D.<lambda>>, 'spearman': <function Rank2D.<lambda>>}

yellowbrick.features.rankd.**rank1d**(*X*, *y=None*, *ax=None*, *algorithm='shapiro'*, *features=None*, *orient='h'*, *show_feature_names=True*, *color=None*, *show=True*, ***kwargs*)

Scores each feature with the algorithm and ranks them in a bar plot.

This helper function is a quick wrapper to utilize the Rank1D Visualizer (Transformer) for one-off analysis.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

ax
[matplotlib axes] the axis to plot the figure on.

algorithm
[one of {'shapiro', }, default: 'shapiro'] The ranking algorithm to use, default is 'Shapiro-Wilk.

features
[list] A list of feature names to use. If a DataFrame is passed to fit and features is None, feature names are selected as the columns of the DataFrame.

orient
['h' or 'v'] Specifies a horizontal or vertical bar chart.

show_feature_names
[boolean, default: True] If True, the feature names are used to label the axis ticks in the plot.

color: string
Specify color for barchart

show: bool, default: True

If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

viz
[Rank1D] Returns the fitted, finalized visualizer.

yellowbrick.features.rankd.**rank2d**(*X*, *y=None*, *ax=None*, *algorithm='pearson'*, *features=None*, *colormap='RdBu_r'*, *show_feature_names=True*, *show=True*, ***kwargs*)

Rank2D quick method

Rank2D performs pairwise comparisons of each feature in the data set with a specific metric or algorithm (e.g. Pearson correlation) then returns them ranked as a lower left triangle diagram.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features to perform the pairwise compairsons on.

y
[ndarray or Series of length n, default: None] An array or series of target or class values, optional (not used).

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

algorithm
[str, default: 'pearson'] The ranking algorithm to use, one of: 'pearson', 'covariance', 'spearman', or 'kendalltau'.

features
[list] A list of feature names to use. If a DataFrame is passed to fit and features is None, feature names are selected as the columns of the DataFrame.

colormap
[string or cmap, default: 'RdBu_r'] optional string or matplotlib cmap to colorize lines Use either color to colorize the lines on a per class basis or colormap to color them on a continuous scale.

show_feature_names
[boolean, default: True] If True, the feature names are used to label the axis ticks in the plot.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

viz
[Rank2D] Returns the fitted, finalized visualizer that created the Rank2D heatmap.

Parallel Coordinates

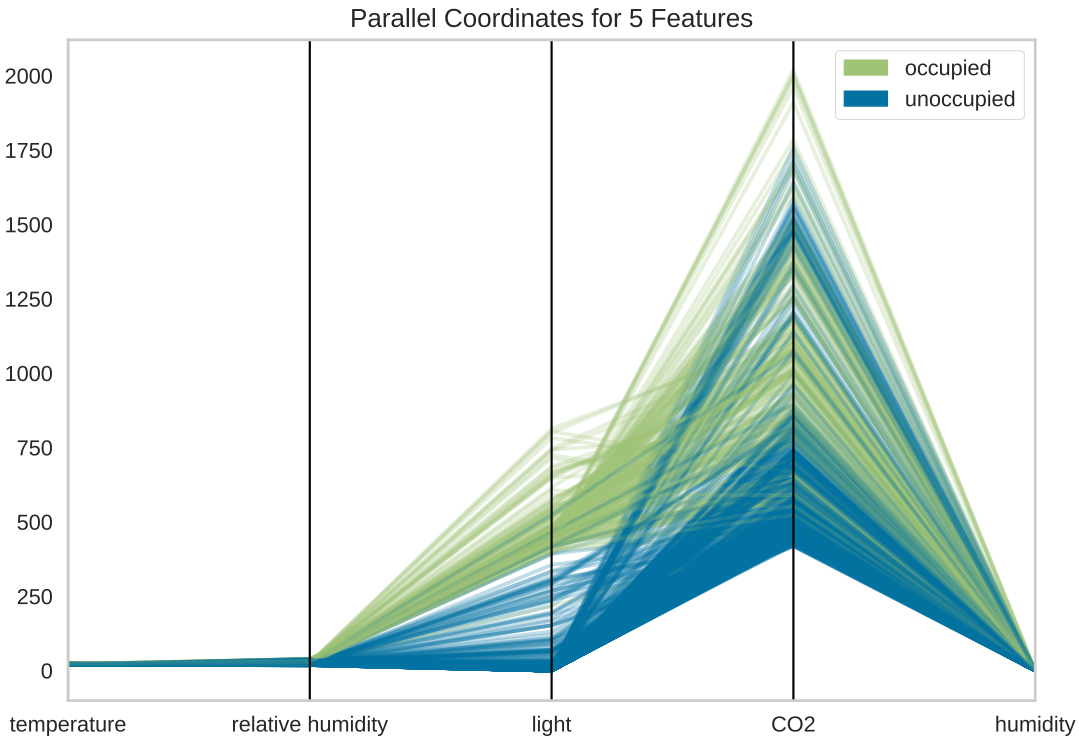
Parallel coordinates is multi-dimensional feature visualization technique where the vertical axis is duplicated horizontally for each feature. Instances are displayed as a single line segment drawn from each vertical axes to the location representing their value for that feature. This allows many dimensions to be visualized at once; in fact given infinite horizontal space (e.g. a scrolling window), technically an infinite number of dimensions can be displayed!

Data scientists use this method to detect clusters of instances that have similar classes, and to note features that have high variance or different distributions. We can see this in action after first loading our occupancy classification dataset.

Visualizer	*ParallelCoordinates*
Quick Method	*parallel_coordinates()*
Models	Classification
Workflow	Feature analysis

```
from yellowbrick.features import ParallelCoordinates
from yellowbrick.datasets import load_occupancy

# Load the classification data set
X, y = load_occupancy()

# Specify the features of interest and the classes of the target
features = [
    "temperature", "relative humidity", "light", "CO2", "humidity"
]
classes = ["unoccupied", "occupied"]

# Instantiate the visualizer
visualizer = ParallelCoordinates(
    classes=classes, features=features, sample=0.05, shuffle=True
)

# Fit and transform the data to the visualizer
visualizer.fit_transform(X, y)

# Finalize the title and axes then display the visualization
visualizer.show()
```

By inspecting the visualization closely, we can see that the combination of transparency and overlap gives us the sense of groups of similar instances, sometimes referred to as "braids". If there are distinct braids of different classes, it suggests that there is enough separability that a classification algorithm might be able to discern between each class.



Unfortunately, as we inspect this class, we can see that the domain of each feature may make the visualization hard to interpret. In the above visualization, the domain of the `light` feature is from in `[0, 1600]`, far larger than the range of temperature in `[50, 96]`. To solve this problem, each feature should be scaled or normalized so they are approximately in the same domain.

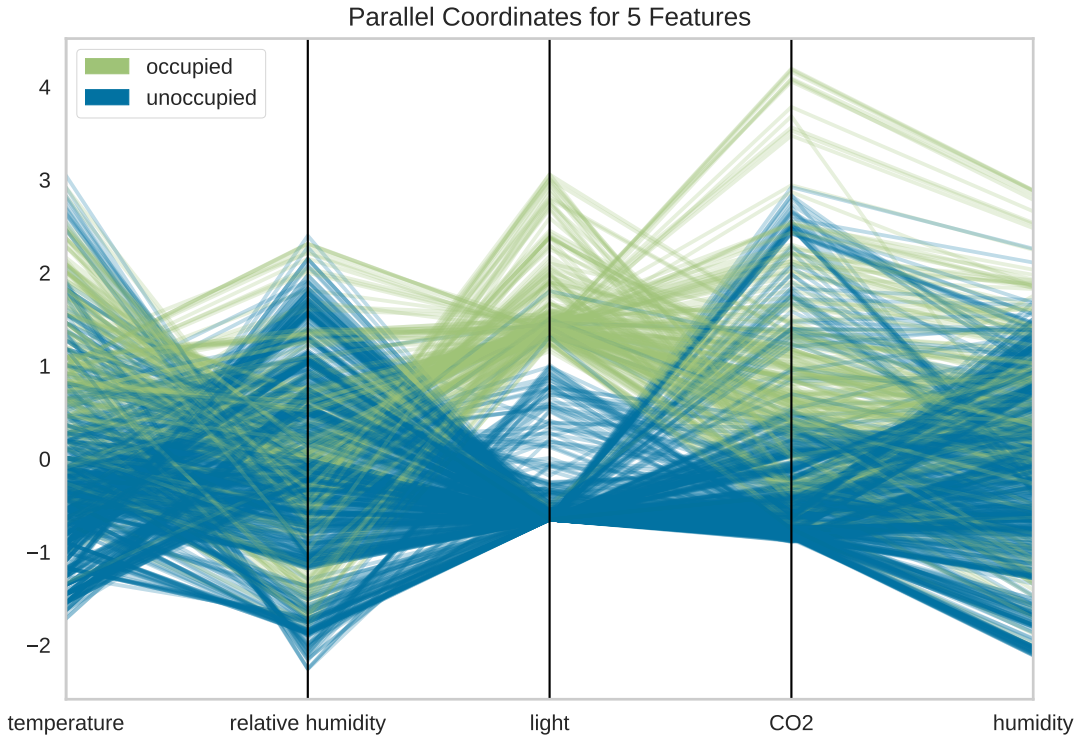
Normalization techniques can be directly applied to the visualizer without pre-transforming the data (though you could also do this) by using the `normalize` parameter. Several transformers are available; try using `minmax`, `maxabs`, `standard`, `l1`, or `l2` normalization to change perspectives in the parallel coordinates as follows:

```
from yellowbrick.features import ParallelCoordinates
from yellowbrick.datasets import load_occupancy

# Load the classification data set
X, y = load_occupancy()

# Specify the features of interest and the classes of the target
features = [
    "temperature", "relative humidity", "light", "CO2", "humidity"
]
classes = ["unoccupied", "occupied"]

# Instantiate the visualizer
visualizer = ParallelCoordinates(
    classes=classes, features=features,
    normalize='standard', sample=0.05, shuffle=True,
```

(continues on next page)

(continued from previous page)

```
)

# Fit the visualizer and display it
visualizer.fit_transform(X, y)
visualizer.show()
```



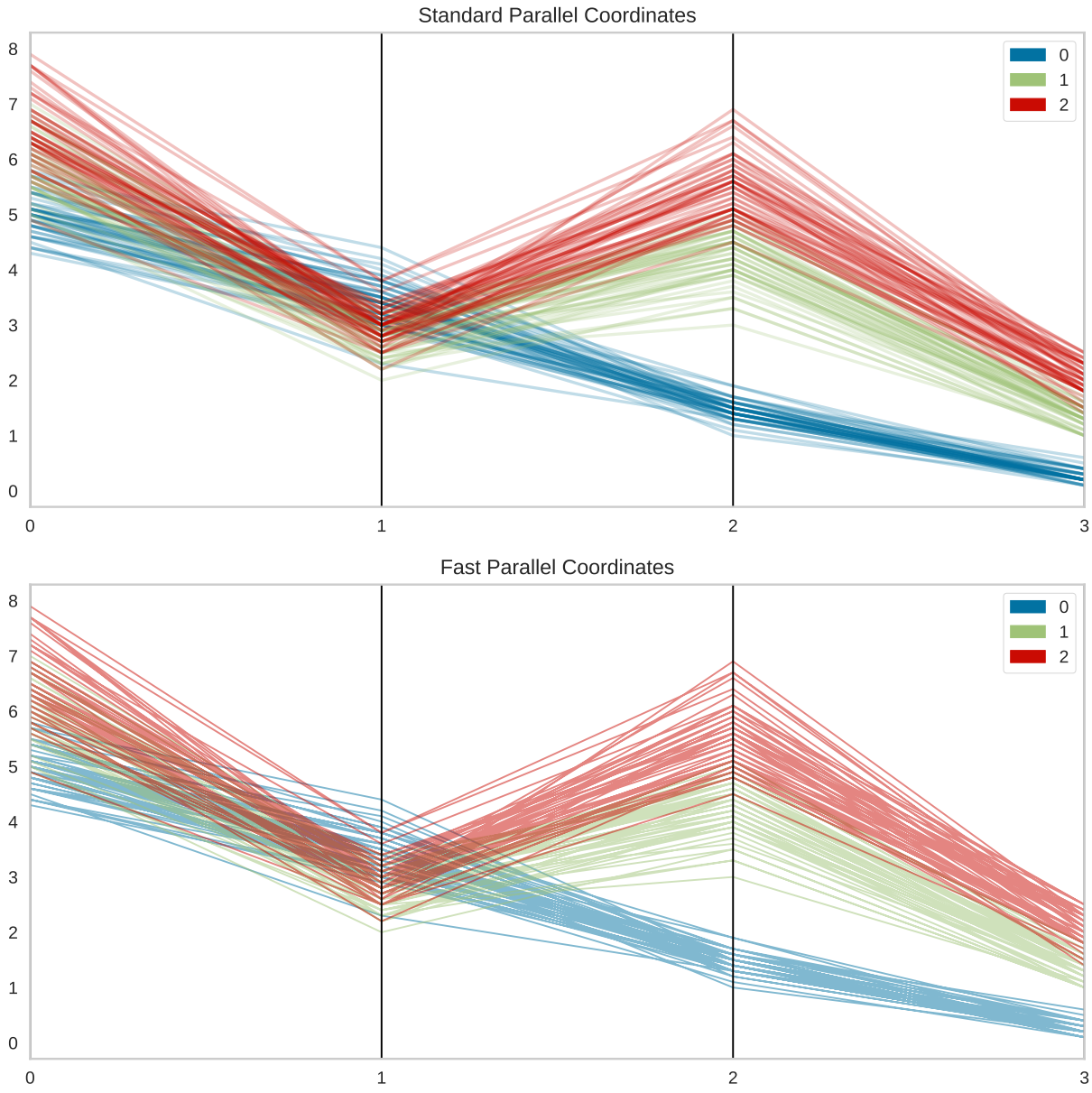
Now we can see that each feature is in the range `[-3, 3]` where the mean of the feature is set to zero and each feature has a unit variance applied between `[-1, 1]` (because we're using the `StandardScaler` via the `standard` normalize parameter). This version of parallel coordinates gives us a much better sense of the distribution of the features and if any features are highly variable with respect to any one class.

Faster Parallel Coordinates

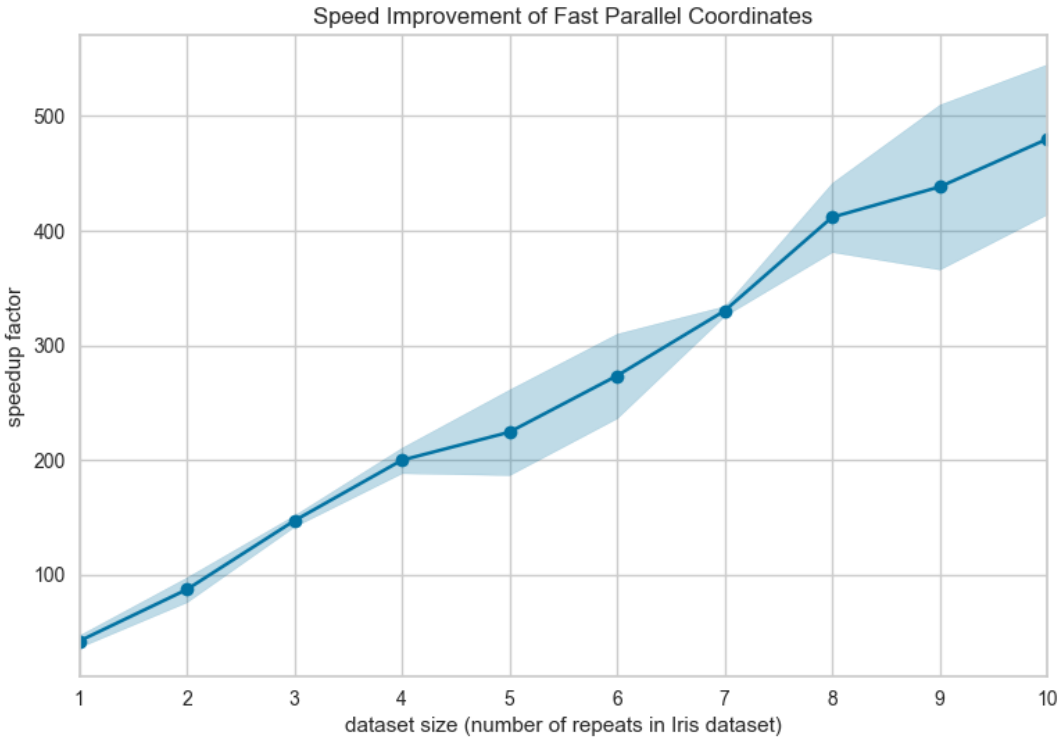
Parallel coordinates can take a long time to draw since each instance is represented by a line for each feature. Worse, this time is not well spent since a lot of overlap in the visualization makes the parallel coordinates less understandable. We propose two solutions to this:

1. Use `sample=0.2` and `shuffle=True` parameters to shuffle and sample the dataset being drawn on the figure. The sample parameter will perform a uniform random sample of the data, selecting the percent specified.
2. Use the `fast=True` parameter to enable "fast drawing mode".

The "fast" drawing mode vastly improves the performance of the parallel coordinates drawing algorithm by drawing each line segment by class rather than each instance individually. However, this improved performance comes at a cost, as the visualization produced is subtly different; compare the visualizations in fast and standard drawing modes below:


Standard Parallel Coordinates
0
1
2
0
1
2
3
4
5
6
7
8
Fast Parallel Coordinates
0
1
2
0
1
2
3
4
5
6
7
8

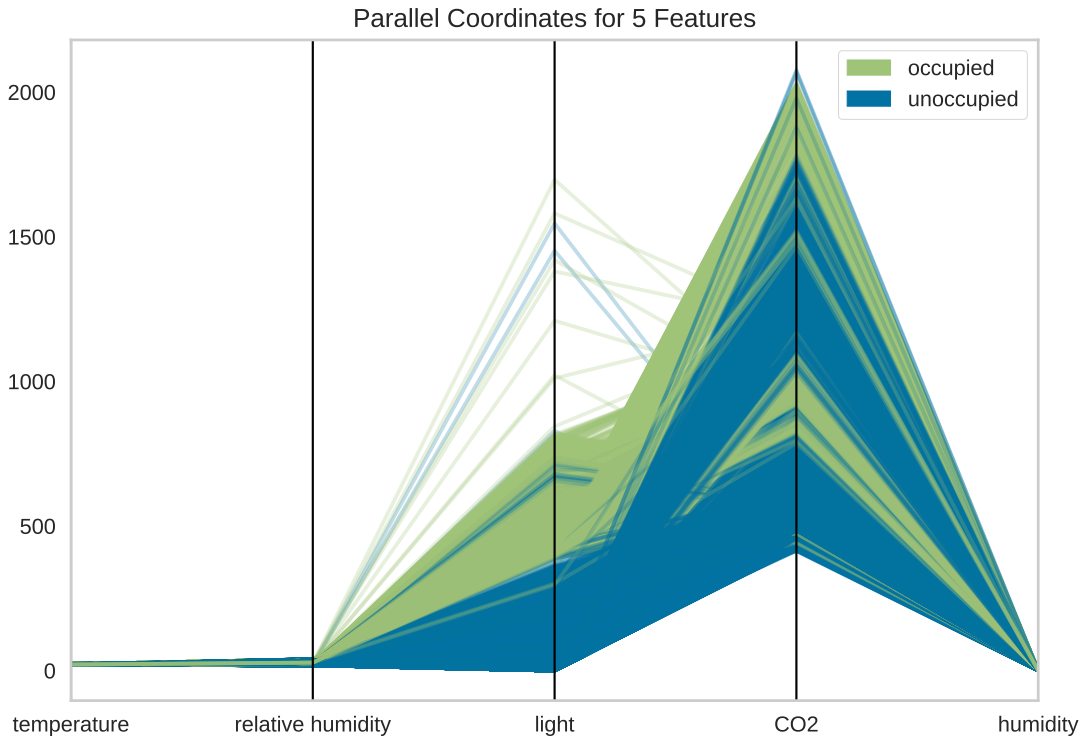
As you can see the "fast" drawing algorithm does not have the same build up of color density where instances of the same class intersect. Because there is only one line per class, there is only a darkening effect between classes. This can lead to a different interpretation of the plot, though it still may be effective for analytical purposes, particularly when you're plotting a lot of data. Needless to say, the performance benefits are dramatic:



Quick Method

The same functionality above can be achieved with the associated quick method `parallel_coordinates`. This method will build the `ParallelCoordinates` object with the associated arguments, fit it, then (optionally) immediately show it.

```
from yellowbrick.features.pcoords import parallel_coordinates
from yellowbrick.datasets import load_occupancy

# Load the classification data set
X, y = load_occupancy()

# Specify the features of interest and the classes of the target
features = [
    "temperature", "relative humidity", "light", "CO2", "humidity"
]
classes = ["unoccupied", "occupied"]

# Instantiate the visualizer
visualizer = parallel_coordinates(X, y, classes=classes, features=features)
```



API Reference

Implementation of parallel coordinates for multi-dimensional feature analysis.

class yellowbrick.features.pcoords.**ParallelCoordinates**(*ax=None*, *features=None*, *classes=None*, *normalize=None*, *sample=1.0*, *random_state=None*, *shuffle=False*, *colors=None*, *colormap=None*, *alpha=None*, *fast=False*, *vlines=True*, *vlines_kwds=None*, ***kwargs*)

Bases: DataVisualizer

Parallel coordinates displays each feature as a vertical axis spaced evenly along the horizontal, and each instance as a line drawn between each individual axis. This allows you to detect braids of similar instances and separability that suggests a good classification problem.

Parameters

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

features
[list, default: None] a list of feature names to use If a DataFrame is passed to fit and features is None, feature names are selected as the columns of the DataFrame.

classes
[list, default: None] a list of class names for the legend The class labels for each class in y,

ordered by sorted class index. These names act as a label encoder for the legend, identifying integer classes or renaming string labels. If omitted, the class labels will be taken from the unique values in y.

Note that the length of this list must match the number of unique values in y, otherwise an exception is raised.

normalize
[string or None, default: None] specifies which normalization method to use, if any Current supported options are 'minmax', 'maxabs', 'standard', 'l1', and 'l2'.

sample
[float or int, default: 1.0] specifies how many examples to display from the data If int, specifies the maximum number of samples to display. If float, specifies a fraction between 0 and 1 to display.

random_state
[int, RandomState instance or None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random; only used if shuffle is True and sample < 1.0

shuffle
[boolean, default: True] specifies whether sample is drawn randomly

colors
[list or tuple, default: None] A single color to plot all instances as or a list of colors to color each instance according to its class. If not enough colors per class are specified then the colors are treated as a cycle.

colormap
[string or cmap, default: None] The colormap used to create the individual colors. If classes are specified the colormap is used to evenly space colors across each class.

alpha
[float, default: None] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered lines more visible. If None, the alpha is set to 0.5 in "fast" mode and 0.25 otherwise.

fast
[bool, default: False] Fast mode improves the performance of the drawing time of parallel coordinates but produces an image that does not show the overlap of instances in the same class. Fast mode should be used when drawing all instances is too burdensome and sampling is not an option.

vlines
[boolean, default: True] flag to determine vertical line display

vlines_kwds
[dict, default: None] options to style or display the vertical lines, default: None

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> visualizer = ParallelCoordinates()
>>> visualizer.fit(X, y)
>>> visualizer.transform(X)
>>> visualizer.show()
```

Attributes

n_samples_
[int] number of samples included in the visualization object

features_
[ndarray, shape (n_features,)] The names of the features discovered or used in the visualizer that can be used as an index to access or modify data in X. If a user passes feature names in, those features are used. Otherwise the columns of a DataFrame are used or just simply the indices of the data array.

classes_
[ndarray, shape (n_classes,)] The class labels that define the discrete values in the target. Only available if the target type is discrete. This is guaranteed to be strings even if the classes are a different type.

```
NORMALIZERS = {'l1':  Normalizer(norm='l1'), 'l2':  Normalizer(), 'maxabs':
MaxAbsScaler(), 'minmax':  MinMaxScaler(), 'standard':  StandardScaler()}
```

draw(*X*, *y*, ***kwargs*)

Called from the fit method, this method creates the parallel coordinates canvas and draws each instance and vertical lines on it.

Parameters

X
[ndarray of shape n x m] A matrix of n instances with m features

y
[ndarray of length n] An array or series of target or class values

kwargs
[dict] Pass generic arguments to the drawing method

draw_classes(*X*, *y*, ***kwargs*)

Draw the instances colored by the target y such that each line is a single class. This is the "fast" mode of drawing, since the number of lines drawn equals the number of classes, rather than the number of instances. However, this drawing method sacrifices inter-class density of points using the alpha parameter.

Parameters

X
[ndarray of shape n x m] A matrix of n instances with m features

y
[ndarray of length n] An array or series of target or class values

draw_instances(*X*, *y*, ***kwargs*)

Draw the instances colored by the target y such that each line is a single instance. This is the "slow" mode of drawing, since each instance has to be drawn individually. However, in so doing, the density of instances in braids is more apparent since lines have an independent alpha that is compounded in the figure.

This is the default method of drawing.

Parameters

X
[ndarray of shape n x m] A matrix of n instances with m features

y
[ndarray of length n] An array or series of target or class values

Notes

This method can be used to draw additional instances onto the parallel coordinates before the figure is finalized.

finalize(***kwargs*)

Performs the final rendering for the multi-axis visualization, including setting and rendering the vertical axes each instance is plotted on. Adds a title, a legend, and manages the grid.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

fit(*X*, *y=None*, ***kwargs*)

The fit method is the primary drawing input for the visualization since it has both the X and y data required for the viz and the transform method does not.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

kwargs
[dict] Pass generic arguments to the drawing method

Returns

self
[instance] Returns the instance of the transformer/visualizer

yellowbrick.features.pcoords.**parallel_coordinates**(*X*, *y*, *ax=None*, *features=None*, *classes=None*, *normalize=None*, *sample=1.0*, *random_state=None*, *shuffle=False*, *colors=None*, *colormap=None*, *alpha=None*, *fast=False*, *vlines=True*, *vlines_kwds=None*, *show=True*, ***kwargs*)

Displays each feature as a vertical axis and each instance as a line.

This helper function is a quick wrapper to utilize the ParallelCoordinates Visualizer (Transformer) for one-off analysis.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

features
[list, default: None] a list of feature names to use If a DataFrame is passed to fit and features is None, feature names are selected as the columns of the DataFrame.

classes
[list, default: None] a list of class names for the legend If classes is None and a y value is passed to fit then the classes are selected from the target vector.

normalize
[string or None, default: None] specifies which normalization method to use, if any Current supported options are 'minmax', 'maxabs', 'standard', 'l1', and 'l2'.

sample
[float or int, default: 1.0] specifies how many examples to display from the data If int, specifies the maximum number of samples to display. If float, specifies a fraction between 0 and 1 to display.

random_state
[int, RandomState instance or None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random; only used if shuffle is True and sample < 1.0

shuffle
[boolean, default: True] specifies whether sample is drawn randomly

colors
[list or tuple, default: None] optional list or tuple of colors to colorize lines Use either color to colorize the lines on a per class basis or colormap to color them on a continuous scale.

colormap
[string or cmap, default: None] optional string or matplotlib cmap to colorize lines Use either color to colorize the lines on a per class basis or colormap to color them on a continuous scale.

alpha
[float, default: None] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered lines more visible. If None, the alpha is set to 0.5 in "fast" mode and 0.25 otherwise.

fast
[bool, default: False] Fast mode improves the performance of the drawing time of parallel coordinates but produces an image that does not show the overlap of instances in the same class. Fast mode should be used when drawing all instances is too burdensome and sampling is not an option.

vlines
[boolean, default: True] flag to determine vertical line display

vlines_kwds
[dict, default: None] options to style or display the vertical lines, default: None

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

viz
[ParallelCoordinates] Returns the fitted, finalized visualizer

PCA Projection

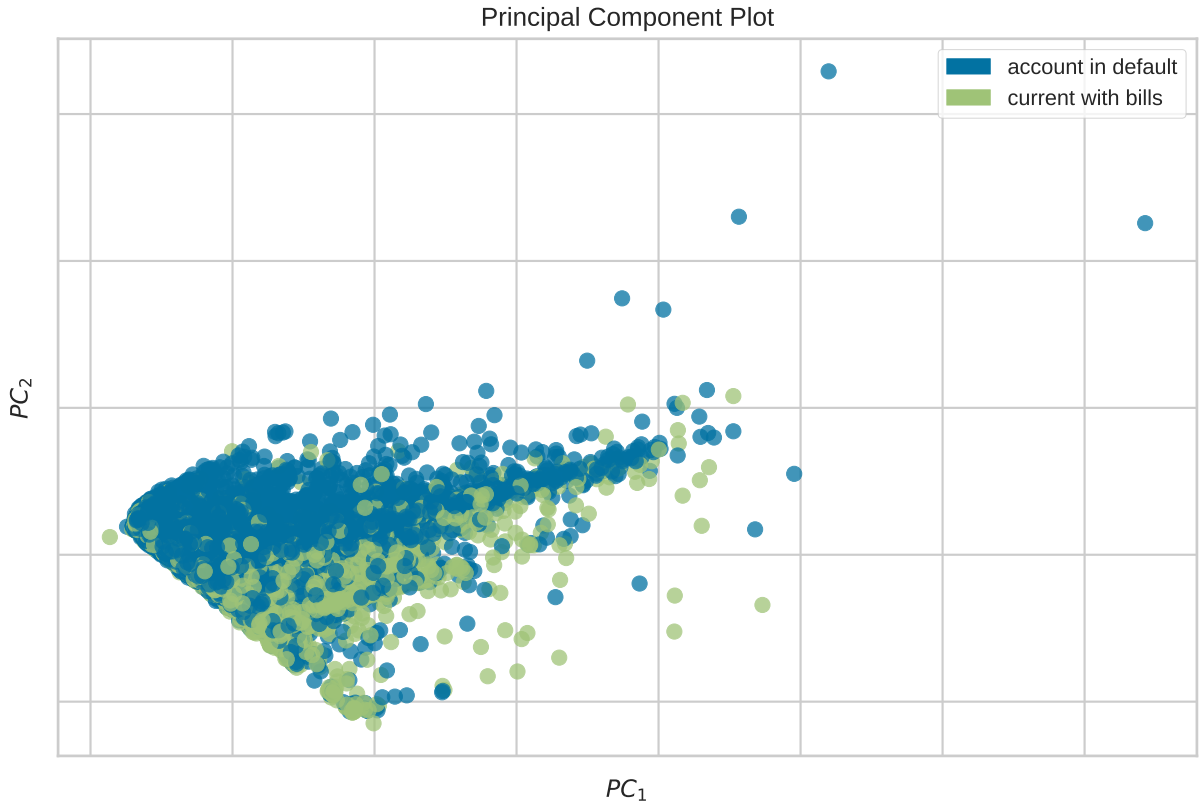
The PCA Decomposition visualizer utilizes principal component analysis to decompose high dimensional data into two or three dimensions so that each instance can be plotted in a scatter plot. The use of PCA means that the projected dataset can be analyzed along axes of principal variation and can be interpreted to determine if spherical distance metrics can be utilized.

Visualizer	*PCA*
Quick Method	*pca_decomposition()*
Models	Classification/Regression
Workflow	Feature Engineering/Selection

```
from yellowbrick.datasets import load_credit
from yellowbrick.features import PCA

# Specify the features of interest and the target
X, y = load_credit()
classes = ['account in default', 'current with bills']

visualizer = PCA(scale=True, classes=classes)
visualizer.fit_transform(X, y)
visualizer.show()
```

The PCA projection can also be plotted in three dimensions to attempt to visualize more principal components and get a better sense of the distribution in high dimensions.

```
from yellowbrick.datasets import load_credit
from yellowbrick.features import PCA

X, y = load_credit()
classes = ['account in default', 'current with bills']

visualizer = PCA(
    scale=True, projection=3, classes=classes
)
visualizer.fit_transform(X, y)
visualizer.show()
```



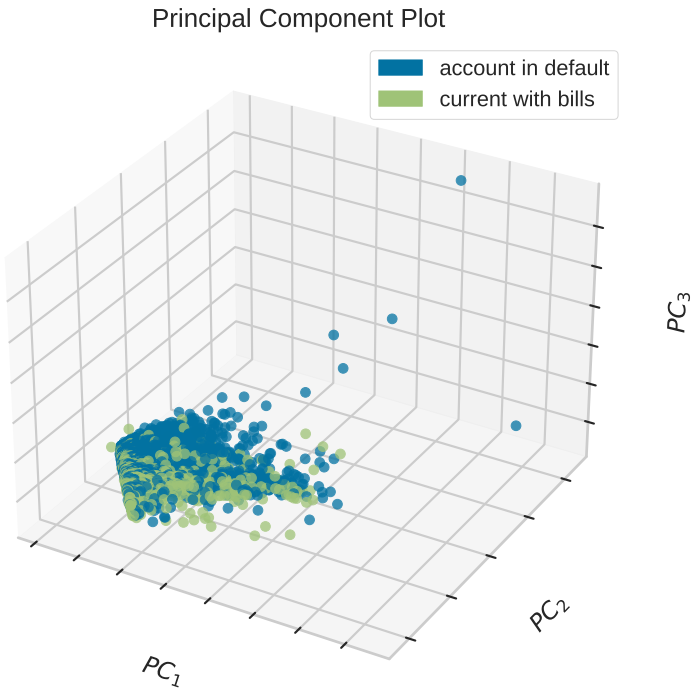
Biplot

The PCA projection can be enhanced to a biplot whose points are the projected instances and whose vectors represent the structure of the data in high dimensional space. By using `proj_features=True`, vectors for each feature in the dataset are drawn on the scatter plot in the direction of the maximum variance for that feature. These structures can be used to analyze the importance of a feature to the decomposition or to find features of related variance for further analysis.

```
from yellowbrick.datasets import load_concrete
from yellowbrick.features import PCA

# Load the concrete dataset
X, y = load_concrete()

visualizer = PCA(scale=True, proj_features=True)
visualizer.fit_transform(X, y)
visualizer.show()
```

```
from yellowbrick.datasets import load_concrete
from yellowbrick.features import PCA

X, y = load_concrete()

visualizer = PCA(scale=True, proj_features=True, projection=3)
```

(continues on next page)



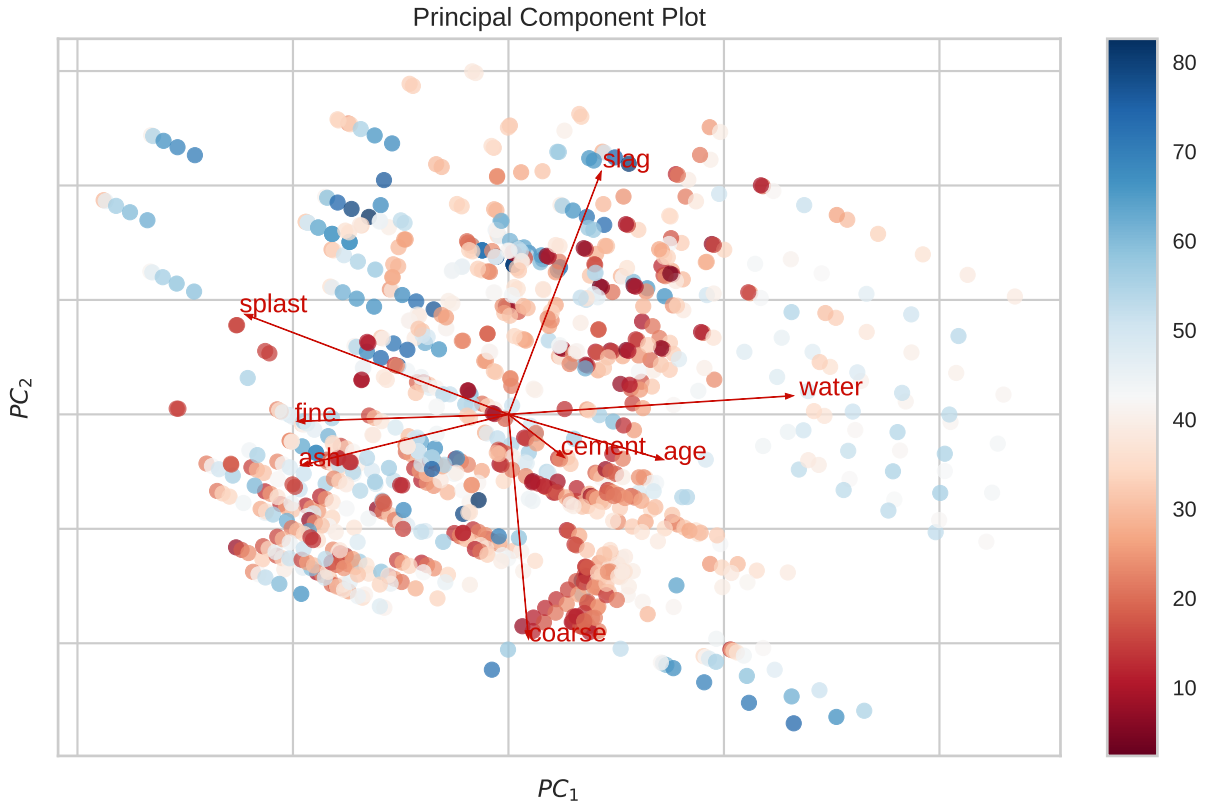
(continued from previous page)

```
visualizer.fit_transform(X, y)
visualizer.show()
```

Quick Method

The same functionality above can be achieved with the associated quick method `pca_decomposition`. This method will build the `PCA` object with the associated arguments, fit it, then (optionally) immediately show it.

```
from yellowbrick.datasets import load_credit
from yellowbrick.features import pca_decomposition

# Specify the features of interest and the target
X, y = load_credit()
classes = ['account in default', 'current with bills']

# Create, fit, and show the visualizer
pca_decomposition(
    X, y, scale=True, classes=classes
)
```



API Reference

Decomposition based feature visualization with PCA.

class yellowbrick.features.pca.**PCA**(*ax=None*, *features=None*, *classes=None*, *scale=True*, *projection=2*, *proj_features=False*, *colors=None*, *colormap=None*, *alpha=0.75*, *random_state=None*, *colorbar=True*, *heatmap=False*, ***kwargs*)

Bases: `ProjectionVisualizer`

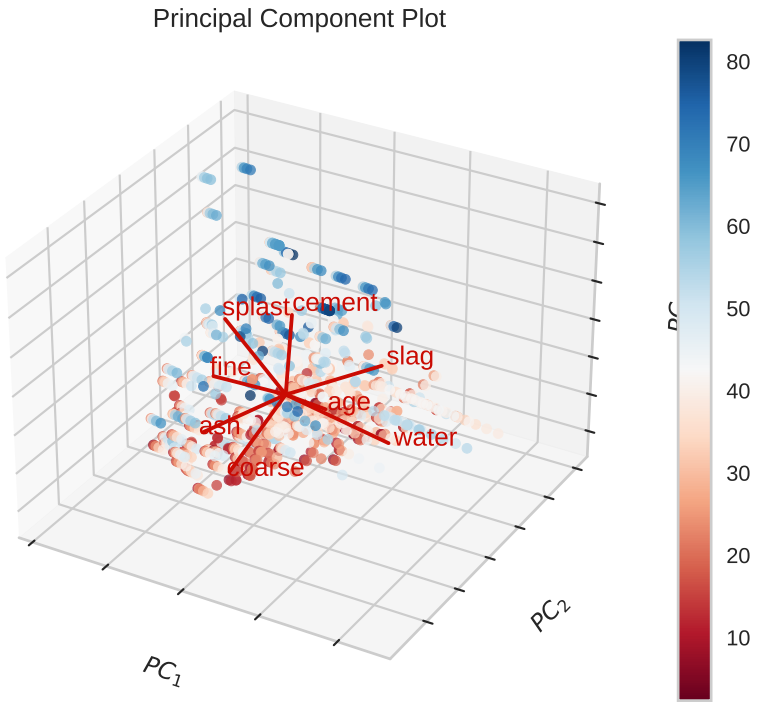
Produce a two or three dimensional principal component plot of a data array projected onto its largest sequential principal components. It is common practice to scale the data array `X` before applying a PC decomposition. Variable scaling can be controlled using the `scale` argument.

Parameters

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in, the current axes will be used (or generated if required).

features
[list, default: None] The names of the features specified by the columns of the input dataset. This length of this list must match the number of columns in X, otherwise an exception will be raised on `fit()`.

classes
[list, default: None] The class labels for each class in y, ordered by sorted class index. These names act as a label encoder for the legend, identifying integer classes or renaming string labels. If omitted, the class labels will be taken from the unique values in y.



Note that the length of this list must match the number of unique values in y, otherwise an exception is raised. This parameter is only used in the discrete target type case and is ignored otherwise.

scale
[bool, default: True] Boolean that indicates if user wants to scale data.

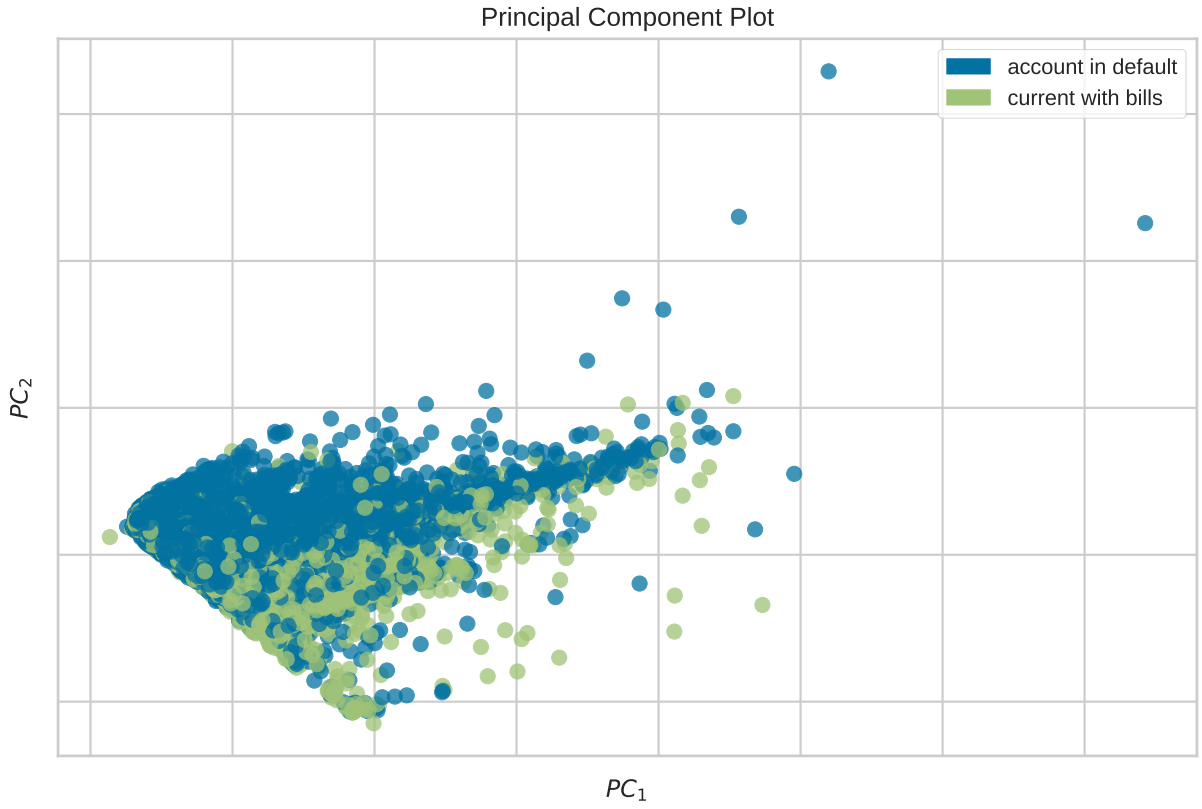
projection
[int or string, default: 2] The number of axes to project into, either 2d or 3d. To plot 3d plots with matplotlib, please ensure a 3d axes is passed to the visualizer, otherwise one will be created using the current figure.

proj_features
[bool, default: False] Boolean that indicates if the user wants to project the features in the projected space. If True the plot will be similar to a biplot.

colors
[list or tuple, default: None] A single color to plot all instances as or a list of colors to color each instance according to its class in the discrete case or as an ordered colormap in the sequential case. If not enough colors per class are specified then the colors are treated as a cycle.

colormap
[string or cmap, default: None] The colormap used to create the individual colors. In the discrete case it is used to compute the number of colors needed for each class and in the continuous case it is used to create a sequential color map based on the range of the target.

alpha



[float, default: 0.75] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

random_state
[int, RandomState instance or None, optional (default None)] This parameter sets the random state on this solver. If the input X is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient *randomized* solver is enabled.

colorbar
[bool, default: True] If the target_type is "continous" draw a colorbar to the right of the scatter plot. The colobar axes is accessible using the cax property.

heatmap
[bool, default: False] Add a heatmap showing contribution of each feature in the principal components. Also draws a colorbar for readability purpose. The heatmap is accessible using lax property and colorbar using uax property.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
>>> y = iris.target
>>> visualizer = PCA()
>>> visualizer.fit_transform(X, y)
>>> visualizer.show()
```

Attributes

pca_components_
[ndarray, shape (n_features, n_components)] This tells about the magnitude of each feature in the pricipal components. This is primarily used to draw the biplots.

classes_
[ndarray, shape (n_classes,)] The class labels that define the discrete values in the target. Only available if the target type is discrete. This is guaranteed to be strings even if the classes are a different type.

features_
[ndarray, shape (n_features,)] The names of the features discovered or used in the visualizer that can be used as an index to access or modify data in X. If a user passes feature names in, those features are used. Otherwise the columns of a DataFrame are used or just simply the indices of the data array.

range_
[(min y, max y)] A tuple that describes the minimum and maximum values in the target. Only available if the target type is continuous.

`draw`(*Xp*, *y*)

Plots a scatterplot of points that represented the decomposition, *pca_features_*, of the original features, *X*, projected into either 2 or 3 dimensions.

If 2 dimensions are selected, a colorbar and heatmap can also be optionally included to show the magnitude of each feature value to the component.

Parameters

Xp
[array-like of shape (n, 2) or (n, 3)] The matrix produced by the `transform()` method.

y
[array-like of shape (n,), optional] The target, used to specify the colors of the points.

Returns

self.ax
[matplotlib Axes object] Returns the axes that the scatter plot was drawn on.

`finalize`(***kwargs*)

Draws the title, labels, legends, heatmap, and colorbar as specified by the keyword arguments.

`fit`(*X*, *y=None*, ***kwargs*)

Fits the PCA transformer, transforms the data in X, then draws the decomposition in either 2D or 3D space as a scatter plot.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features.

y
[ndarray or Series of length n] An array or series of target or class values.

Returns

self
[visualizer] Returns self for use in Pipelines.

`property lax`

The axes of the heatmap below scatter plot.

`layout`(*divider=None*)

Creates the layout for colorbar and heatmap, adding new axes for the heatmap if necessary and modifying the aspect ratio. Does not modify the axes or the layout if `self.heatmap` is `False` or `None`.

Parameters

divider: AxesDivider
An AxesDivider to be passed among all layout calls.

`property random_state`

`transform`(*X*, *y=None*, ***kwargs*)

Calls the internal *transform* method of the scikit-learn PCA transformer, which performs a dimensionality reduction on the input features `X`. Next calls the `draw` method of the Yellowbrick visualizer, finally returning a new array of transformed features of shape `(len(X), projection)`.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features.

y
[ndarray or Series of length n] An array or series of target or class values.

Returns

Xp
[ndarray or DataFrame of shape n x m] Returns a new array-like object of transformed features of shape `(len(X), projection)`.

`property uax`

The axes of the colorbar, bottom of scatter plot. This is the colorbar for heatmap and not for the scatter plot.

`yellowbrick.features.pca.pca_decomposition`(*X*, *y=None*, *ax=None*, *features=None*, *classes=None*, *scale=True*, *projection=2*, *proj_features=False*, *colors=None*, *colormap=None*, *alpha=0.75*, *random_state=None*, *colorbar=True*, *heatmap=False*, *show=True*, ***kwargs*)

Produce a two or three dimensional principal component plot of the data array `X` projected onto its largest sequential principal components. It is common practice to scale the data array `X` before applying a PC decomposition. Variable scaling can be controlled using the `scale` argument.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features.

y
[ndarray or Series of length n] An array or series of target or class values.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in, the current axes will be used (or generated if required).

features
[list, default: None] The names of the features specified by the columns of the input dataset. This length of this list must match the number of columns in X, otherwise an exception will be raised on `fit()`.

classes
[list, default: None] The class labels for each class in y, ordered by sorted class index. These names act as a label encoder for the legend, identifying integer classes or renaming string labels. If omitted, the class labels will be taken from the unique values in y.

Note that the length of this list must match the number of unique values in y, otherwise an exception is raised. This parameter is only used in the discrete target type case and is ignored otherwise.

scale
[bool, default: True] Boolean that indicates if user wants to scale data.

projection
[int or string, default: 2] The number of axes to project into, either 2d or 3d. To plot 3d plots with matplotlib, please ensure a 3d axes is passed to the visualizer, otherwise one will be created using the current figure.

proj_features
[bool, default: False] Boolean that indicates if the user wants to project the features in the projected space. If True the plot will be similar to a biplot.

colors
[list or tuple, default: None] A single color to plot all instances as or a list of colors to color each instance according to its class in the discrete case or as an ordered colormap in the sequential case. If not enough colors per class are specified then the colors are treated as a cycle.

colormap
[string or cmap, default: None] The colormap used to create the individual colors. In the discrete case it is used to compute the number of colors needed for each class and in the continuous case it is used to create a sequential color map based on the range of the target.

alpha
[float, default: 0.75] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

random_state
[int, RandomState instance or None, optional (default None)] This parameter sets the random state on this solver. If the input X is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient *randomized* solver is enabled.

colorbar
[bool, default: True] If the target_type is "continous" draw a colorbar to the right of the scatter plot. The colobar axes is accessible using the cax property.

heatmap
[bool, default: False] Add a heatmap showing contribution of each feature in the principal

components. Also draws a colorbar for readability purpose. The heatmap is accessible using lax property and colorbar using uax property.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
>>> y = iris.target
>>> pca_decomposition(X, y, colors=['r', 'g', 'b'], projection=3)
```

Attributes

pca_components_
[ndarray, shape (n_features, n_components)] This tells about the magnitude of each feature in the pricipal components. This is primarily used to draw the biplots.

classes_
[ndarray, shape (n_classes,)] The class labels that define the discrete values in the target. Only available if the target type is discrete. This is guaranteed to be strings even if the classes are a different type.

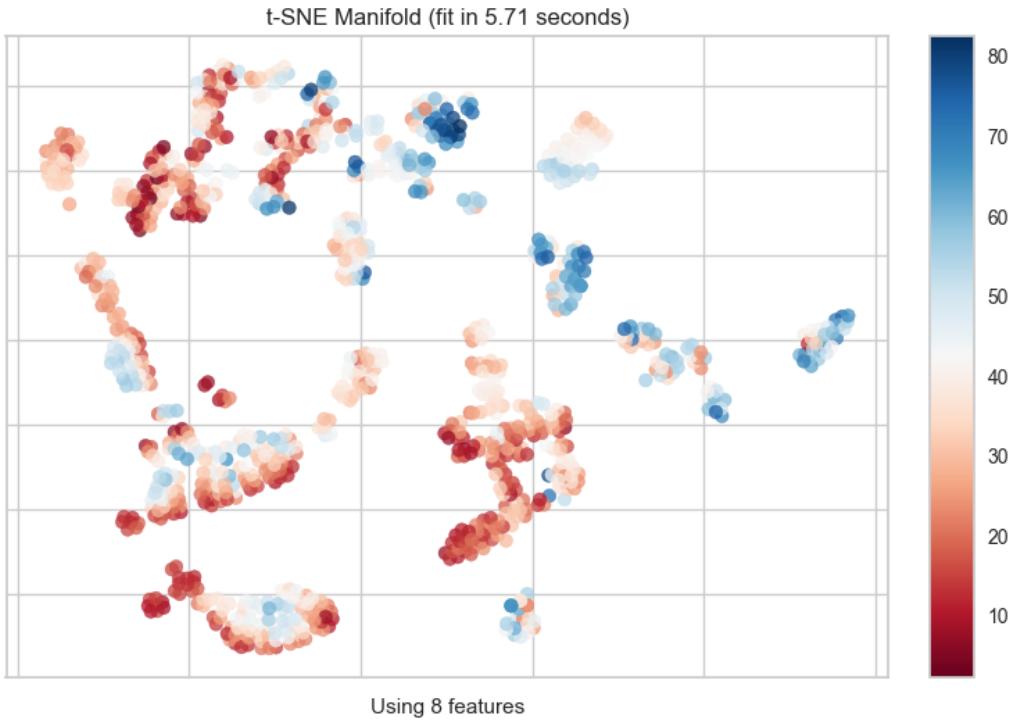
features_
[ndarray, shape (n_features,)] The names of the features discovered or used in the visualizer that can be used as an index to access or modify data in X. If a user passes feature names in, those features are used. Otherwise the columns of a DataFrame are used or just simply the indices of the data array.

range_
[(min y, max y)] A tuple that describes the minimum and maximum values in the target. Only available if the target type is continuous.

Manifold Visualization

The `Manifold` visualizer provides high dimensional visualization using manifold learning to embed instances described by many dimensions into 2, thus allowing the creation of a scatter plot that shows latent structures in data. Unlike decomposition methods such as PCA and SVD, manifolds generally use nearest-neighbors approaches to embedding, allowing them to capture non-linear structures that would be otherwise lost. The projections that are produced can then be analyzed for noise or separability to determine if it is possible to create a decision space in the data.

Visualizer	*Manifold*
Quick Method	*manifold_embedding()*
Models	Classification, Regression
Workflow	Feature Engineering



The `Manifold` visualizer allows access to all currently available scikit-learn manifold implementations by specifying the manifold as a string to the visualizer. The currently implemented default manifolds are as follows:

Manifold	Description
`"lle"`	Locally Linear Embedding (LLE) uses many local linear decompositions to preserve globally non-linear structures.
`"ltsa"`	LTSA LLE: local tangent space alignment is similar to LLE in that it uses locality to preserve neighborhood distances.
`"hessian"`	Hessian LLE an LLE regularization method that applies a hessian-based quadratic form at each neighborhood
`"modified"`	Modified LLE applies a regularization parameter to LLE.
`"isomap"`	Isomap seeks a lower dimensional embedding that maintains geometric distances between each instance.
`"mds"`	MDS: multi-dimensional scaling uses similarity to plot points that are near to each other close in the embedding.
`"spectral"`	Spectral Embedding a discrete approximation of the low dimensional manifold using a graph representation.
`"tsne"`	t-SNE: converts the similarity of points into probabilities then uses those probabilities to create an embedding.

Each manifold algorithm produces a different embedding and takes advantage of different properties of the underlying data. Generally speaking, it requires multiple attempts on new data to determine the manifold that works best for the structures latent in your data. Note however, that different manifold algorithms have different time, complexity, and resource requirements.

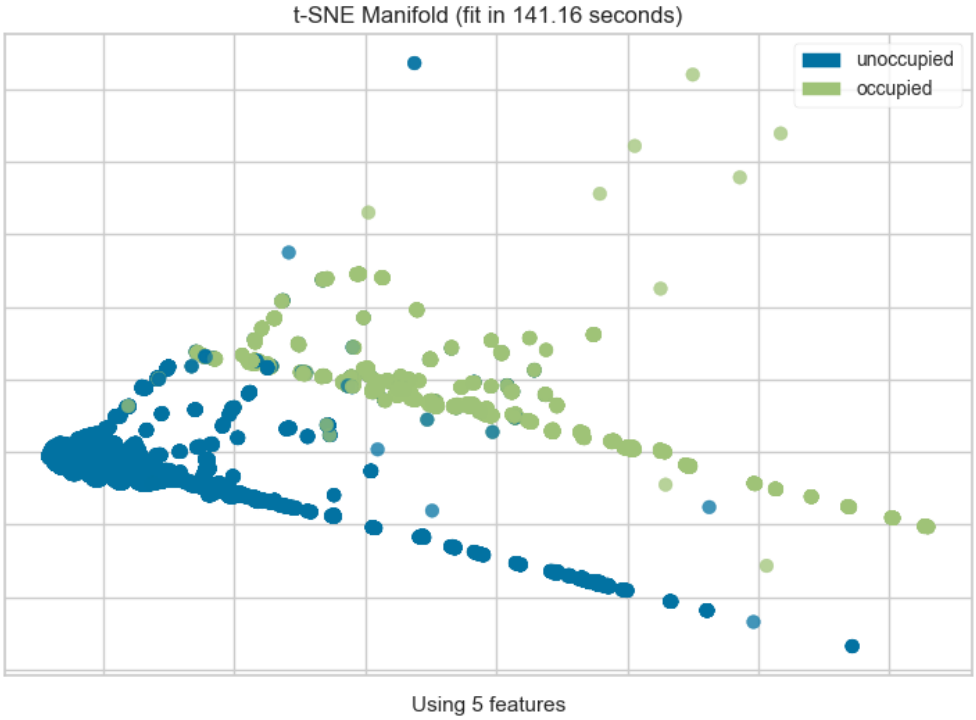
Manifolds can be used on many types of problems, and the color used in the scatter plot can describe the target instance. In an unsupervised or clustering problem, a single color is used to show structure and overlap. In a classification problem

discrete colors are used for each class. In a regression problem, a color map can be used to describe points as a heat map of their regression values.

Discrete Target

In a classification or clustering problem, the instances can be described by discrete labels - the classes or categories in the supervised problem, or the clusters they belong to in the unsupervised version. The manifold visualizes this by assigning a color to each label and showing the labels in a legend.

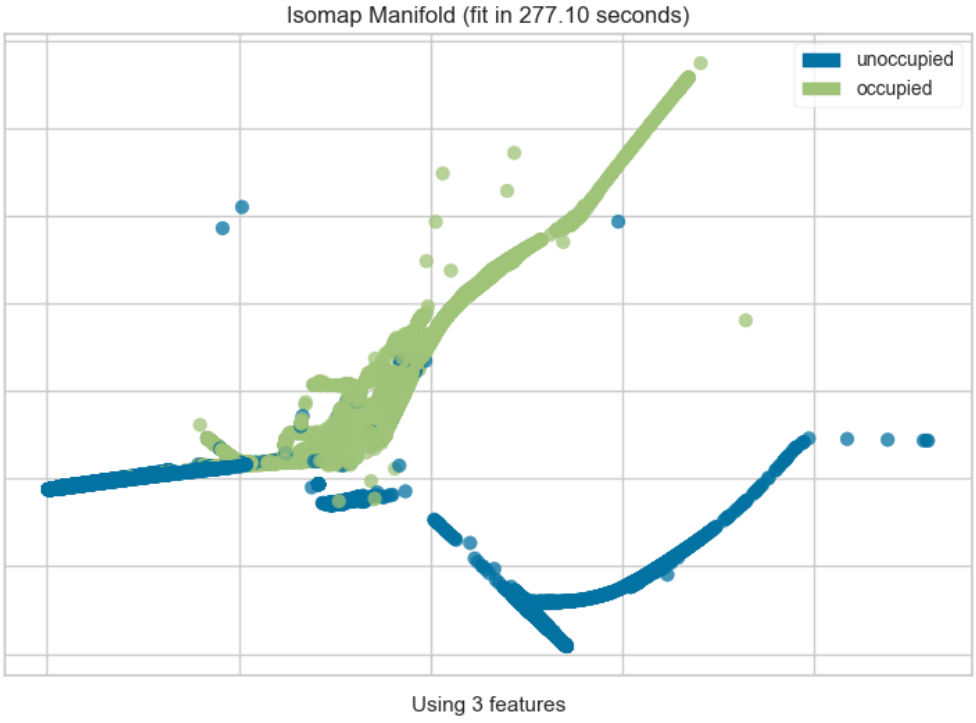
```
from yellowbrick.features import Manifold
from yellowbrick.datasets import load_occupancy

# Load the classification dataset
X, y = load_occupancy()
classes = ["unoccupied", "occupied"]

# Instantiate the visualizer
viz = Manifold(manifold="tsne", classes=classes)

viz.fit_transform(X, y)  # Fit the data to the visualizer
viz.show()               # Finalize and render the figure
```



The visualization also displays the amount of time it takes to generate the embedding; as you can see, this can take a long time even for relatively small datasets. One tip is scale your data using the `StandardScalar`; another is to sample your instances (e.g. using `train_test_split` to preserve class stratification) or to filter features to decrease sparsity in the dataset.

One common mechanism is to use `SelectKBest` to select the features that have a statistical correlation with the target dataset. For example, we can use the `f_classif` score to find the 3 best features in our occupancy dataset.

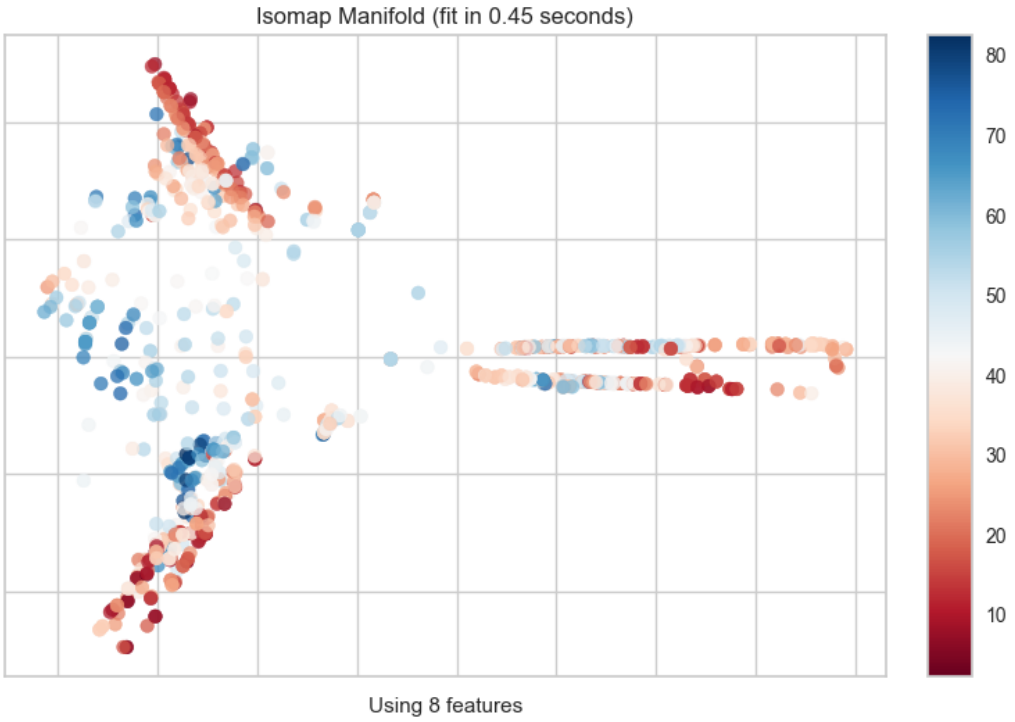
```
from sklearn.pipeline import Pipeline
from sklearn.feature_selection import f_classif, SelectKBest

from yellowbrick.features import Manifold
from yellowbrick.datasets import load_occupancy

# Load the classification dataset
X, y = load_occupancy()
classes = ["unoccupied", "occupied"]

# Create a pipeline
model = Pipeline([
    ("selectk", SelectKBest(k=3, score_func=f_classif)),
    ("viz", Manifold(manifold="isomap", n_neighbors=10, classes=classes)),
])

model.fit_transform(X, y)          # Fit the data to the model
model.named_steps['viz'].show()    # Finalize and render the figure
```



Continuous Target

For a regression target or to specify color as a heat-map of continuous values, specify `target_type="continuous"`. Note that by default the param `target_type="auto"` is set, which determines if the target is discrete or continuous by counting the number of unique values in `y`.

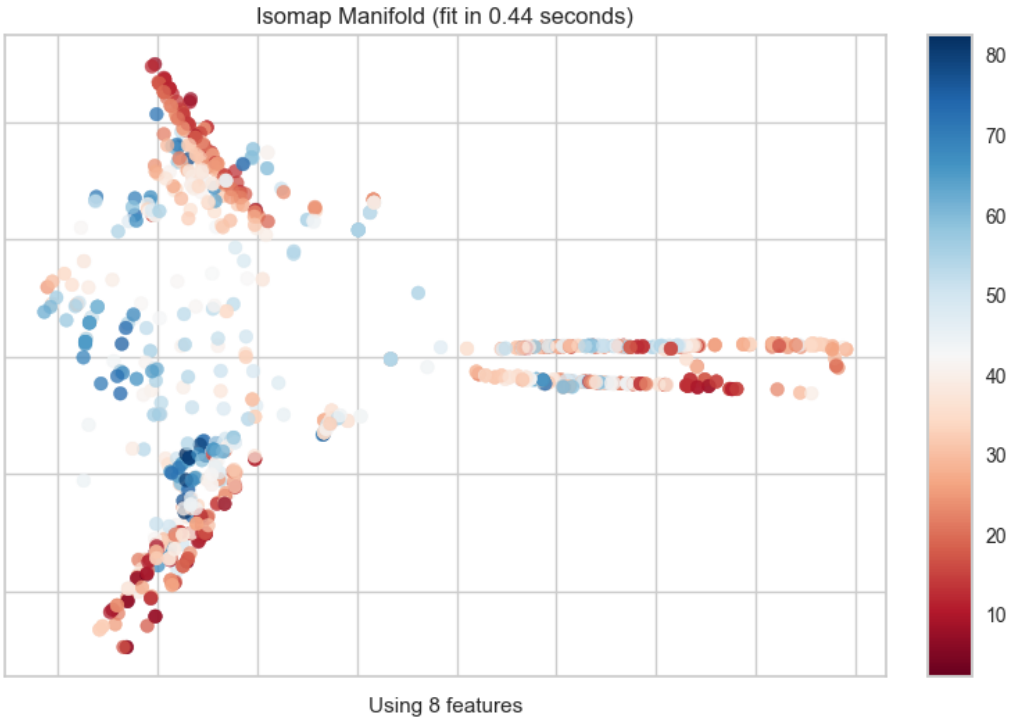
```
from yellowbrick.features import Manifold
from yellowbrick.datasets import load_concrete

# Load the regression dataset
X, y = load_concrete()

# Instantiate the visualizer
viz = Manifold(manifold="isomap", n_neighbors=10)

viz.fit_transform(X, y)  # Fit the data to the visualizer
viz.show()               # Finalize and render the figure
```



Quick Method

The same functionality above can be achieved with the associated quick method `manifold_embedding`. This method will build the `Manifold` object with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from yellowbrick.features.manifold import manifold_embedding
from yellowbrick.datasets import load_concrete

# Load the regression dataset
X, y = load_concrete()

# Instantiate the visualizer
manifold_embedding(X, y, manifold="isomap", n_neighbors=10)
```



API Reference

Use manifold algorithms for high dimensional visualization.

class yellowbrick.features.manifold.**Manifold**(*ax=None*, *manifold='mds'*, *n_neighbors=None*, *features=None*, *classes=None*, *colors=None*, *colormap=None*, *target_type='auto'*, *projection=2*, *alpha=0.75*, *random_state=None*, *colorbar=True*, ***kwargs*)

Bases: `ProjectionVisualizer`

The Manifold visualizer provides high dimensional visualization for feature analysis by embedding data into 2 dimensions using the sklearn.manifold package for manifold learning. In brief, manifold learning algorithms are unsuperivsed approaches to non-linear dimensionality reduction (unlike PCA or SVD) that help visualize latent structures in data.

The manifold algorithm used to do the embedding in scatter plot space can either be a transformer or a string representing one of the already specified manifolds as follows:

Manifold	Description
`"lle"`	Locally Linear Embedding
`"ltsa"`	LTSA LLE
`"hessian"`	Hessian LLE
`"modified"`	Modified LLE
`"isomap"`	Isomap
`"mds"`	Multi-Dimensional Scaling
`"spectral"`	Spectral Embedding
`"tsne"`	t-SNE

Each of these algorithms embeds non-linear relationships in different ways, allowing for an exploration of various structures in the feature space. Note however, that each of these algorithms has different time, memory and complexity requirements; take special care when using large datasets!

The Manifold visualizer also shows the specified target (if given) as the color of the scatter plot. If a classification or clustering target is given, then discrete colors will be used with a legend. If a regression or continuous target is specified, then a colormap and colorbar will be shown.

Parameters

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None, the current axes will be used or generated if required.

manifold
[str or Transformer, default: "mds"] Specify the manifold algorithm to perform the embedding. Either one of the strings listed in the table above, or an actual scikit-learn transformer. The constructed manifold is accessible with the manifold property, so as to modify hyperparameters before fit.

n_neighbors
[int, default: None] Many manifold algorithms are nearest neighbors based, for those that are, this parameter specfies the number of neighbors to use in the embedding. If n_neighbors is not specified for those embeddings, it is set to 5 and a warning is issued. If the manifold algorithm doesn't use nearest neighbors, then this parameter is ignored.

features
[list, default: None] The names of the features specified by the columns of the input dataset. This length of this list must match the number of columns in X, otherwise an exception will be raised on `fit()`.

classes
[list, default: None] The class labels for each class in y, ordered by sorted class index. These names act as a label encoder for the legend, identifying integer classes or renaming string labels. If omitted, the class labels will be taken from the unique values in y.

Note that the length of this list must match the number of unique values in y, otherwise an exception is raised. This parameter is only used in the discrete target type case and is ignored otherwise.

colors
[list or tuple, default: None] A single color to plot all instances as or a list of colors to color each instance according to its class in the discrete case or as an ordered colormap in the sequential case. If not enough colors per class are specified then the colors are treated as a cycle.

colormap
[string or cmap, default: None] The colormap used to create the individual colors. In the discrete case it is used to compute the number of colors needed for each class and in the continuous case it is used to create a sequential color map based on the range of the target.

target_type
[str, default: "auto"] Specify the type of target as either "discrete" (classes) or "continuous" (real numbers, usually for regression). If "auto", then it will attempt to determine the type by counting the number of unique values.

If the target is discrete, the colors are returned as a dict with classes being the keys. If continuous the colors will be list having value of color for each point. In either case, if no target is specified, then color will be specified as the first color in the color cycle.

projection
[int or string, default: 2] The number of axes to project into, either 2d or 3d. To plot 3d plots with matplotlib, please ensure a 3d axes is passed to the visualizer, otherwise one will be created using the current figure.

alpha
[float, default: 0.75] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

random_state
[int or RandomState, default: None] Fixes the random state for stochastic manifold algorithms.

colorbar
[bool, default: True] If the target_type is "continous" draw a colorbar to the right of the scatter plot. The colobar axes is accessible using the cax property.

kwargs
[dict] Keyword arguments passed to the base class and may influence the feature visualization properties.

Notes

Specifying the target as `'continuous'` or `'discrete'` will influence how the visualizer is finally displayed, don't rely on the automatic determination from the Manifold!

Scaling your data with the standard scalar before applying it to the visualizer is a great way of increasing performance. Additionally using the `SelectKBest` transformer may also improve performance and lead to better visualizations.

Warning: Manifold visualizers have extremly varying time, resource, and complexity requirements. Sampling data or features may be necessary in order to finish a manifold computation.

See also:

The Scikit-Learn discussion on Manifold Learning.

Examples

```
>>> viz = Manifold(manifold='isomap', target='discrete')
>>> viz.fit_transform(X, y)
>>> viz.show()
```

Attributes

fit_time_
[yellowbrick.utils.timer.Timer] The amount of time in seconds it took to fit the Manifold.

classes_
[ndarray, shape (n_classes,)] The class labels that define the discrete values in the target. Only available if the target type is discrete. This is guaranteed to be strings even if the classes are a different type.

features_
[ndarray, shape (n_features,)] The names of the features discovered or used in the visualizer that can be used as an index to access or modify data in X. If a user passes feature names in, those features are used. Otherwise the columns of a DataFrame are used or just simply the indices of the data array.

range_
[(min y, max y)] A tuple that describes the minimum and maximum values in the target. Only available if the target type is continuous.

ALGORITHMS = {'hessian': LocallyLinearEmbedding(method='hessian'), 'isomap': Isomap(), 'lle': LocallyLinearEmbedding(), 'ltsa': LocallyLinearEmbedding(method='ltsa'), 'mds': MDS(), 'modified': LocallyLinearEmbedding(method='modified'), 'spectral': SpectralEmbedding(), 'tsne': TSNE(init='pca')}

draw(*Xp*, *y=None*)

Draws the points described by Xp and colored by the points in y. Can be called multiple times before finalize to add more scatter plots to the axes, however `fit()` must be called before use.

Parameters

Xp
[array-like of shape (n, 2) or (n, 3)] The matrix produced by the `transform()` method.

y
[array-like of shape (n,), optional] The target, used to specify the colors of the points.

Returns

self.ax
[matplotlib Axes object] Returns the axes that the scatter plot was drawn on.

finalize()

Add title and modify axes to make the image ready for display.

fit(*X*, *y=None*, ***kwargs*)

Fits the manifold on X and transforms the data to plot it on the axes. See fit_transform() for more details.

Parameters

X
[array-like of shape (n, m)] A matrix or data frame with n instances and m features

y
[array-like of shape (n,), optional] A vector or series with target values for each instance in X. This vector is used to determine the color of the points in X.

Returns

self
[Manifold] Returns the visualizer object.

`fit_transform`(*X*, *y=None*, ***kwargs*)

Fits the manifold on X and transforms the data to plot it on the axes. The optional y specified can be used to declare discrete colors. If the target is set to 'auto', this method also determines the target type, and therefore what colors will be used.

Note also that fit records the amount of time it takes to fit the manifold and reports that information in the visualization.

Parameters

X
[array-like of shape (n, m)] A matrix or data frame with n instances and m features

y
[array-like of shape (n,), optional] A vector or series with target values for each instance in X. This vector is used to determine the color of the points in X.

Returns

Xprime
[array-like of shape (n, 2)] Returns the 2-dimensional embedding of the instances.

`property manifold`

Property containing the manifold transformer constructed from the supplied hyperparameter. Use this property to modify the manifold before fit with `manifold.set_params()`.

`transform`(*X*, *y=None*, ***kwargs*)

Returns the transformed data points from the manifold embedding.

Parameters

X
[array-like of shape (n, m)] A matrix or data frame with n instances and m features

y
[array-like of shape (n,), optional] The target, used to specify the colors of the points.

Returns

Xprime
[array-like of shape (n, 2)] Returns the 2-dimensional embedding of the instances.

`yellowbrick.features.manifold.`**`manifold_embedding`**(*X*, *y=None*, *ax=None*, *manifold='mds'*, *n_neighbors=None*, *features=None*, *classes=None*, *colors=None*, *colormap=None*, *target_type='auto'*, *projection=2*, *alpha=0.75*, *random_state=None*, *colorbar=True*, *show=True*, ***kwargs*)

Quick method for Manifold visualizer.

The Manifold visualizer provides high dimensional visualization for feature analysis by embedding data into 2 dimensions using the sklearn.manifold package for manifold learning. In brief, manifold learning algorithms are unsuperivsed approaches to non-linear dimensionality reduction (unlike PCA or SVD) that help visualize latent structures in data.

See also:

See Manifold for more details.

Parameters

X
[array-like of shape (n, m)] A matrix or data frame with n instances and m features where m > 2.

y
[array-like of shape (n,), optional] A vector or series with target values for each instance in X. This vector is used to determine the color of the points in X.

ax
[matplotlib.Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

manifold
[str or Transformer, default: "lle"] Specify the manifold algorithm to perform the embedding. Either one of the strings listed in the table above, or an actual scikit-learn transformer. The constructed manifold is accessible with the manifold property, so as to modify hyperparameters before fit.

n_neighbors
[int, default: None] Many manifold algorithms are nearest neighbors based, for those that are, this parameter specfies the number of neighbors to use in the embedding. If n_neighbors is not specified for those embeddings, it is set to 5 and a warning is issued. If the manifold algorithm doesn't use nearest neighbors, then this parameter is ignored.

features
[list, default: None] The names of the features specified by the columns of the input dataset. This length of this list must match the number of columns in X, otherwise an exception will be raised on `fit()`.

classes
[list, default: None] The class labels for each class in y, ordered by sorted class index. These names act as a label encoder for the legend, identifying integer classes or renaming string labels. If omitted, the class labels will be taken from the unique values in y.

Note that the length of this list must match the number of unique values in y, otherwise an exception is raised. This parameter is only used in the discrete target type case and is ignored otherwise.

colors
[list or tuple, default: None] A single color to plot all instances as or a list of colors to color each instance according to its class in the discrete case or as an ordered colormap in the sequential case. If not enough colors per class are specified then the colors are treated as a cycle.

colormap
[string or cmap, default: None] The colormap used to create the individual colors. In the discrete case it is used to compute the number of colors needed for each class and in the continuous case it is used to create a sequential color map based on the range of the target.

target_type
[str, default: "auto"] Specify the type of target as either "discrete" (classes) or "continuous" (real numbers, usually for regression). If "auto", then it will attempt to determine the type by counting the number of unique values.

If the target is discrete, the colors are returned as a dict with classes being the keys. If continuous the colors will be list having value of color for each point. In either case, if no target is specified, then color will be specified as the first color in the color cycle.

projection
[int or string, default: 2] The number of axes to project into, either 2d or 3d. To plot 3d plots with matplotlib, please ensure a 3d axes is passed to the visualizer, otherwise one will be created using the current figure.

alpha
[float, default: 0.75] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

random_state
[int or RandomState, default: None] Fixes the random state for stochastic manifold algorithms.

colorbar
[bool, default: True] If the target_type is "continous" draw a colorbar to the right of the scatter plot. The colobar axes is accessible using the cax property.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments passed to the base class and may influence the feature visualization properties.

Returns

viz
[Manifold] Returns the fitted, finalized visualizer

Direct Data Visualization

Sometimes for feature analysis you simply need a scatter plot to determine the distribution of data. Machine learning operates on high dimensional data, so the number of dimensions has to be filtered. As a result these visualizations are typically used as the base for larger visualizers; however you can also use them to quickly plot data during ML analysis.

Joint Plot Visualization

The `JointPlotVisualizer` plots a feature against the target and shows the distribution of each via a histogram on each axis.

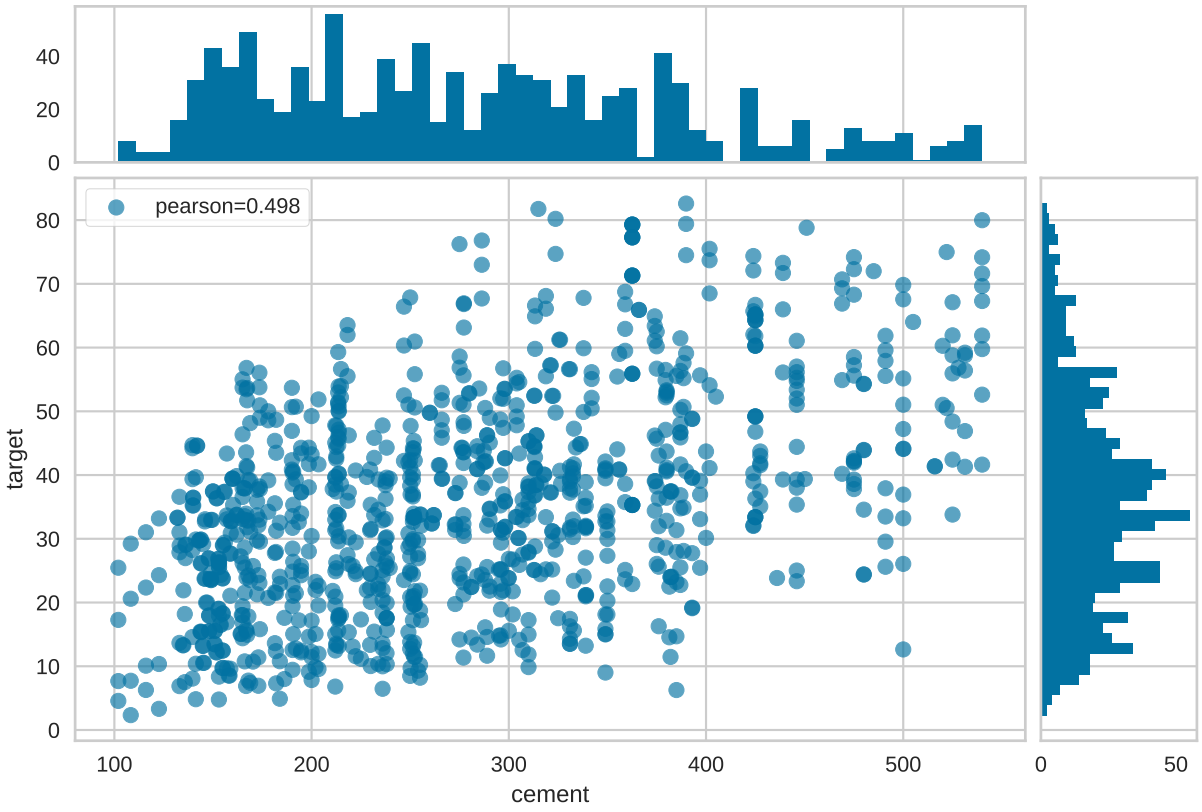
Visualizer	*JointPlot*
Quick Method	*joint_plot()*
Models	Classification/Regression
Workflow	Feature Engineering/Selection

```
from yellowbrick.datasets import load_concrete
from yellowbrick.features import JointPlotVisualizer

# Load the dataset
X, y = load_concrete()
```

(continues on next page)

(continued from previous page)

```
# Instantiate the visualizer
visualizer = JointPlotVisualizer(columns="cement")

visualizer.fit_transform(X, y)        # Fit and transform the data
visualizer.show()                     # Finalize and render the figure
```



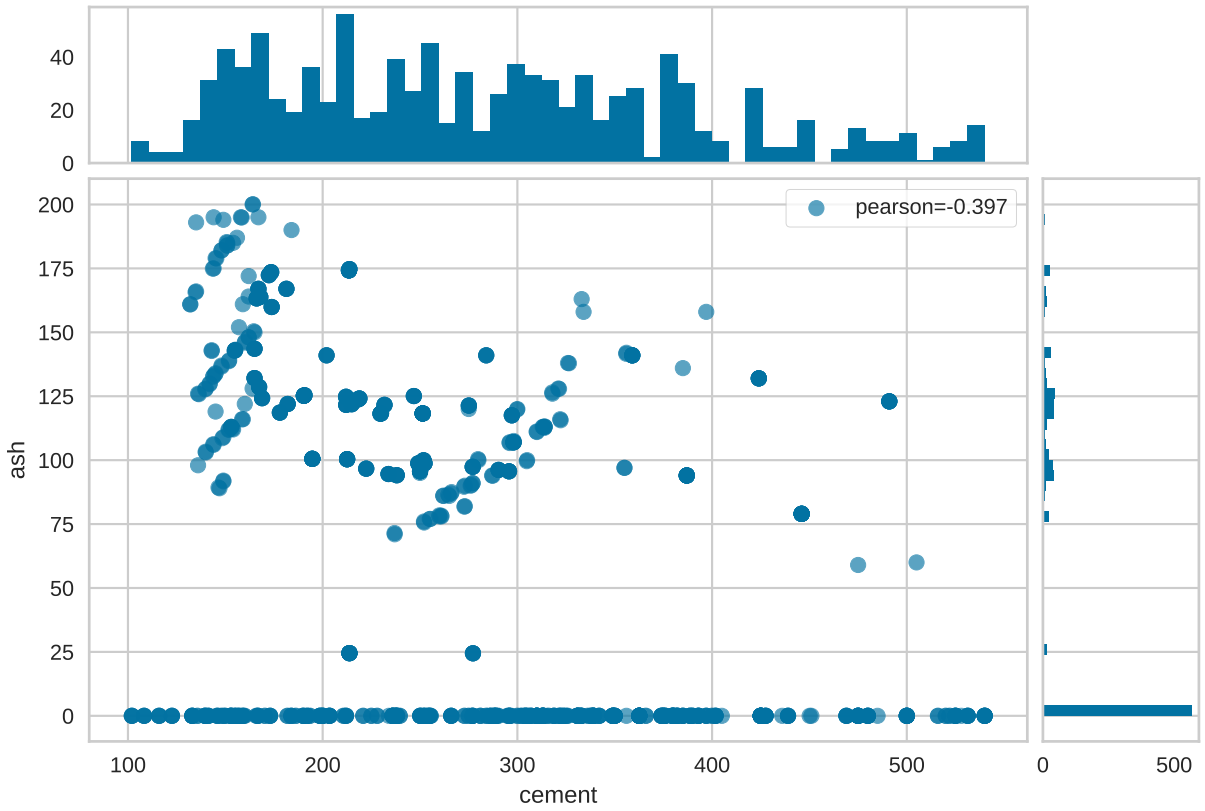
The `JointPlotVisualizer` can also be used to compare two features.

```
from yellowbrick.datasets import load_concrete
from yellowbrick.features import JointPlotVisualizer

# Load the dataset
X, y = load_concrete()

# Instantiate the visualizer
visualizer = JointPlotVisualizer(columns=["cement", "ash"])

visualizer.fit_transform(X, y)        # Fit and transform the data
visualizer.show()                     # Finalize and render the figure
```

In addition, the `JointPlotVisualizer` can be plotted with hexbins in the case of many, many points.

```
from yellowbrick.datasets import load_concrete
```

(continues on next page)



(continued from previous page)

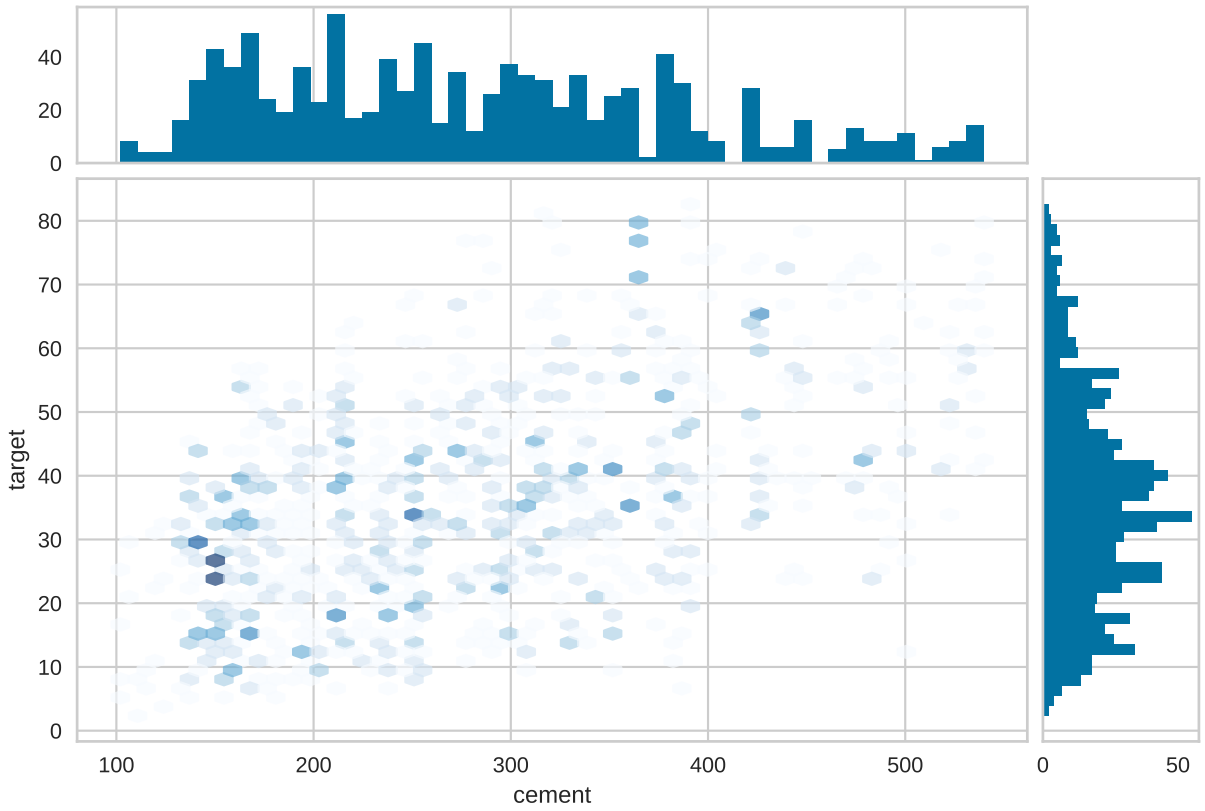
```
from yellowbrick.features import JointPlotVisualizer

# Load the dataset
X, y = load_concrete()

# Instantiate the visualizer
visualizer = JointPlotVisualizer(columns="cement", kind="hexbin")

visualizer.fit_transform(X, y)        # Fit and transform the data
visualizer.show()                     # Finalize and render the figure
```

Quick Method

The same functionality above can be achieved with the associated quick method `joint_plot`. This method will build the `JointPlot` object with the associated arguments, fit it, then (optionally) immediately show it.

```
from yellowbrick.datasets import load_concrete
from yellowbrick.features import joint_plot

# Load the dataset
X, y = load_concrete()
```

(continues on next page)



(continued from previous page)

```
# Instantiate the visualizer
visualizer = joint_plot(X, y, columns="cement")
```

API Reference

class yellowbrick.features.jointplot.**JointPlot**(*ax=None*, *columns=None*, *correlation='pearson'*, *kind='scatter'*, *hist=True*, *alpha=0.65*, *joint_kws=None*, *hist_kws=None*, ***kwargs*)

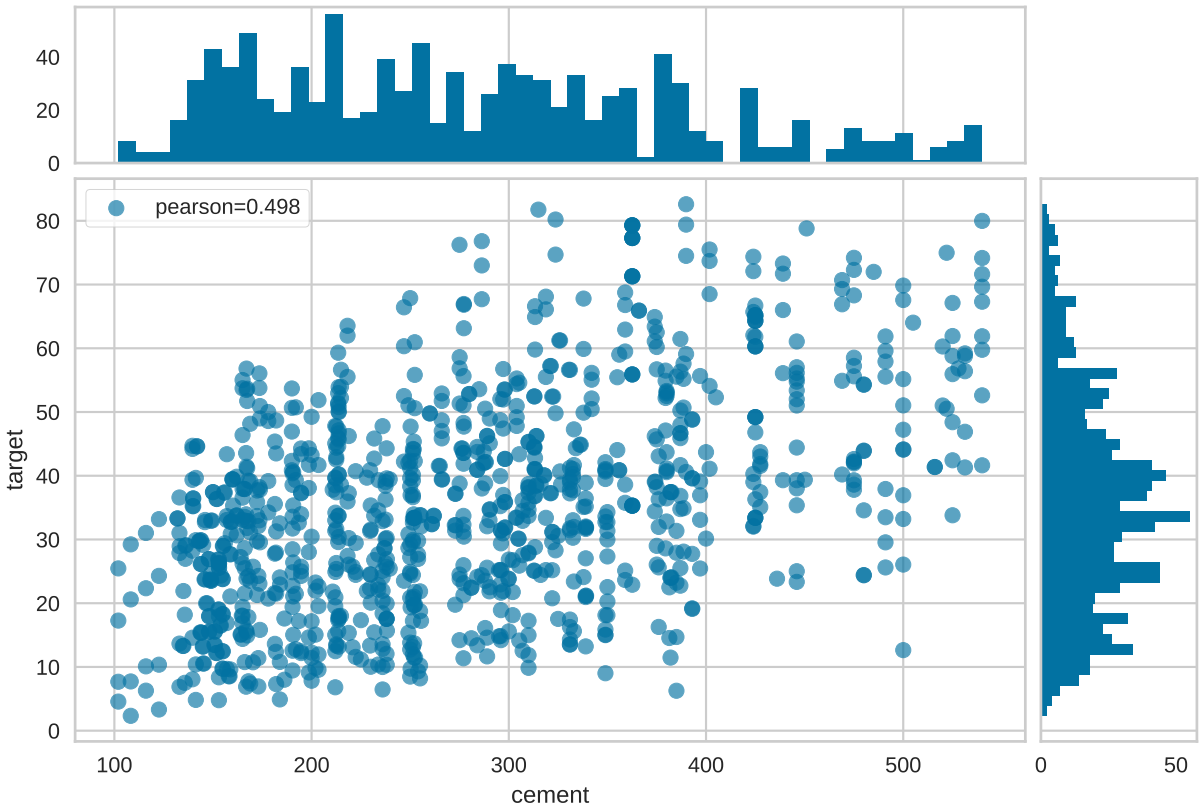
Bases: `FeatureVisualizer`

Joint plots are useful for machine learning on multi-dimensional data, allowing for the visualization of complex interactions between different data dimensions, their varying distributions, and even their relationships to the target variable for prediction.

The Yellowbrick `JointPlot` can be used both for pairwise feature analysis and feature-to-target plots. For pairwise feature analysis, the `columns` argument can be used to specify the index of the two desired columns in `X`. If `y` is also specified, the plot can be colored with a heatmap or by class. For feature-to-target plots, the user can provide either `X` and `y` as 1D vectors, or a `columns` argument with an index to a single feature in `X` to be plotted against `y`.

Histograms can be included by setting the `hist` argument to `True` for a frequency distribution, or to `"density"` for a probability density function. Note that histograms requires matplotlib 2.0.2 or greater.

Parameters



ax

[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required). This is considered the base axes where the the primary joint plot is drawn. It will be shifted and two additional axes added above (xhax) and to the right (yhax) if hist=True.

columns

[int, str, [int, int], [str, str], default: None] Determines what data is plotted in the joint plot and acts as a selection index into the data passed to `fit(X, y)`. This data therefore must be indexable by the column type (e.g. an int for a numpy array or a string for a DataFrame).

If None is specified then either both X and y must be 1D vectors and they will be plotted against each other or X must be a 2D array with only 2 columns. If a single index is specified then the data is indexed as `X[columns]` and plotted jointly with the target variable, y. If two indices are specified then they are both selected from X, additionally in this case, if y is specified, then it is used to plot the color of points.

Note that these names are also used as the x and y axes labels if they aren't specified in the joint_kws argument.

correlation

[str, default: 'pearson'] The algorithm used to compute the relationship between the variables in the joint plot, one of: 'pearson', 'covariance', 'spearman', 'kendalltau'.

kind

[str in {'scatter', 'hex'}, default: 'scatter'] The type of plot to render in the joint axes. Note that when kind='hex' the target cannot be plotted by color.

hist
[{True, False, None, 'density', 'frequency'}, default: True] Draw histograms showing the distribution of the variables plotted jointly. If set to 'density', the probability density function will be plotted. If set to True or 'frequency' then the frequency will be plotted. Requires Matplotlib >= 2.0.2.

alpha
[float, default: 0.65] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

{joint, hist}_kws
[dict, default: None] Additional keyword arguments for the plot components.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> viz = JointPlot(columns=["temp", "humidity"])
>>> viz.fit(X, y)
>>> viz.show()
```

Attributes

corr_
[float] The correlation or relationship of the data in the joint plot, specified by the correlation algorithm.

correlation_methods = {'covariance': <function JointPlot.<lambda>>, 'kendalltau': <function JointPlot.<lambda>>, 'pearson': <function JointPlot.<lambda>>, 'spearman': <function JointPlot.<lambda>>}

draw(*x*, *y*, *xlabel=None*, *ylabel=None*)

Draw the joint plot for the data in x and y.

Parameters

x, y
[1D array-like] The data to plot for the x axis and the y axis

xlabel, ylabel
[str] The labels for the x and y axes.

finalize(***kwargs*)

Finalize executes any remaining image modifications making it ready to show.

fit(*X*, *y=None*)

Fits the JointPlot, creating a correlative visualization between the columns specified during initialization and the data and target passed into fit:

- If self.columns is None then X and y must both be specified as 1D arrays or X must be a 2D array with only 2 columns.
- If self.columns is a single int or str, that column is selected to be visualized against the target y.
- If self.columns is two ints or strs, those columns are visualized against each other. If y is specified then it is used to color the points.

This is the main entry point into the joint plot visualization.

Parameters

X
[array-like] An array-like object of either 1 or 2 dimensions depending on self.columns. Usually this is a 2D table with shape (n, m)

y
[array-like, default: None] An vector or 1D array that has the same length as X. May be used to either directly plot data or to color data points.

property xhax

The axes of the histogram for the top of the JointPlot (X-axis)

property yhax

The axes of the histogram for the right of the JointPlot (Y-axis)

yellowbrick.features.jointplot.**joint_plot**(*X*, *y*, *ax=None*, *columns=None*, *correlation='pearson'*, *kind='scatter'*, *hist=True*, *alpha=0.65*, *joint_kws=None*, *hist_kws=None*, *show=True*, ***kwargs*)

Joint plots are useful for machine learning on multi-dimensional data, allowing for the visualization of complex interactions between different data dimensions, their varying distributions, and even their relationships to the target variable for prediction.

The Yellowbrick `JointPlot` can be used both for pairwise feature analysis and feature-to-target plots. For pairwise feature analysis, the `columns` argument can be used to specify the index of the two desired columns in `X`. If `y` is also specified, the plot can be colored with a heatmap or by class. For feature-to-target plots, the user can provide either `X` and `y` as 1D vectors, or a `columns` argument with an index to a single feature in `X` to be plotted against `y`.

Histograms can be included by setting the `hist` argument to `True` for a frequency distribution, or to `"density"` for a probability density function. Note that histograms requires matplotlib 2.0.2 or greater.

Parameters

X
[array-like] An array-like object of either 1 or 2 dimensions depending on self.columns. Usually this is a 2D table with shape (n, m)

y
[array-like, default: None] An vector or 1D array that has the same length as X. May be used to either directly plot data or to color data points.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required). This is considered the base axes where the the primary joint plot is drawn. It will be shifted and two additional axes added above (xhax) and to the right (yhax) if hist=True.

columns
[int, str, [int, int], [str, str], default: None] Determines what data is plotted in the joint plot and acts as a selection index into the data passed to `fit(X, y)`. This data therefore must be indexable by the column type (e.g. an int for a numpy array or a string for a DataFrame).

If None is specified then either both X and y must be 1D vectors and they will be plotted against each other or X must be a 2D array with only 2 columns. If a single index is specified then the data is indexed as `X[columns]` and plotted jointly with the target variable, y. If two indices are specified then they are both selected from X, additionally in this case, if y is specified, then it is used to plot the color of points.

Note that these names are also used as the x and y axes labels if they aren't specified in the joint_kws argument.

correlation
[str, default: 'pearson'] The algorithm used to compute the relationship between the variables in the joint plot, one of: 'pearson', 'covariance', 'spearman', 'kendalltau'.

kind
[str in {'scatter', 'hex'}, default: 'scatter'] The type of plot to render in the joint axes. Note that when kind='hex' the target cannot be plotted by color.

hist
[{True, False, None, 'density', 'frequency'}, default: True] Draw histograms showing the distribution of the variables plotted jointly. If set to 'density', the probability density function will be plotted. If set to True or 'frequency' then the frequency will be plotted. Requires Matplotlib >= 2.0.2.

alpha
[float, default: 0.65] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

{joint, hist}_kws
[dict, default: None] Additional keyword arguments for the plot components.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Attributes

corr_
[float] The correlation or relationship of the data in the joint plot, specified by the correlation algorithm.

8.3.4 Target Visualizers

Target visualizers specialize in visually describing the dependent variable for supervised modeling, often referred to as y or the target.

The following visualizations are currently implemented:

- *Balanced Binning Reference*: Generate histogram with vertical lines showing the recommended value point to bin data into evenly distributed bins.
- *Class Balance*: Visual inspection of the target to show the support of each class to the final estimator.
- *Feature Correlation*: Plot correlation between features and dependent variables.

```
# Target Visualizers Imports
from yellowbrick.target import BalancedBinningReference
from yellowbrick.target import ClassBalance
from yellowbrick.target import FeatureCorrelation
```

Balanced Binning Reference

Frequently, machine learning problems in the real world suffer from the curse of dimensionality; you have fewer training instances than you'd like and the predictive signal is distributed (often unpredictably!) across many different features.

Sometimes when the your target variable is continuously-valued, there simply aren't enough instances to predict these values to the precision of regression. In this case, we can sometimes transform the regression problem into a classification problem by binning the continuous values into makeshift classes.

To help the user select the optimal number of bins, the `BalancedBinningReference` visualizer takes the target variable `y` as input and generates a histogram with vertical lines indicating the recommended value points to ensure that the data is evenly distributed into each bin.

Visualizer	*BalancedBinningReference*
Quick Method	*balanced_binning_reference()*
Models	Classification
Workflow	Feature analysis, Target analysis, Model selection

```
from yellowbrick.datasets import load_concrete
from yellowbrick.target import BalancedBinningReference

# Load the concrete dataset
X, y = load_concrete()

# Instantiate the visualizer
visualizer = BalancedBinningReference()

visualizer.fit(y)        # Fit the data to the visualizer
visualizer.show()        # Finalize and render the figure
```

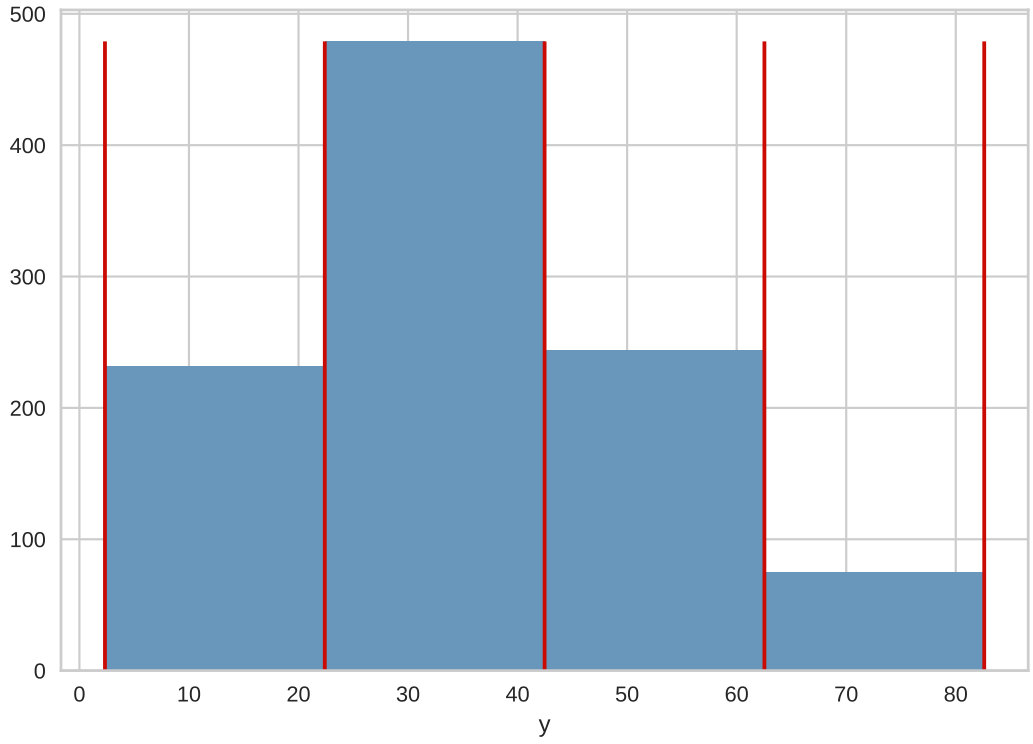
Quick Method

The same functionality above can be achieved with the associated quick method `balanced_binning_reference`. This method will build the `BalancedBinningReference` object with the associated arguments, fit it, then (optionally) immediately show it.

```
from yellowbrick.datasets import load_concrete
from yellowbrick.target import balanced_binning_reference

# Load the dataset
X, y = load_concrete()

# Use the quick method and immediately show the figure
balanced_binning_reference(y)
```

See also:

To learn more, please read Rebecca Bilbro's article "Creating Categorical Variables from Continuous Data."



API Reference

Implements histogram with vertical lines to help with balanced binning.

class yellowbrick.target.binning.**BalancedBinningReference**(*ax=None*, *target=None*, *bins=4*, ***kwargs*)

Bases: TargetVisualizer

BalancedBinningReference generates a histogram with vertical lines showing the recommended value point to bin your data so they can be evenly distributed in each bin.

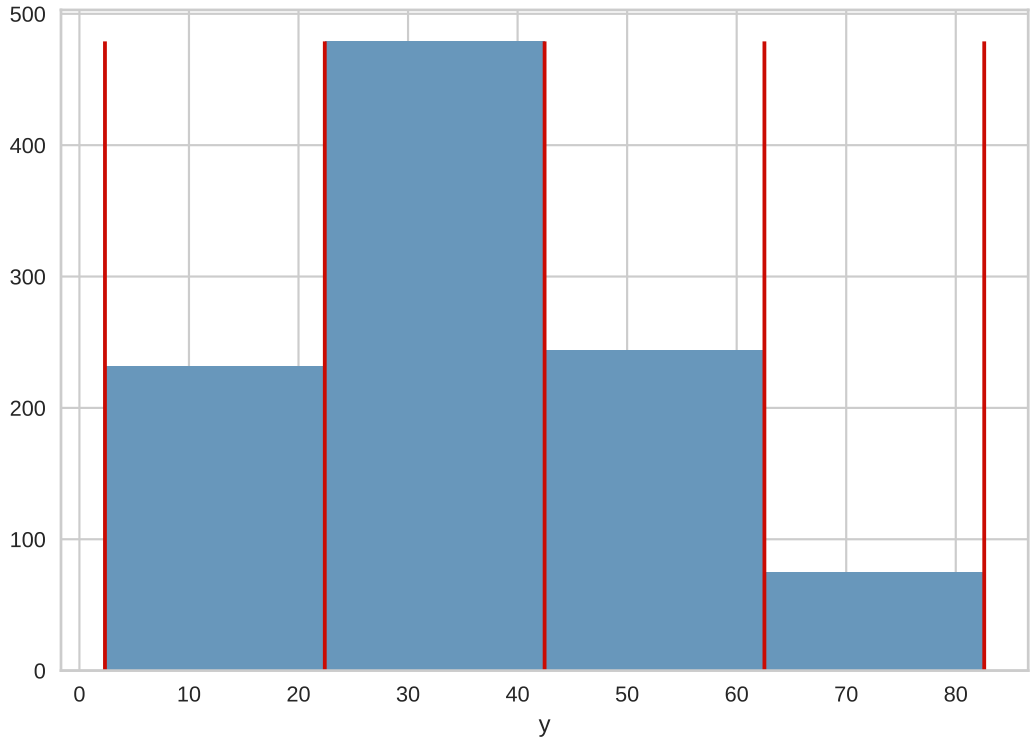
Parameters

ax
[matplotlib Axes, default: None] This is inherited from FeatureVisualizer and is defined within BalancedBinningReference.

target
[string, default: "y"] The name of the y variable

bins
[number of bins to generate the histogram, default: 4]

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.



Notes

These parameters can be influenced later on in the visualization process, but can and should be set as early as possible.

Examples

```
>>> visualizer = BalancedBinningReference()
>>> visualizer.fit(y)
>>> visualizer.show()
```

Attributes

bin_edges_
[binning reference values]

draw(*y*, ***kwargs*)

Draws a histogram with the reference value for binning as vertical lines.

Parameters

y
[an array of one dimension or a pandas Series]

finalize(***kwargs*)

Adds the x-axis label and manages the tick labels to ensure they're visible.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

fit(*y*, ***kwargs*)

Sets up y for the histogram and checks to ensure that `y` is of the correct data type. Fit calls draw.

Parameters

y
[an array of one dimension or a pandas Series]

kwargs
[dict] keyword arguments passed to scikit-learn API.

yellowbrick.target.binning.**balanced_binning_reference**(*y*, *ax=None*, *target='y'*, *bins=4*, *show=True*, ***kwargs*)

BalancedBinningReference generates a histogram with vertical lines showing the recommended value point to bin your data so they can be evenly distributed in each bin.

Parameters

y
[an array of one dimension or a pandas Series]

ax
[matplotlib Axes, default: None] This is inherited from FeatureVisualizer and is defined within `BalancedBinningReference`.

target
[string, default: "y"] The name of the `y` variable

bins
[number of bins to generate the histogram, default: 4]

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()`. However, you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

visualizer
[BalancedBinningReference] Returns fitted visualizer

Class Balance

One of the biggest challenges for classification models is an imbalance of classes in the training data. Severe class imbalances may be masked by relatively good F1 and accuracy scores – the classifier is simply guessing the majority class and not making any evaluation on the underrepresented class.

There are several techniques for dealing with class imbalance such as stratified sampling, down sampling the majority class, weighting, etc. But before these actions can be taken, it is important to understand what the class balance is in the training data. The `ClassBalance` visualizer supports this by creating a bar chart of the *support* for each class, that is the frequency of the classes' representation in the dataset.

Visualizer	*ClassBalance*
Quick Method	*class_balance()*
Models	Classification
Workflow	Feature analysis, Target analysis, Model selection

```
from yellowbrick.datasets import load_game
from yellowbrick.target import ClassBalance

# Load the classification dataset
X, y = load_game()

# Instantiate the visualizer
visualizer = ClassBalance(labels=["draw", "loss", "win"])

visualizer.fit(y)        # Fit the data to the visualizer
visualizer.show()        # Finalize and render the figure
```

The resulting figure allows us to diagnose the severity of the balance issue. In this figure we can see that the `"win"` class dominates the other two classes. One potential solution might be to create a binary classifier: `"win"` vs `"not win"` and combining the `"loss"` and `"draw"` classes into one class.

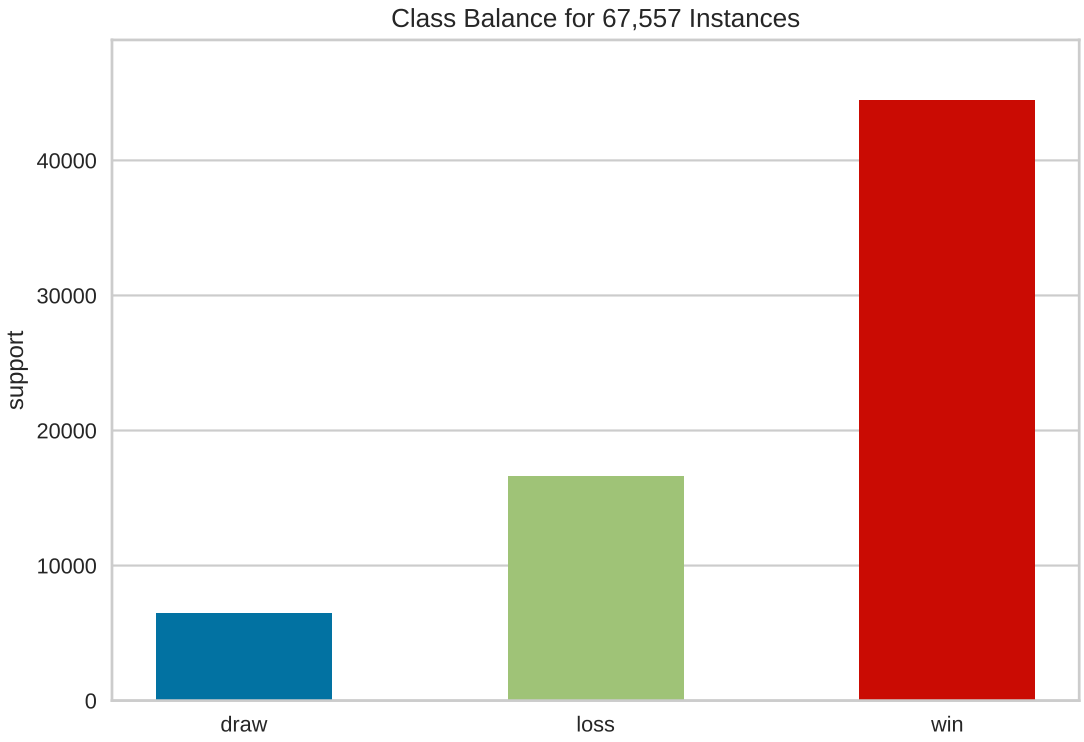
Warning: The `ClassBalance` visualizer interface has changed in version 0.9, a classification model is no longer required to instantiate the visualizer, it can operate on data only. Additionally, the signature of the fit method has changed from `fit(X, y=None)` to `fit(y_train, y_test=None)`, passing in `X` is no longer required.

If a class imbalance must be maintained during evaluation (e.g. the event being classified is actually as rare as the frequency implies) then *stratified sampling* should be used to create train and test splits. This ensures that the test data has roughly the same proportion of classes as the training data. While scikit-learn does this by default in `train_test_split` and other `cv` methods, it can be useful to compare the support of each class in both splits.

The `ClassBalance` visualizer has a "compare" mode, where the train and test data can be passed to `fit()`, creating a side-by-side bar chart instead of a single bar chart as follows:

```
from sklearn.model_selection import TimeSeriesSplit

from yellowbrick.datasets import load_occupancy
from yellowbrick.target import ClassBalance

# Load the classification dataset
X, y = load_occupancy()
```

(continues on next page)

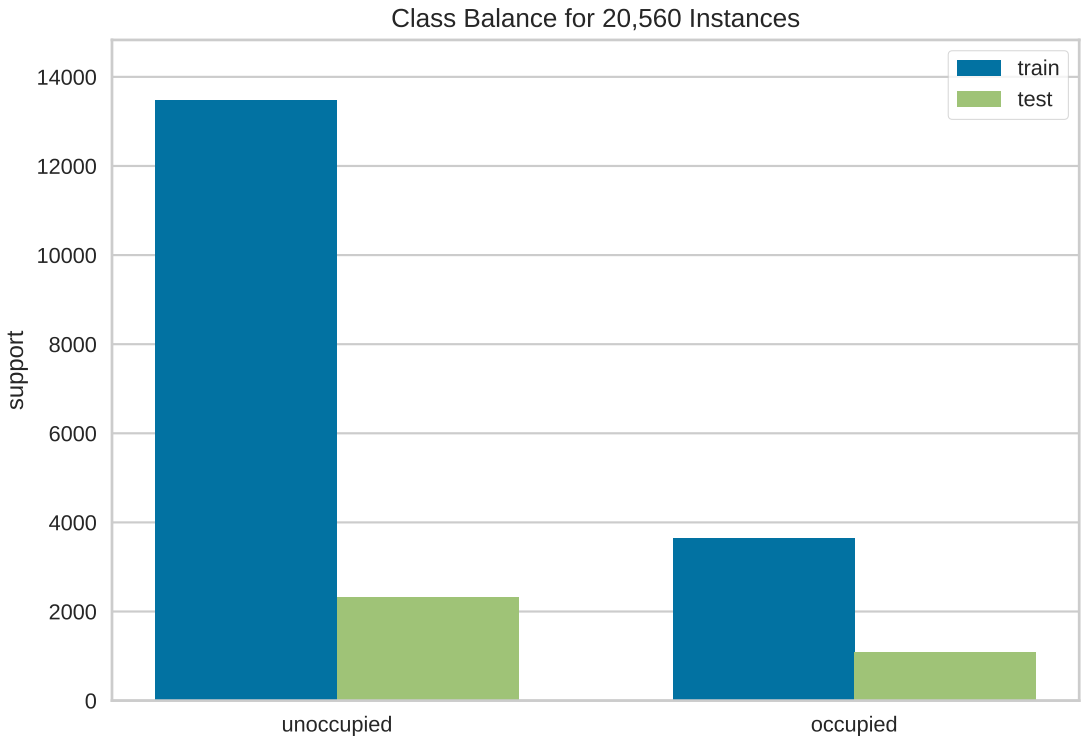


(continued from previous page)

```
# Create the training and test data
tscv = TimeSeriesSplit()
for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

# Instantiate the visualizer
visualizer = ClassBalance(labels=["unoccupied", "occupied"])

visualizer.fit(y_train, y_test)        # Fit the data to the visualizer
visualizer.show()                      # Finalize and render the figure
```

This visualization allows us to do a quick check to ensure that the proportion of each class is roughly similar in both splits. This visualization should be a first stop particularly when evaluation metrics are highly variable across different splits.

Note: This example uses `TimeSeriesSplit` to split the data into the training and test sets. For more information on this cross-validation method, please refer to the scikit-learn documentation.



Quick Method

The same functionalities above can be achieved with the associated quick method *class_balance*. This method will build the `ClassBalance` object with the associated arguments, fit it, then (optionally) immediately show it.

```
from yellowbrick.datasets import load_game
from yellowbrick.target import class_balance

# Load the dataset
X, y = load_game()

# Use the quick method and immediately show the figure
class_balance(y)
```

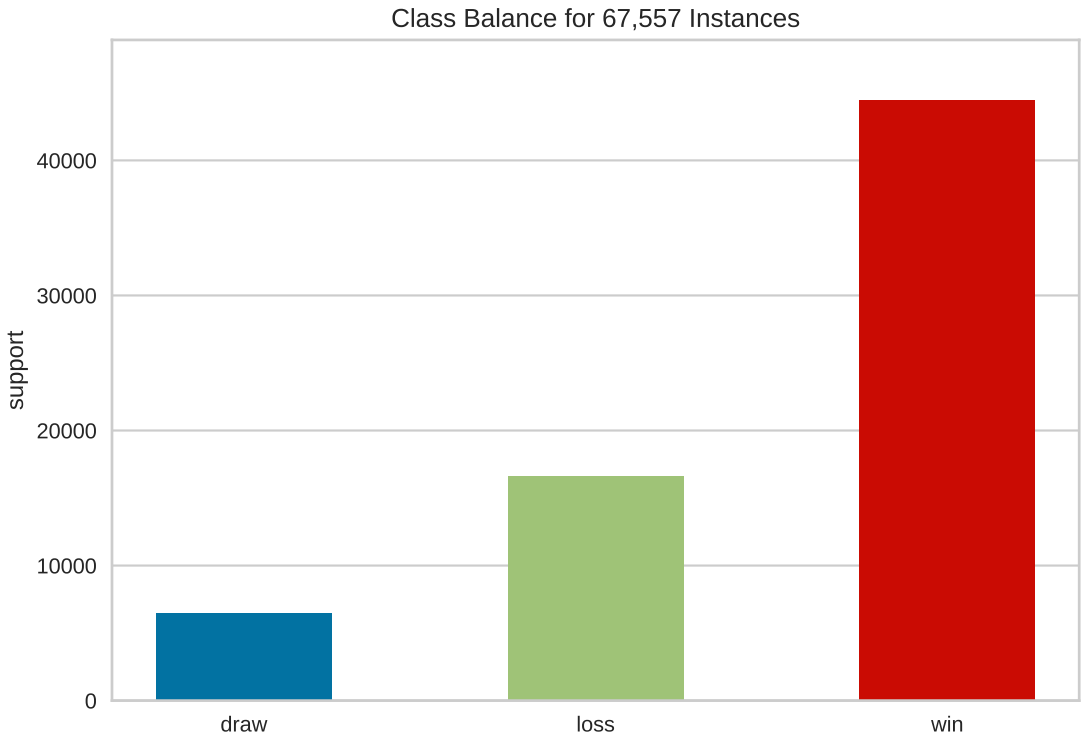
API Reference

Class balance visualizer for showing per-class support.

class yellowbrick.target.class_balance.**ClassBalance**(*ax=None*, *labels=None*, *colors=None*, *colormap=None*, ***kwargs*)

Bases: `TargetVisualizer`

One of the biggest challenges for classification models is an imbalance of classes in the training data. The ClassBalance visualizer shows the relationship of the support for each class in both the training and test data by displaying how frequently each class occurs as a bar graph.



The ClassBalance visualizer can be displayed in two modes:

1. Balance mode: show the frequency of each class in the dataset.
2. Compare mode: show the relationship of support in train and test data.

These modes are determined by what is passed to the `fit()` method.

Parameters

ax
: [matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

labels: list, optional
: A list of class names for the x-axis if the target is already encoded. Ensure that the labels are ordered lexicographically with respect to the values in the target. A common use case is to pass `LabelEncoder.classes_` as this parameter. If not specified, the labels in the data will be used.

colors: list of strings
: Specify colors for the barchart (will override colormap if both are provided).

colormap
: [string or matplotlib cmap] Specify a colormap to color the classes.

kwargs: dict, optional
: Keyword arguments passed to the super class. Here, used to colorize the bars in the histogram.

Examples

To simply observe the balance of classes in the target:

```
>>> viz = ClassBalance().fit(y)
>>> viz.show()
```

To compare the relationship between training and test data:

```
>>> _, _, y_train, y_test = train_test_split(X, y, test_size=0.2)
>>> viz = ClassBalance()
>>> viz.fit(y_train, y_test)
>>> viz.show()
```

Attributes

classes_
[array-like] The actual unique classes discovered in the target.

support_
[array of shape (n_classes,) or (2, n_classes)] A table representing the support of each class in the target. It is a vector when in balance mode, or a table with two rows in compare mode.

draw()

Renders the class balance chart on the specified axes from support.

finalize(***kwargs*)

Finalizes the figure for drawing by setting a title, the legend, and axis labels, removing the grid, and making sure the figure is correctly zoomed into the bar chart.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

fit(*y_train*, *y_test=None*)

Fit the visualizer to the the target variables, which must be 1D vectors containing discrete (classification) data. Fit has two modes:

1. Balance mode: if only y_train is specified
2. Compare mode: if both train and test are specified

In balance mode, the bar chart is displayed with each class as its own color. In compare mode, a side-by-side bar chart is displayed colored by train or test respectively.

Parameters

y_train
[array-like] Array or list of shape (n,) that contains discrete data.

y_test
[array-like, optional] Array or list of shape (m,) that contains discrete data. If specified, the bar chart will be drawn in compare mode.

yellowbrick.target.class_balance.**class_balance**(*y_train*, *y_test=None*, *ax=None*, *labels=None*, *color=None*, *colormap=None*, *show=True*, ***kwargs*)

Quick method:

One of the biggest challenges for classification models is an imbalance of classes in the training data. This function vizualizes the relationship of the support for each class in both the training and test data by displaying how frequently each class occurs as a bar graph.

The figure can be displayed in two modes:

1. Balance mode: show the frequency of each class in the dataset.
2. Compare mode: show the relationship of support in train and test data.

Balance mode is the default if only y_train is specified. Compare mode happens when both y_train and y_test are specified.

Parameters

y_train
[array-like] Array or list of shape (n,) that containes discrete data.

y_test
[array-like, optional] Array or list of shape (m,) that contains discrete data. If specified, the bar chart will be drawn in compare mode.

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

labels: list, optional
A list of class names for the x-axis if the target is already encoded. Ensure that the labels are ordered lexicographically with respect to the values in the target. A common use case is to pass `LabelEncoder.classes_` as this parameter. If not specified, the labels in the data will be used.

colors: list of strings
Specify colors for the barchart (will override colormap if both are provided).

colormap
[string or matplotlib cmap] Specify a colormap to color the classes.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs: dict, optional
Keyword arguments passed to the super class. Here, used to colorize the bars in the histogram.

Returns

visualizer
[ClassBalance] Returns the fitted visualizer

Feature Correlation

This visualizer calculates Pearson correlation coefficients and mutual information between features and the dependent variable. This visualization can be used in feature selection to identify features with high correlation or large mutual information with the dependent variable.

Pearson Correlation

The default calculation is Pearson correlation, which is performed with `scipy.stats.pearsonr`.

Visualizer	*FeatureCorrelation*
Quick Method	*feature_correlation()*
Models	Regression/Classification/Clustering
Workflow	Feature Engineering/Model Selection

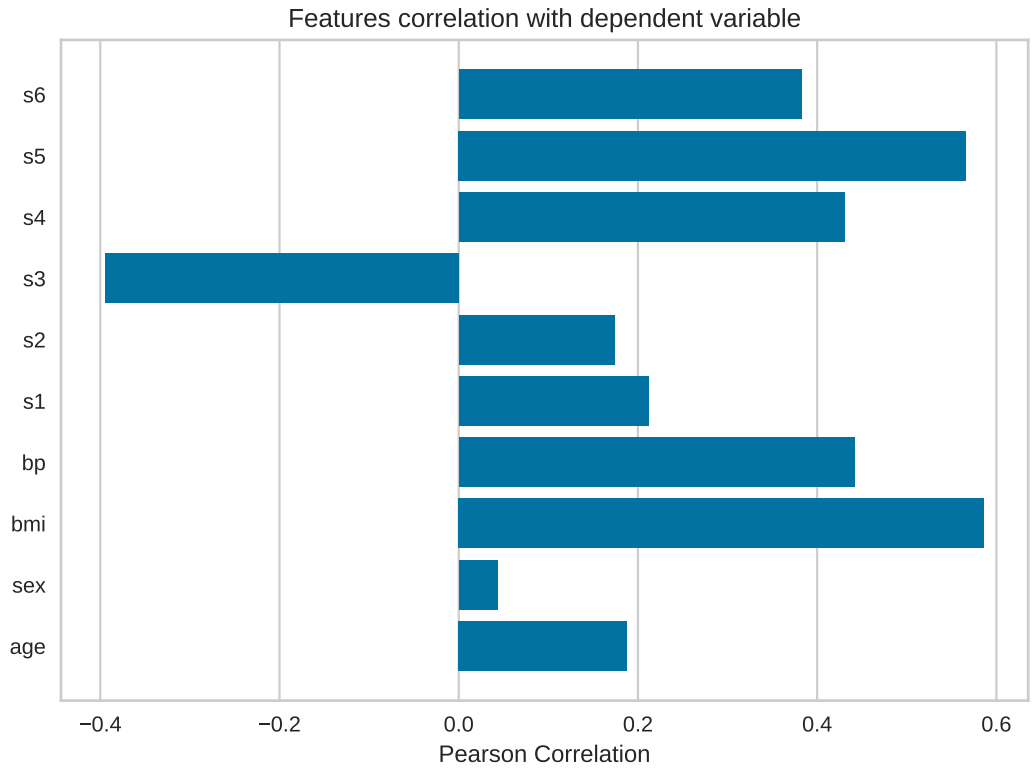
```
from sklearn import datasets
from yellowbrick.target import FeatureCorrelation

# Load the regression dataset
data = datasets.load_diabetes()
X, y = data['data'], data['target']

# Create a list of the feature names
features = np.array(data['feature_names'])

# Instantiate the visualizer
visualizer = FeatureCorrelation(labels=features)

visualizer.fit(X, y)        # Fit the data to the visualizer
visualizer.show()           # Finalize and render the figure
```

Mutual Information - Regression

Mutual information between features and the dependent variable is calculated with `sklearn.feature_selection.mutual_info_classif` when `method='mutual_info-classification'` and `mutual_info_regression` when `method='mutual_info-regression'`. It is very important to specify discrete features when calculating mutual information because the calculation for continuous and discrete variables are different. See scikit-learn documentation for more details.

```
from sklearn import datasets
from yellowbrick.target import FeatureCorrelation

# Load the regression dataset
data = datasets.load_diabetes()
X, y = data['data'], data['target']

# Create a list of the feature names
features = np.array(data['feature_names'])

# Create a list of the discrete features
```

(continues on next page)



(continued from previous page)

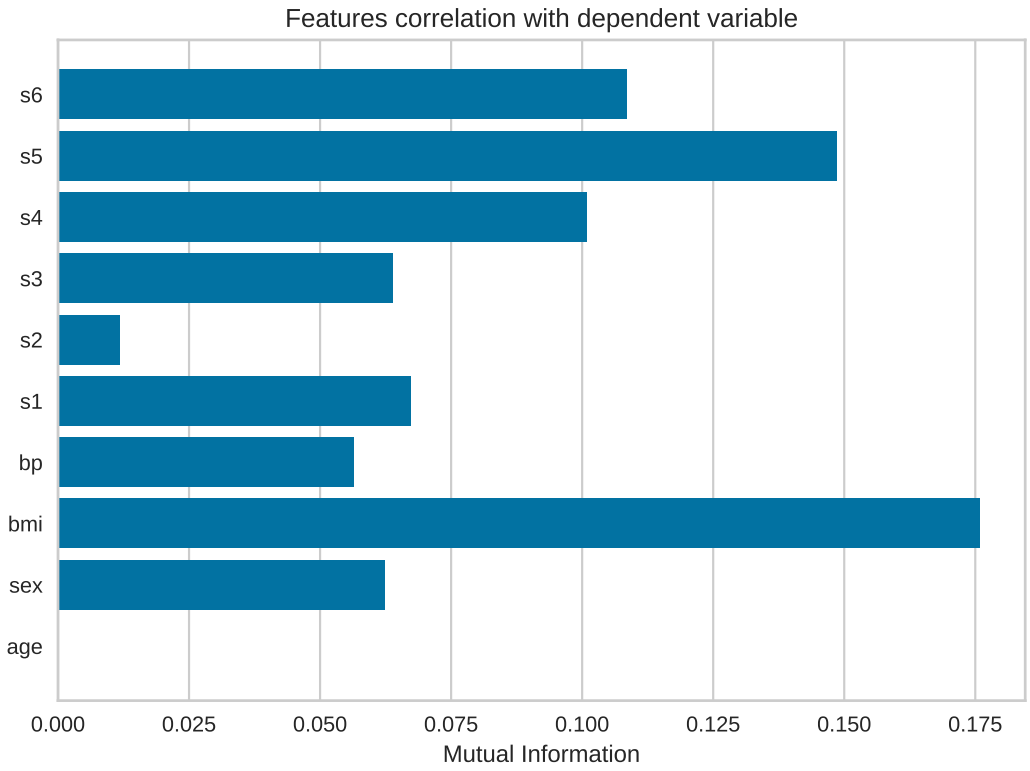
```
discrete = [False for _ in range(len(features))]
discrete[1] = True

# Instantiate the visualizer
visualizer = FeatureCorrelation(method='mutual_info-regression', labels=features)

visualizer.fit(X, y, discrete_features=discrete, random_state=0)
visualizer.show()
```

Mutual Information - Classification

By fitting with a pandas DataFrame, the feature labels are automatically obtained from the column names. This visualizer also allows sorting of the bar plot according to the calculated mutual information (or Pearson correlation coefficients) and selecting features to plot by specifying the names of the features or the feature index.

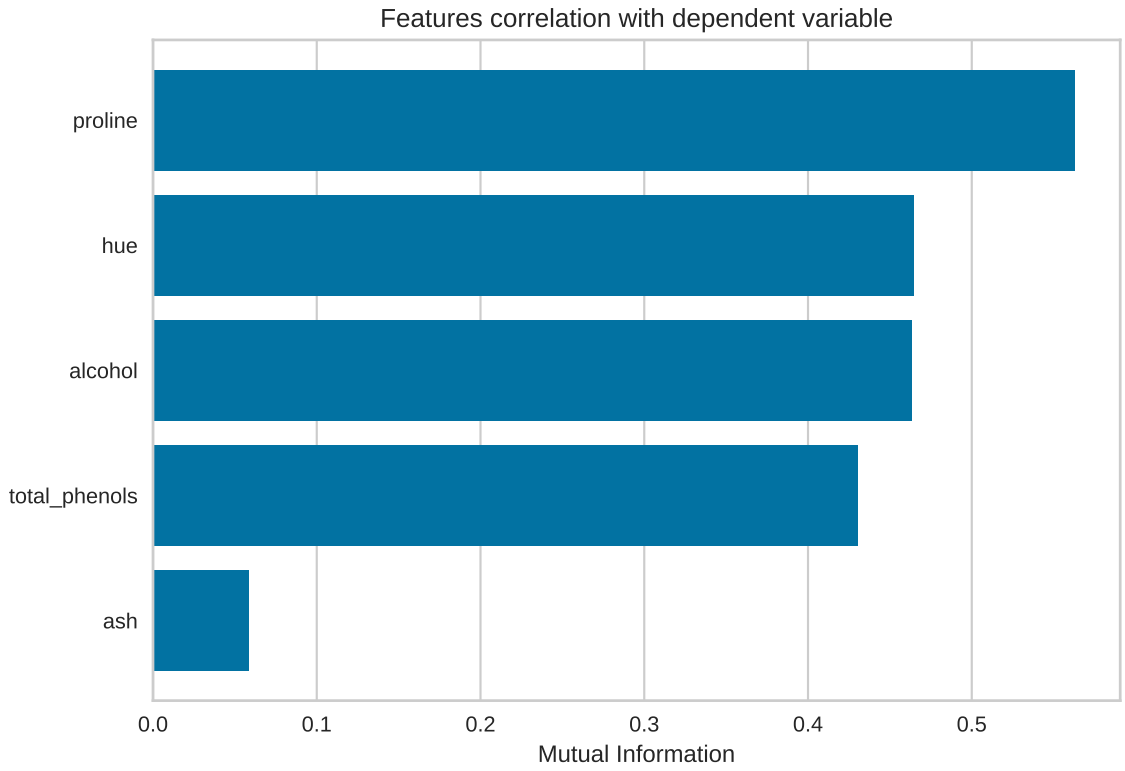
```
import pandas as pd

from sklearn import datasets
from yellowbrick.target import FeatureCorrelation

# Load the regression dataset
data = datasets.load_wine()
```

(continues on next page)



(continued from previous page)

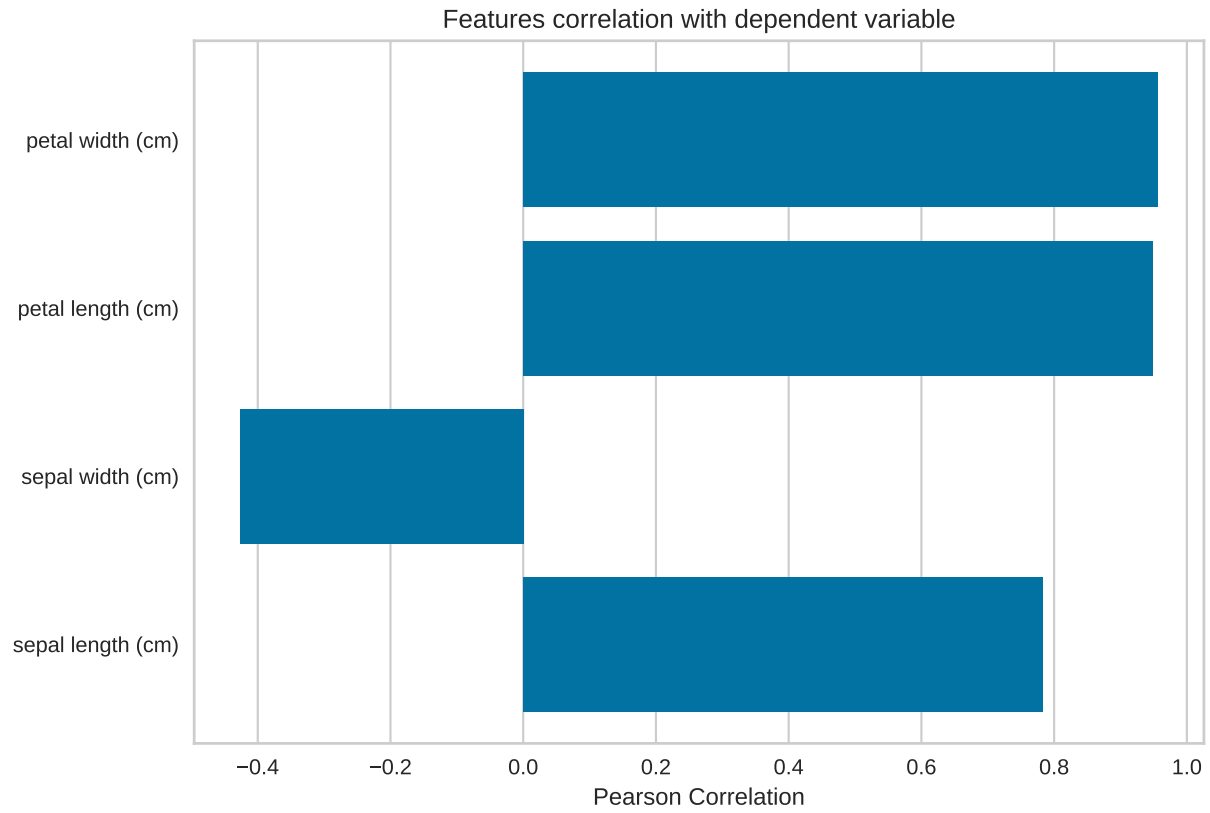
```
X, y = data['data'], data['target']
X_pd = pd.DataFrame(X, columns=data['feature_names'])

# Create a list of the features to plot
features = ['alcohol', 'ash', 'hue', 'proline', 'total_phenols']

# Instaniate the visualizer
visualizer = FeatureCorrelation(
    method='mutual_info-classification', feature_names=features, sort=True
)

visualizer.fit(X_pd, y)        # Fit the data to the visualizer
visualizer.show()              # Finalize and render the figure
```



Quick Method

The same functionality above can be achieved with the associated quick method `feature_correlation`. This method will build the `FeatureCorrelation` object with the associated arguments, fit it, then (optionally) immediately show it

```
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
from yellowbrick.target.feature_correlation import feature_correlation

#Load the diabetes dataset
data = datasets.load_iris()
X, y = data['data'], data['target']

features = np.array(data['feature_names'])
visualizer = feature_correlation(X, y, labels=features)
plt.tight_layout()
```



API Reference

Feature Correlation to Dependent Variable Visualizer.

class yellowbrick.target.feature_correlation.**FeatureCorrelation**(*ax=None*, *method='pearson'*, *labels=None*, *sort=False*, *feature_index=None*, *feature_names=None*, *color=None*, ***kwargs*)

Bases: TargetVisualizer

Displays the correlation between features and dependent variables.

This visualizer can be used side-by-side with yellowbrick.features.JointPlotVisualizer that plots a feature against the target and shows the distribution of each via a histogram on each axis.

Parameters

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

method
[str, default: 'pearson'] The method to calculate correlation between features and target. Options include:

- 'pearson', which uses scipy.stats.pearsonr

- 'mutual_info-regression', which uses `mutual_info-regression` from `sklearn.feature_selection`
- 'mutual_info-classification', which uses `mutual_info_classif` from `sklearn.feature_selection`

labels
[list, default: None] A list of feature names to use. If a DataFrame is passed to fit and features is None, feature names are selected as the column names.

sort
[boolean, default: False] If false, the features are are not sorted in the plot; otherwise features are sorted in ascending order of correlation.

feature_index
[list,] A list of feature index to include in the plot.

feature_names
[list of feature names] A list of feature names to include in the plot. Must have labels or the fitted data is a DataFrame with column names. If feature_index is provided, feature_names will be ignored.

color: string
Specify color for barchart

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> viz = FeatureCorrelation()
>>> viz.fit(X, y)
>>> viz.show()
```

Attributes

features_
[np.array] The feature labels

scores_
[np.array] Correlation between features and dependent variable.

`draw()`

Draws the feature correlation to dependent variable, called from fit.

`finalize()`

Finalize the drawing setting labels and title.

`fit`(*X*, *y*, ***kwargs*)

Fits the estimator to calculate feature correlation to dependent variable.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

kwargs
[dict] Keyword arguments passed to the fit method of the estimator.

Returns

self
[visualizer] The fit method must always return self to support pipelines.

yellowbrick.target.feature_correlation.**feature_correlation**(*X*, *y*, *ax=None*, *method='pearson'*, *labels=None*, *sort=False*, *feature_index=None*, *feature_names=None*, *color=None*, *show=True*, ***kwargs*)

Displays the correlation between features and dependent variables.

This visualizer can be used side-by-side with yellowbrick.features.JointPlotVisualizer that plots a feature against the target and shows the distribution of each via a histogram on each axis.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

method
[str, default: 'pearson'] The method to calculate correlation between features and target. Options include:

- 'pearson', which uses `scipy.stats.pearsonr`
- 'mutual_info-regression', which uses `mutual_info-regression` from `sklearn.feature_selection`
- 'mutual_info-classification', which uses `mutual_info_classif` from `sklearn.feature_selection`

labels
[list, default: None] A list of feature names to use. If a DataFrame is passed to fit and features is None, feature names are selected as the column names.

sort
[boolean, default: False] If false, the features are are not sorted in the plot; otherwise features are sorted in ascending order of correlation.

feature_index
[list,] A list of feature index to include in the plot.

feature_names
[list of feature names] A list of feature names to include in the plot. Must have labels or the fitted data is a DataFrame with column names. If feature_index is provided, feature_names will be ignored.

color: string
Specify color for barchart

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

visualizer
[FeatureCorrelation] Returns the fitted visualizer.

8.3.5 Regression Visualizers

Regression models attempt to predict a target in a continuous space. Regressor score visualizers display the instances in model space to better understand how the model is making predictions. We currently have implemented three regressor evaluations:

- *Residuals Plot*: plot the difference between the expected and actual values
- *Prediction Error Plot*: plot the expected vs. actual values in model space
- *Alpha Selection*: visual tuning of regularization hyperparameters

Estimator score visualizers *wrap* Scikit-Learn estimators and expose the Estimator API such that they have `fit()`, `predict()`, and `score()` methods that call the appropriate estimator methods under the hood. Score visualizers can wrap an estimator and be passed in as the final step in a `Pipeline` or `VisualPipeline`.

```
# Regression Evaluation Imports

from sklearn.linear_model import Ridge, Lasso
from sklearn.model_selection import train_test_split

from yellowbrick.regressor import PredictionError, ResidualsPlot
from yellowbrick.regressor.alphas import AlphaSelection
```

Residuals Plot

Residuals, in the context of regression models, are the difference between the observed value of the target variable (y) and the predicted value (ŷ), i.e. the error of the prediction. The residuals plot shows the difference between residuals on the vertical axis and the dependent variable on the horizontal axis, allowing you to detect regions within the target that may be susceptible to more or less error.

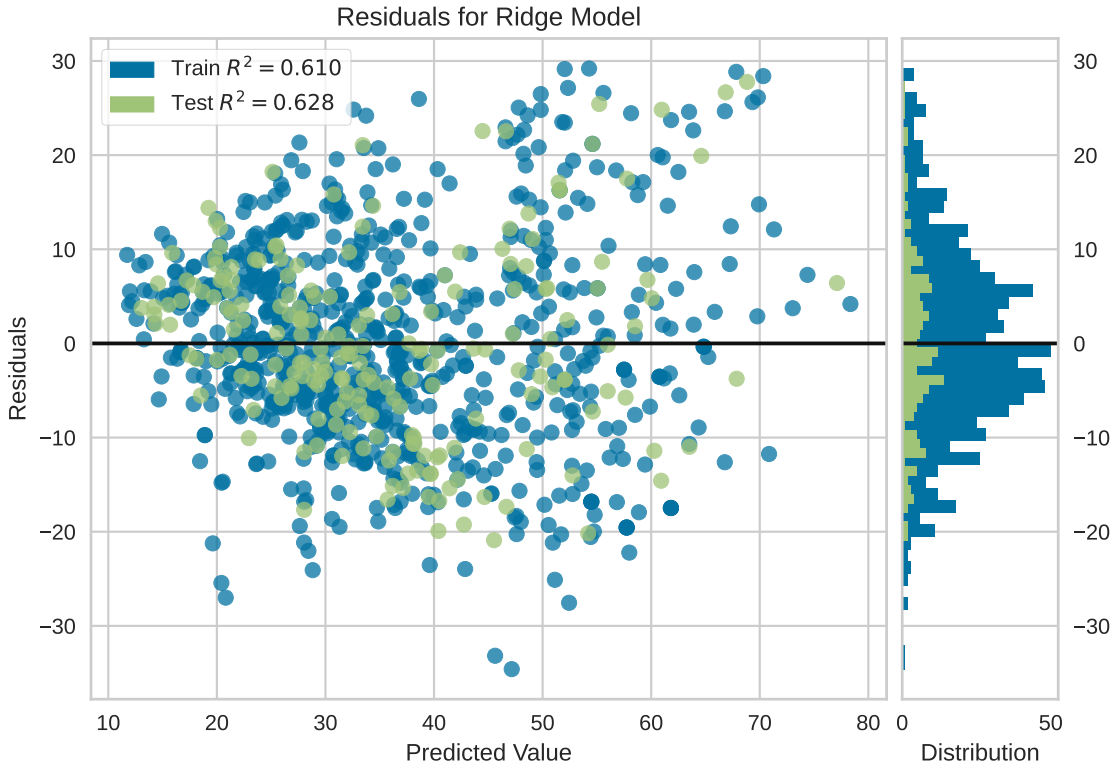
Visualizer	*ResidualsPlot*
Quick Method	*residuals_plot()*
Models	Regression
Workflow	Model evaluation

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split

from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import ResidualsPlot
```

(continues on next page)

(continued from previous page)

```
# Load a regression dataset
X, y = load_concrete()

# Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Instantiate the linear model and visualizer
model = Ridge()
visualizer = ResidualsPlot(model)

visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
visualizer.show()                 # Finalize and render the figure
```



A common use of the residuals plot is to analyze the variance of the error of the regressor. If the points are randomly dispersed around the horizontal axis, a linear regression model is usually appropriate for the data; otherwise, a non-linear model is more appropriate. In the case above, we see a fairly random, uniform distribution of the residuals against the target in two dimensions. This seems to indicate that our linear model is performing well. We can also see from the histogram that our error is normally distributed around zero, which also generally indicates a well fitted model.

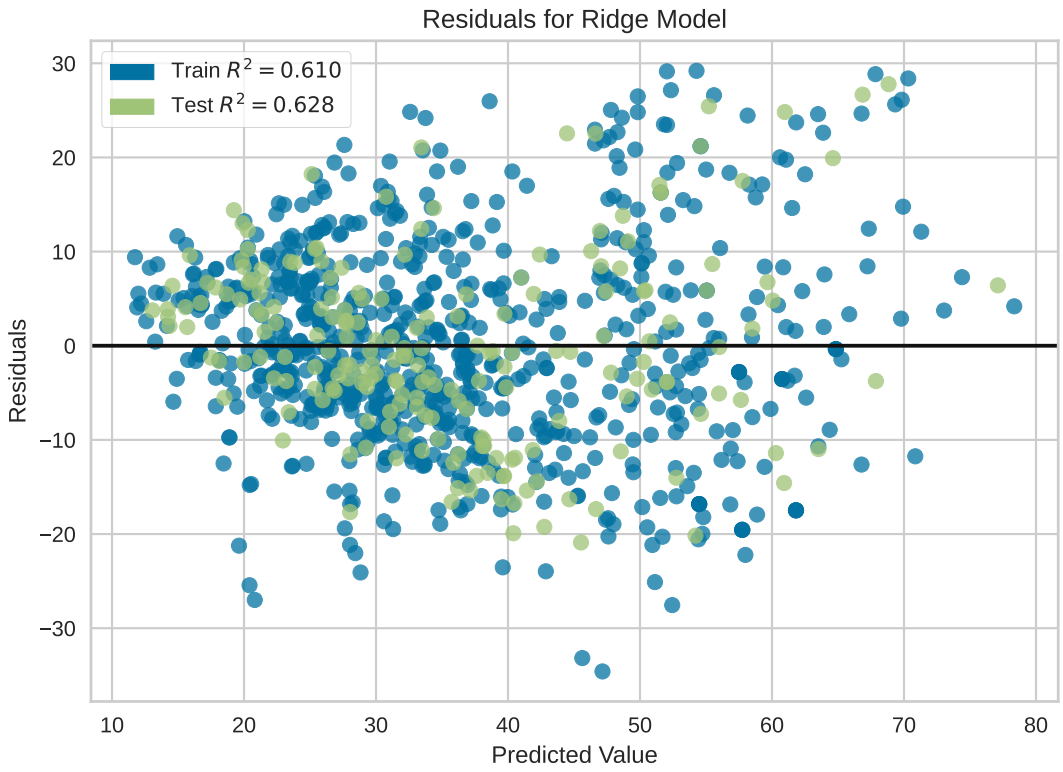
Note that if the histogram is not desired, it can be turned off with the `hist=False` flag:

```
visualizer = ResidualsPlot(model, hist=False)
```

(continues on next page)

(continued from previous page)

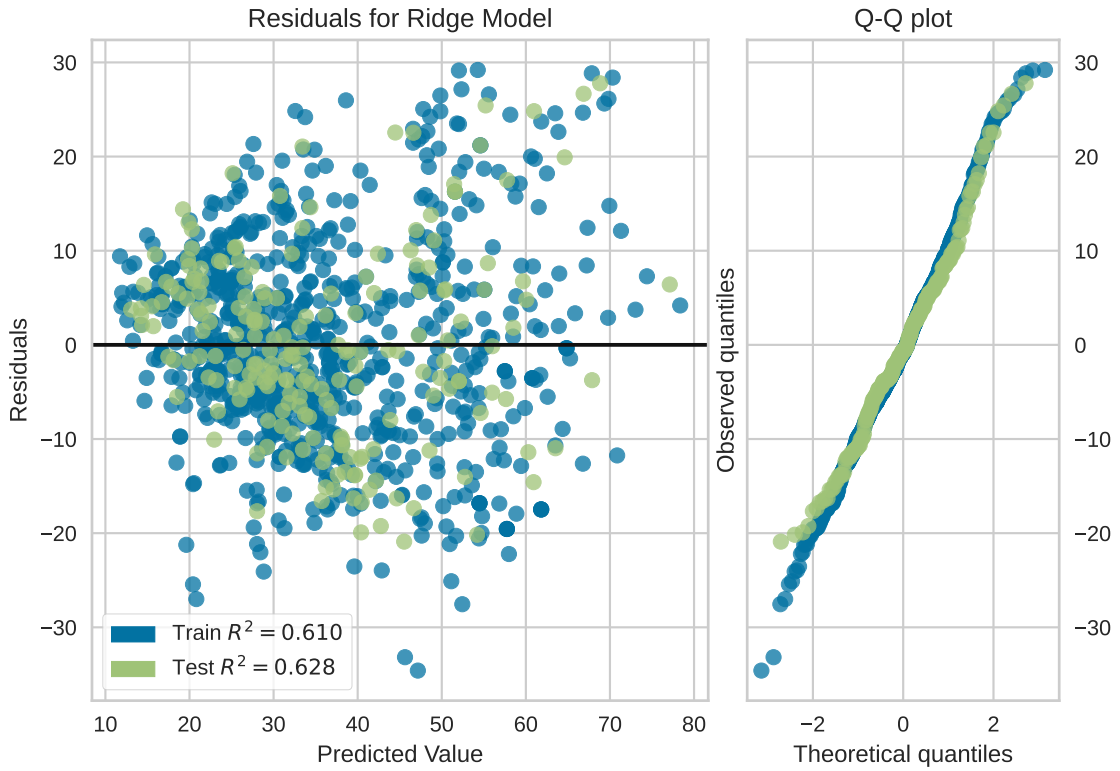
```
visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.show()
```



Warning: The histogram on the residuals plot requires matplotlib 2.0.2 or greater. If you are using an earlier version of matplotlib, simply set the `hist=False` flag so that the histogram is not drawn.

Histogram can be replaced with a Q-Q plot, which is a common way to check that residuals are normally distributed. If the residuals are normally distributed, then their quantiles when plotted against quantiles of normal distribution should form a straight line. The example below shows, how Q-Q plot can be drawn with a `qqplot=True` flag. Notice that `hist` has to be set to `False` in this case.

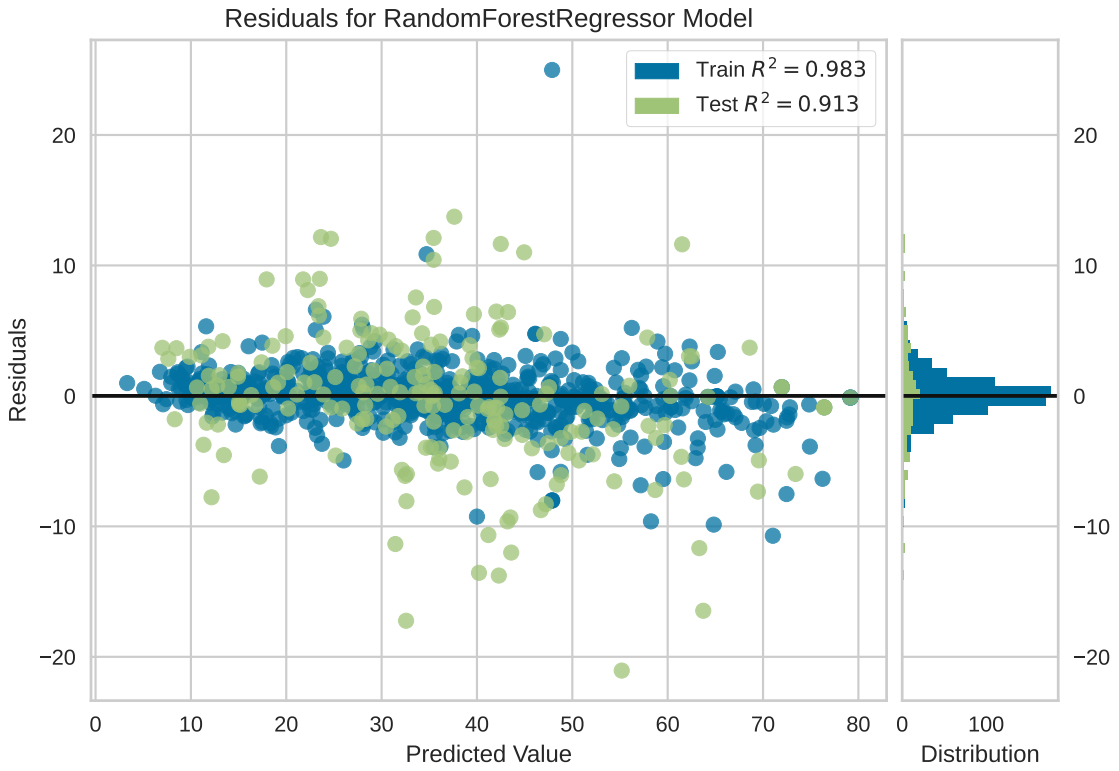
```
visualizer = ResidualsPlot(model, hist=False, qqplot=True)
visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.show()
```



Quick Method

Similar functionality as above can be achieved in one line using the associated quick method, `residuals_plot`. This method will instantiate and fit a `ResidualsPlot` visualizer on the training data, then will score it on the optionally provided test data (or the training data if it is not provided).

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split as tts

from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import residuals_plot

# Load the dataset and split into train/test splits
X, y = load_concrete()

X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2, shuffle=True)

# Create the visualizer, fit, score, and show it
viz = residuals_plot(RandomForestRegressor(), X_train, y_train, X_test, y_test)
```



API Reference

Visualize the residuals between predicted and actual data for regression problems

class yellowbrick.regressor.residuals.**ResidualsPlot**(*estimator*, *ax=None*, *hist=True*, *qqplot=False*, *train_color='b'*, *test_color='g'*, *line_color='#111111'*, *train_alpha=0.75*, *test_alpha=0.75*, *is_fitted='auto'*, ***kwargs*)

Bases: RegressionScoreVisualizer

A residual plot shows the residuals on the vertical axis and the independent variable on the horizontal axis.

If the points are randomly dispersed around the horizontal axis, a linear regression model is appropriate for the data; otherwise, a non-linear model is more appropriate.

Parameters

estimator
[a Scikit-Learn regressor] Should be an instance of a regressor, otherwise will raise a YellowbrickTypeError exception on instantiation. If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

hist
[{True, False, None, 'density', 'frequency'}, default: True] Draw a histogram showing the

distribution of the residuals on the right side of the figure. Requires Matplotlib >= 2.0.2. If set to 'density', the probability density function will be plotted. If set to True or 'frequency' then the frequency will be plotted.

qqplot
[{True, False}, default: False] Draw a Q-Q plot on the right side of the figure, comparing the quantiles of the residuals against quantiles of a standard normal distribution. Q-Q plot and histogram of residuals can not be plotted simultaneously, either *hist* or *qqplot* has to be set to False.

train_color
[color, default: 'b'] Residuals for training data are ploted with this color but also given an opacity of 0.5 to ensure that the test data residuals are more visible. Can be any matplotlib color.

test_color
[color, default: 'g'] Residuals for test data are plotted with this color. In order to create generalizable models, reserved test data residuals are of the most analytical interest, so these points are highlighted by having full opacity. Can be any matplotlib color.

line_color
[color, default: dark grey] Defines the color of the zero error line, can be any matplotlib color.

train_alpha
[float, default: 0.75] Specify a transparency for traininig data, where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

test_alpha
[float, default: 0.75] Specify a transparency for test data, where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

ResidualsPlot is a ScoreVisualizer, meaning that it wraps a model and its primary entry point is the `score()` method.

The residuals histogram feature requires matplotlib 2.0.2 or greater.

Examples

```
>>> from yellowbrick.regressor import ResidualsPlot
>>> from sklearn.linear_model import Ridge
>>> model = ResidualsPlot(Ridge())
>>> model.fit(X_train, y_train)
>>> model.score(X_test, y_test)
>>> model.show()
```

Attributes

train_score_
[float] The R^2 score that specifies the goodness of fit of the underlying regression model to the training data.

test_score_
[float] The R^2 score that specifies the goodness of fit of the underlying regression model to the test data.

`draw`(*y_pred*, *residuals*, *train=False*, ***kwargs*)

Draw the residuals against the predicted value for the specified split. It is best to draw the training split first, then the test split so that the test split (usually smaller) is above the training split; particularly if the histogram is turned on.

Parameters

y_pred
[ndarray or Series of length n] An array or series of predicted target values

residuals
[ndarray or Series of length n] An array or series of the difference between the predicted and the target values

train
[boolean, default: False] If False, *draw* assumes that the residual points being plotted are from the test data; if True, *draw* assumes the residuals are the train data.

Returns

ax
[matplotlib Axes] The axis with the plotted figure

`finalize`(***kwargs*)

Prepares the plot for rendering by adding a title, legend, and axis labels. Also draws a line at the zero residuals to show the baseline.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

fit(*X*, *y*, ***kwargs*)

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target values

kwargs: keyword arguments passed to Scikit-Learn API.

Returns

self
[ResidualsPlot] The visualizer instance

property hax

Returns the histogram axes, creating it only on demand.

property qqax

Returns the Q-Q plot axes, creating it only on demand.

score(*X*, *y=None*, *train=False*, ***kwargs*)

Generates predicted target values using the Scikit-Learn estimator.

Parameters

X
[array-like] X (also X_test) are the dependent variables of test set to predict

y
[array-like] y (also y_test) is the independent actual variables to score against

train
[boolean] If False, *score* assumes that the residual points being plotted are from the test data; if True, *score* assumes the residuals are the train data.

Returns

score
[float] The score of the underlying estimator, usually the R-squared score for regression estimators.

yellowbrick.regressor.residuals.**residuals_plot**(*estimator*, *X_train*, *y_train*, *X_test=None*, *y_test=None*, *ax=None*, *hist=True*, *qqplot=False*, *train_color='b'*, *test_color='g'*, *line_color='#111111'*, *train_alpha=0.75*, *test_alpha=0.75*, *is_fitted='auto'*, *show=True*, ***kwargs*)

ResidualsPlot quick method:

A residual plot shows the residuals on the vertical axis and the independent variable on the horizontal axis.

If the points are randomly dispersed around the horizontal axis, a linear regression model is appropriate for the data; otherwise, a non-linear model is more appropriate.

Parameters

estimator
[a Scikit-Learn regressor] Should be an instance of a regressor, otherwise will raise a YellowbrickTypeError exception on instantiation. If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X_train
[ndarray or DataFrame of shape n x m] A feature array of n instances with m features the model is trained on. Used to fit the visualizer and also to score the visualizer if test splits are not directly specified.

y_train
[ndarray or Series of length n] An array or series of target or class values. Used to fit the visualizer and also to score the visualizer if test splits are not specified.

X_test
[ndarray or DataFrame of shape n x m, default: None] An optional feature array of n instances with m features that the model is scored on if specified, using X_train as the training data.

y_test
[ndarray or Series of length n, default: None] An optional array or series of target or class values that serve as actual labels for X_test for scoring purposes.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

hist
[{True, False, None, 'density', 'frequency'}, default: True] Draw a histogram showing the distribution of the residuals on the right side of the figure. Requires Matplotlib >= 2.0.2. If set to 'density', the probability density function will be plotted. If set to True or 'frequency' then the frequency will be plotted.

qqplot
[{True, False}, default: False] Draw a Q-Q plot on the right side of the figure, comparing the quantiles of the residuals against quantiles of a standard normal distribution. Q-Q plot and histogram of residuals can not be plotted simultaneously, either *hist* or *qqplot* has to be set to False.

train_color
[color, default: 'b'] Residuals for training data are ploted with this color but also given an opacity of 0.5 to ensure that the test data residuals are more visible. Can be any matplotlib color.

test_color
[color, default: 'g'] Residuals for test data are plotted with this color. In order to create generalizable models, reserved test data residuals are of the most analytical interest, so these points are highlighted by having full opacity. Can be any matplotlib color.

line_color
[color, default: dark grey] Defines the color of the zero error line, can be any matplotlib color.

train_alpha
[float, default: 0.75] Specify a transparency for traininig data, where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

test_alpha
[float, default: 0.75] Specify a transparency for test data, where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

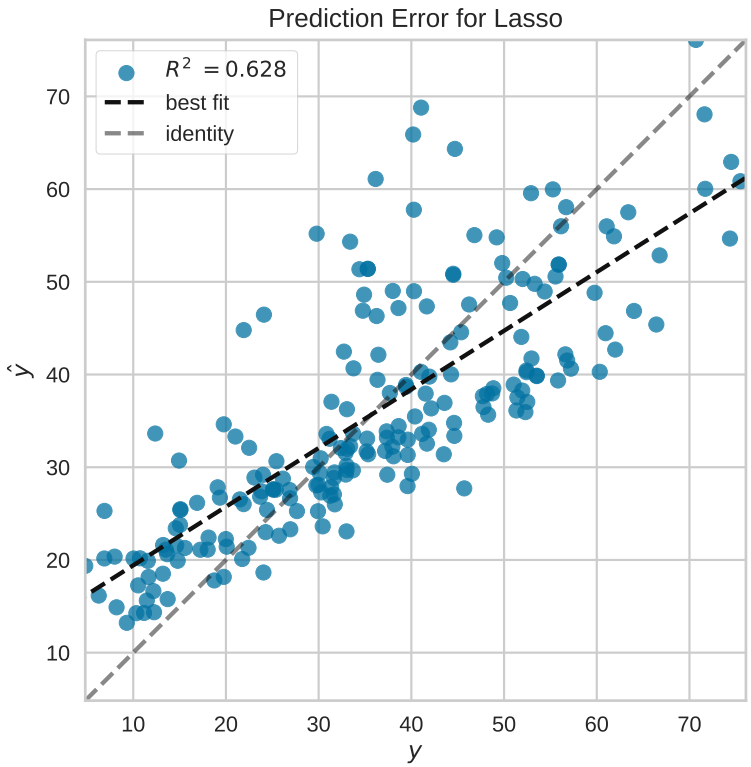
viz
[ResidualsPlot] Returns the fitted ResidualsPlot that created the figure.

Prediction Error Plot

A prediction error plot shows the actual targets from the dataset against the predicted values generated by our model. This allows us to see how much variance is in the model. Data scientists can diagnose regression models using this plot by comparing against the 45 degree line, where the prediction exactly matches the model.

Visualizer	`PredictionError`
Quick Method	`prediction_error()`
Models	Regression
Workflow	Model Evaluation

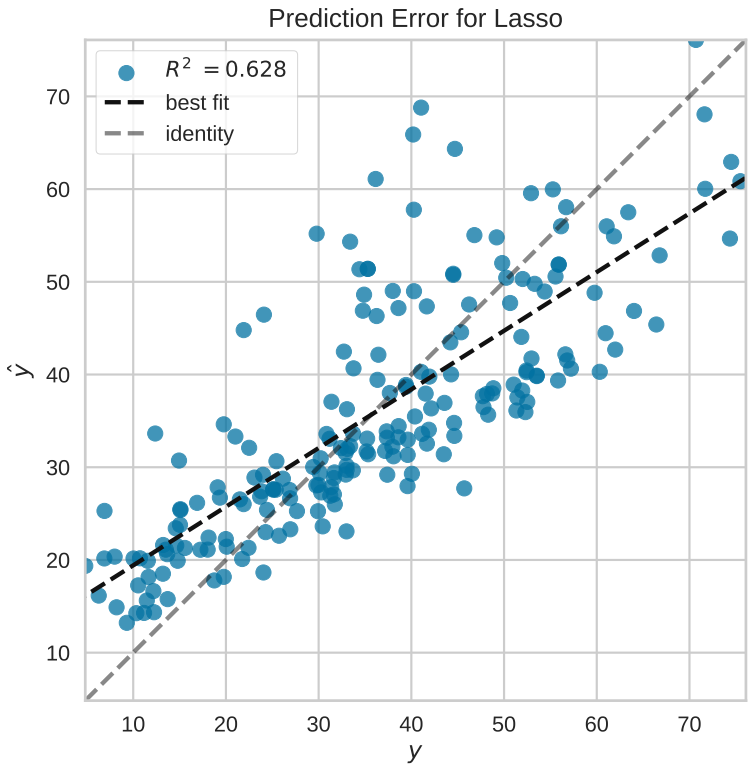
```
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split

from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import PredictionError

# Load a regression dataset
X, y = load_concrete()

# Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Instantiate the linear model and visualizer
model = Lasso()
visualizer = PredictionError(model)

visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
visualizer.show()                 # Finalize and render the figure
```



Quick Method

The same functionality can be achieved with the associated quick method `prediction_error`. This method will build the `PredictionError` object with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split

from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import prediction_error

# Load a regression dataset
X, y = load_concrete()

# Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Instantiate the linear model and visualizer
model = Lasso()
visualizer = prediction_error(model, X_train, y_train, X_test, y_test)
```



API Reference

Comparison of the predicted vs. actual values for regression problems

class yellowbrick.regressor.prediction_error.**PredictionError**(*estimator*, *ax=None*, *shared_limits=True*, *bestfit=True*, *identity=True*, *alpha=0.75*, *is_fitted='auto'*, ***kwargs*)

Bases: RegressionScoreVisualizer

The prediction error visualizer plots the actual targets from the dataset against the predicted values generated by our model(s). This visualizer is used to detect noise or heteroscedasticity along a range of the target domain.

Parameters

estimator
[a Scikit-Learn regressor] Should be an instance of a regressor, otherwise will raise a YellowbrickTypeError exception on instantiation. If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

shared_limits
[bool, default: True] If shared_limits is True, the range of the X and Y axis limits will be identical, creating a square graphic with a true 45 degree line. In this form, it is easier to diagnose under- or over- prediction, though the figure will become more sparse. To localize

points, set shared_limits to False, but note that this will distort the figure and should be accounted for during analysis.

bestfit
[bool, default: True] Draw a linear best fit line to estimate the correlation between the predicted and measured value of the target variable. The color of the bestfit line is determined by the `line_color` argument.

identity
[bool, default: True] Draw the 45 degree identity line, y=x in order to better show the relationship or pattern of the residuals. E.g. to estimate if the model is over- or under- estimating the given values. The color of the identity line is a muted version of the `line_color` argument.

alpha
[float, default: 0.75] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

PredictionError is a ScoreVisualizer, meaning that it wraps a model and its primary entry point is the *score()* method.

Examples

```
>>> from yellowbrick.regressor import PredictionError
>>> from sklearn.linear_model import Lasso
>>> model = PredictionError(Lasso())
>>> model.fit(X_train, y_train)
>>> model.score(X_test, y_test)
>>> model.show()
```

Attributes

score_
[float] The R^2 score that specifies the goodness of fit of the underlying regression model to the test data.

draw(*y*, *y_pred*)

Parameters

y
[ndarray or Series of length n] An array or series of target or class values

y_pred
[ndarray or Series of length n] An array or series of predicted target values

Returns

ax
[matplotlib Axes] The axis with the plotted figure

`finalize`(***kwargs*)

Finalizes the figure by ensuring the aspect ratio is correct and adding the identity line for comparison. Also adds a title, axis labels, and the legend.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

`score`(*X*, *y*, ***kwargs*)

The score function is the hook for visual interaction. Pass in test data and the visualizer will create predictions on the data and evaluate them with respect to the test values. The evaluation will then be passed to draw() and the result of the estimator score will be returned.

Parameters

X
[array-like] X (also X_test) are the dependent variables of test set to predict

y
[array-like] y (also y_test) is the independent actual variables to score against

Returns

score
[float]

yellowbrick.regressor.prediction_error.`prediction_error`(*estimator*, *X_train*, *y_train*, *X_test=None*, *y_test=None*, *ax=None*, *shared_limits=True*, *bestfit=True*, *identity=True*, *alpha=0.75*, *is_fitted='auto'*, *show=True*, ***kwargs*)

Quickly plot a prediction error visualizer

Plot the actual targets from the dataset against the predicted values generated by our model(s).

This helper function is a quick wrapper to utilize the PredictionError ScoreVisualizer for one-off analysis.

Parameters

estimator
[the Scikit-Learn estimator (should be a regressor)] Should be an instance of a regressor, otherwise will raise a YellowbrickTypeError exception on instantiation. If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X_train
[ndarray or DataFrame of shape n x m] A feature array of n instances with m features the model is trained on. Used to fit the visualizer and also to score the visualizer if test splits are not directly specified.

y_train
[ndarray or Series of length n] An array or series of target or class values. Used to fit the visualizer and also to score the visualizer if test splits are not specified.

X_test
[ndarray or DataFrame of shape n x m, default: None] An optional feature array of n instances with m features that the model is scored on if specified, using X_train as the training data.

y_test
[ndarray or Series of length n, default: None] An optional array or series of target or class values that serve as actual labels for X_test for scoring purposes.

ax
[matplotlib Axes] The axes to plot the figure on.

shared_limits
[bool, default: True] If shared_limits is True, the range of the X and Y axis limits will be identical, creating a square graphic with a true 45 degree line. In this form, it is easier to diagnose under- or over- prediction, though the figure will become more sparse. To localize points, set shared_limits to False, but note that this will distort the figure and should be accounted for during analysis.

bestfit
[bool, default: True] Draw a linear best fit line to estimate the correlation between the predicted and measured value of the target variable. The color of the bestfit line is determined by the `line_color` argument.

identity: bool, default: True
Draw the 45 degree identity line, y=x in order to better show the relationship or pattern of the residuals. E.g. to estimate if the model is over- or under- estimating the given values. The color of the identity line is a muted version of the `line_color` argument.

alpha
[float, default: 0.75] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

ax
[matplotlib Axes] Returns the axes that the prediction error plot was drawn on.

Alpha Selection

Regularization is designed to penalize model complexity, therefore the higher the alpha, the less complex the model, decreasing the error due to variance (overfit). Alphas that are too high on the other hand increase the error due to bias (underfit). It is important, therefore to choose an optimal alpha such that the error is minimized in both directions.

The `AlphaSelection` Visualizer demonstrates how different values of alpha influence model selection during the regularization of linear models. Generally speaking, alpha increases the affect of regularization, e.g. if alpha is zero there is no regularization and the higher the alpha, the more the regularization parameter influences the final model.

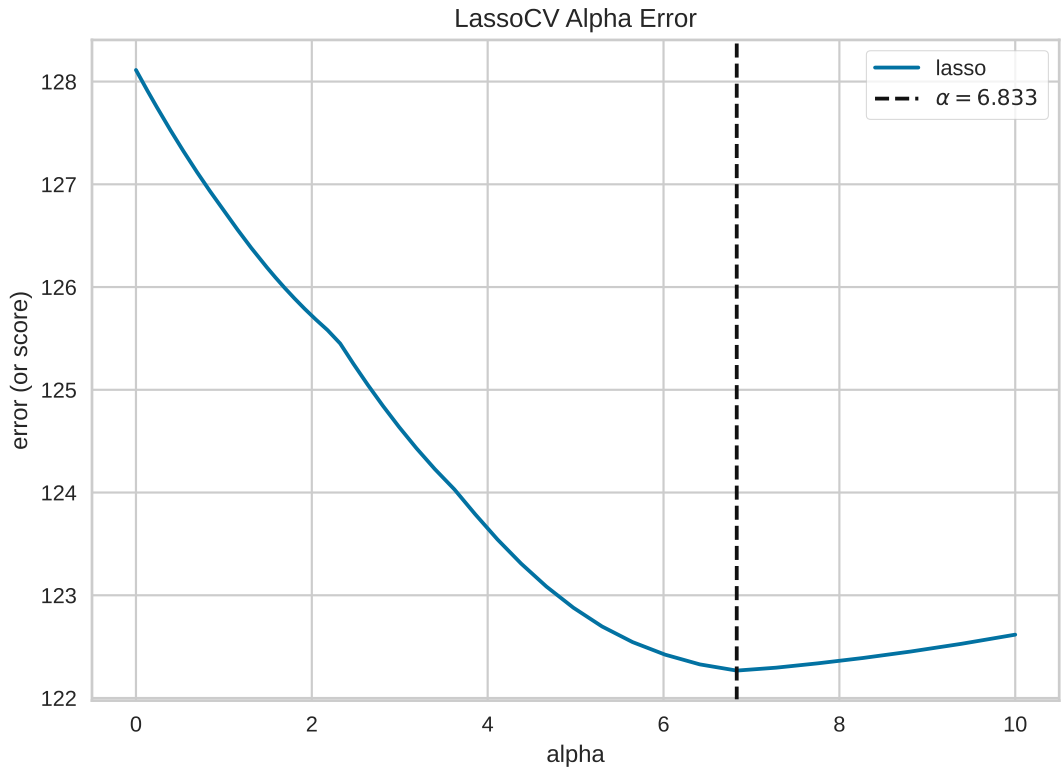
Visualizer	*AlphaSelection*
Quick Method	*alphas()*
Models	Regression
Workflow	Model selection, Hyperparameter tuning

For Estimators *with* Built-in Cross-Validation

The `AlphaSelection` visualizer wraps a "RegressionCV" model and visualizes the alpha/error curve. Use this visualization to detect if the model is responding to regularization, e.g. as you increase or decrease alpha, the model responds and error is decreased. If the visualization shows a jagged or random plot, then potentially the model is not sensitive to that type of regularization and another is required (e.g. L1 or `Lasso` regularization).

Note: The `AlphaSelection` visualizer requires a "RegressorCV" model, e.g. a specialized class that performs cross-validated alpha-selection on behalf of the model. See the `ManualAlphaSelection` visualizer if your regression model does not include cross-validation.

```
import numpy as np

from sklearn.linear_model import LassoCV
from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import AlphaSelection

# Load the regression dataset
X, y = load_concrete()

# Create a list of alphas to cross-validate against
alphas = np.logspace(-10, 1, 400)

# Instantiate the linear model and visualizer
model = LassoCV(alphas=alphas)
visualizer = AlphaSelection(model)
visualizer.fit(X, y)
visualizer.show()
```



For Estimators *without* Built-in Cross-Validation

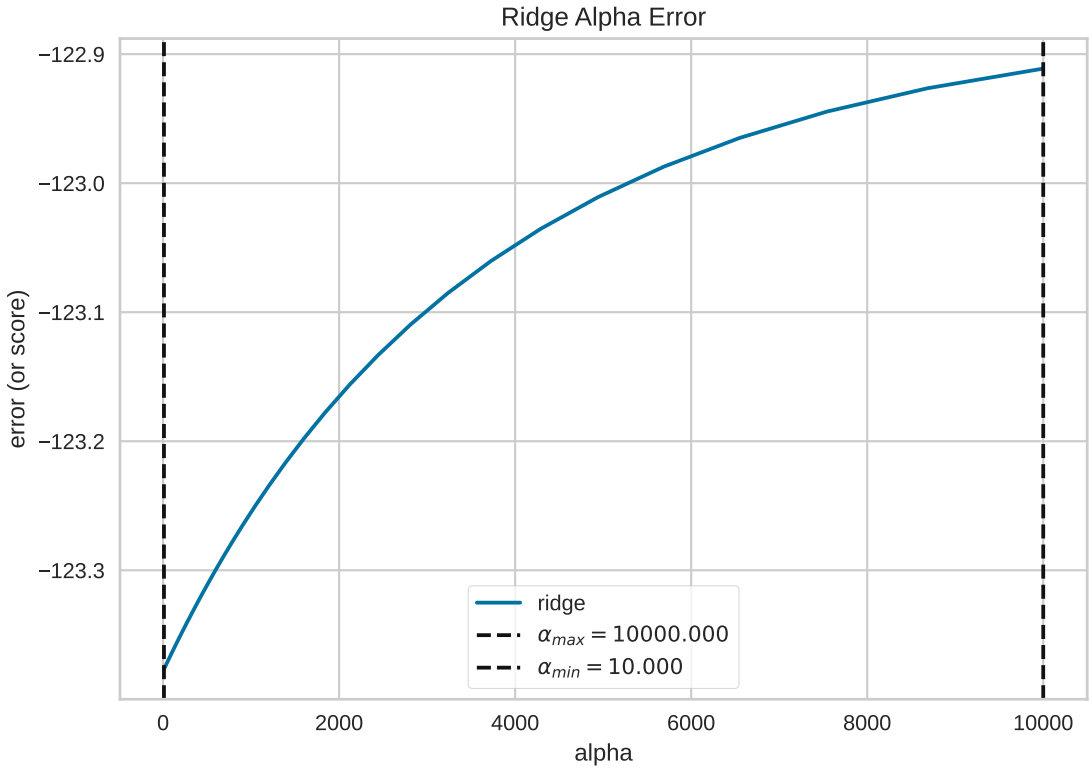
Most scikit-learn `Estimators` with `alpha` parameters have a version with built-in cross-validation. However, if the regressor you wish to use doesn't have an associated "CV" estimator, or for some reason you would like to specify more control over the alpha selection process, then you can use the `ManualAlphaSelection` visualizer. This visualizer is essentially a wrapper for scikit-learn's `cross_val_score` method, fitting a model for each alpha specified.

```
import numpy as np

from sklearn.linear_model import Ridge
from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import ManualAlphaSelection

# Load the regression dataset
X, y = load_concrete()

# Create a list of alphas to cross-validate against
alphas = np.logspace(1, 4, 50)

# Instantiate the visualizer
visualizer = ManualAlphaSelection(
    Ridge(),
    alphas=alphas,
    cv=12,
```

(continues on next page)

(continued from previous page)

```
    scoring="neg_mean_squared_error"
)

visualizer.fit(X, y)
visualizer.show()
```

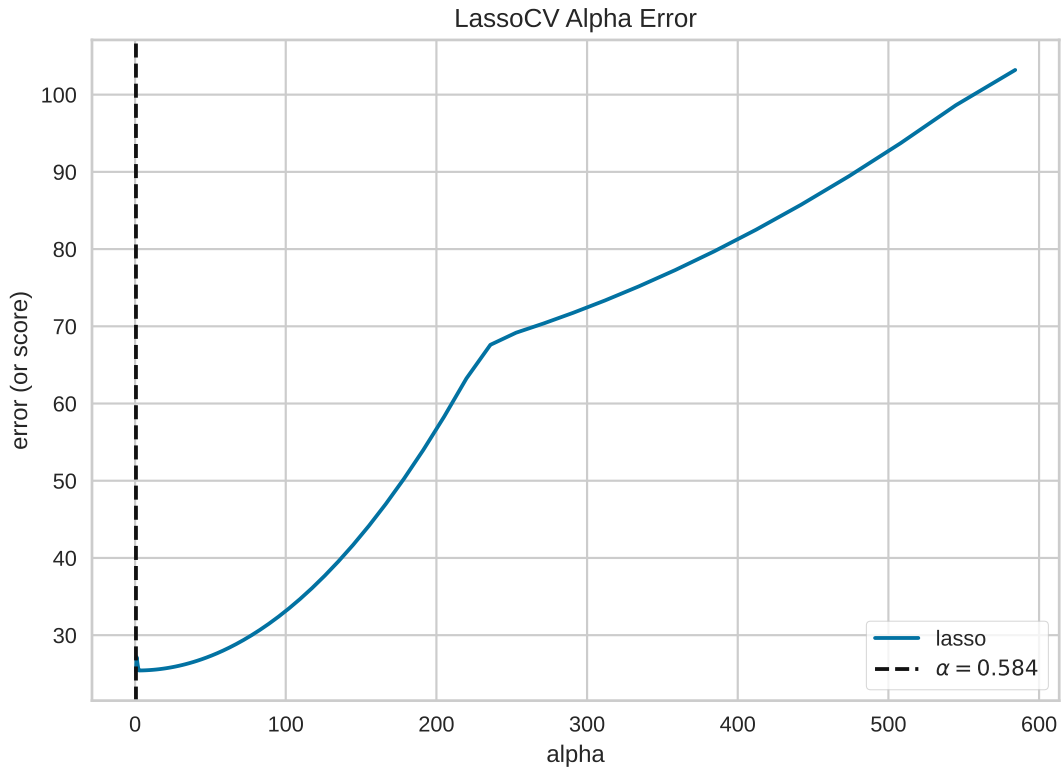


Quick Methods

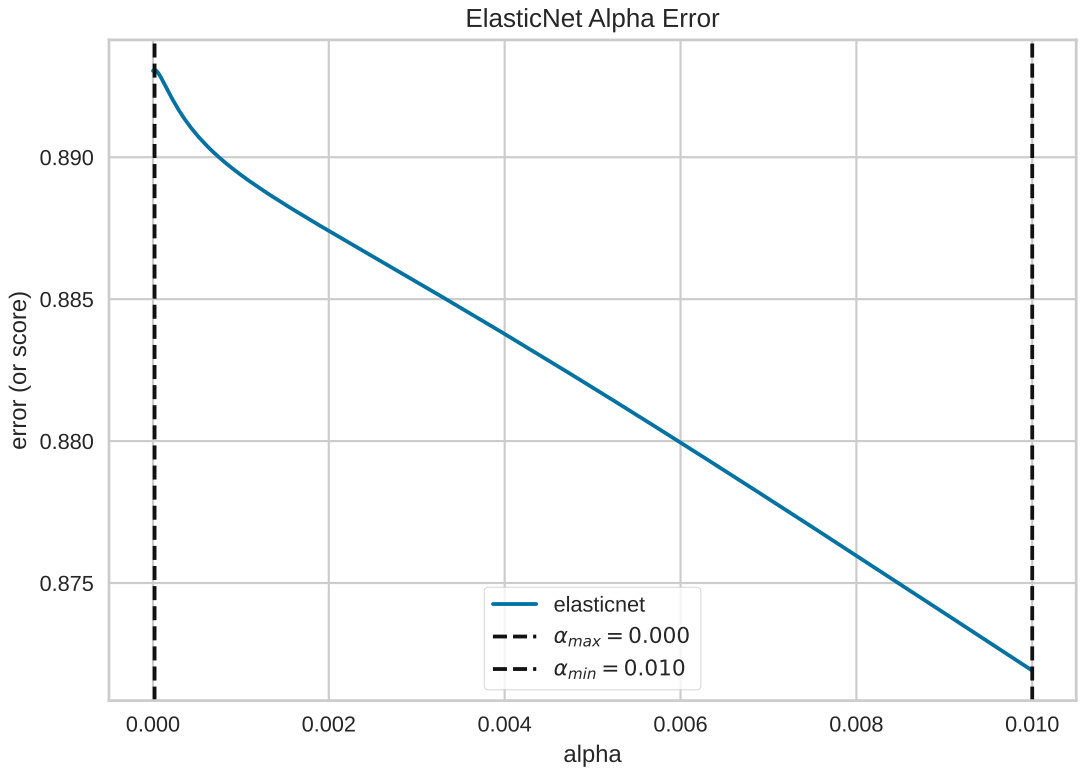
The same functionality above can be achieved with the associated quick method *alphas*. This method will build the `AlphaSelection` Visualizer object with the associated arguments, fit it, then (optionally) immediately show it.

```
from sklearn.linear_model import LassoCV
from yellowbrick.regressor.alphas import alphas

from yellowbrick.datasets import load_energy

# Load dataset
X, y = load_energy()

# Use the quick method and immediately show the figure
alphas(LassoCV(random_state=0), X, y)
```

The `ManualAlphaSelection` visualizer can also be used as a oneliner:




```
from sklearn.linear_model import ElasticNet
from yellowbrick.regressor.alphas import manual_alphas

from yellowbrick.datasets import load_energy

# Load dataset
X, y = load_energy()

# Instantiate a model
model = ElasticNet(tol=0.01, max_iter=10000)

# Use the quick method and immediately show the figure
manual_alphas(model, X, y, cv=6)
```



API Reference

Implements alpha selection visualizers for regularization

class yellowbrick.regressor.alphas.**AlphaSelection**(*estimator*, *ax=None*, *is_fitted='auto'*, ***kwargs*)

Bases: RegressionScoreVisualizer

The Alpha Selection Visualizer demonstrates how different values of alpha influence model selection during the regularization of linear models. Generally speaking, alpha increases the affect of regularization, e.g. if alpha is zero there is no regularization and the higher the alpha, the more the regularization parameter influences the final model.

Regularization is designed to penalize model complexity, therefore the higher the alpha, the less complex the model, decreasing the error due to variance (overfit). Alphas that are too high on the other hand increase the error due to bias (underfit). It is important, therefore to choose an optimal Alpha such that the error is minimized in both directions.

To do this, typically you would you use one of the "RegressionCV" models in Scikit-Learn. E.g. instead of using the Ridge (L2) regularizer, you can use RidgeCV and pass a list of alphas, which will be selected based on the cross-validation score of each alpha. This visualizer wraps a "RegressionCV" model and visualizes the alpha/error curve. Use this visualization to detect if the model is responding to regularization, e.g. as you increase or decrease alpha, the model responds and error is decreased. If the visualization shows a jagged or random plot, then potentially the model is not sensitive to that type of regularization and another is required (e.g. L1 or Lasso regularization).

Parameters

estimator

[a Scikit-Learn regressor] Should be an instance of a regressor, and specifically one whose name ends with "CV" otherwise a will raise a YellowbrickTypeError exception on instantiation. To use non-CV regressors see: `ManualAlphaSelection`. If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

This class expects an estimator whose name ends with "CV". If you wish to use some other estimator, please see the `ManualAlphaSelection` Visualizer for manually iterating through all alphas and selecting the best one.

This Visualizer hooks into the Scikit-Learn API during `fit()`. In order to pass a fitted model to the Visualizer, call the `draw()` method directly after instantiating the visualizer with the fitted model.

Note, each "RegressorCV" module has many different methods for storing alphas and error. This visualizer attempts to get them all and is known to work for RidgeCV, LassoCV, LassoLarsCV, and ElasticNetCV. If your favorite regularization method doesn't work, please submit a bug report.

For RidgeCV, make sure `store_cv_values=True`.

Examples

```
>>> from yellowbrick.regressor import AlphaSelection
>>> from sklearn.linear_model import LassoCV
>>> model = AlphaSelection(LassoCV())
>>> model.fit(X, y)
>>> model.show()
```

draw()
Draws the alpha plot based on the values on the estimator.

finalize()
Prepare the figure for rendering by setting the title as well as the X and Y axis labels and adding the legend.

fit(*X*, *y*, ***kwargs*)
A simple pass-through method; calls fit on the estimator and then draws the alpha-error plot.

class yellowbrick.regressor.alphas.**ManualAlphaSelection**(*estimator*, *ax=None*, *alphas=None*, *cv=None*, *scoring=None*, ***kwargs*)

Bases: *AlphaSelection*

The `AlphaSelection` visualizer requires a "RegressorCV", that is a specialized class that performs cross-validated alpha-selection on behalf of the model. If the regressor you wish to use doesn't have an associated "CV" estimator, or for some reason you would like to specify more control over the alpha selection process, then

you can use this manual alpha selection visualizer, which is essentially a wrapper for `cross_val_score`, fitting a model for each alpha specified.

Parameters

estimator
[an unfitted Scikit-Learn regressor] Should be an instance of an unfitted regressor, and specifically one whose name doesn't end with "CV". The regressor must support a call to `set_params(alpha=alpha)` and be fit multiple times. If the regressor name ends with "CV" a `YellowbrickValueError` is raised.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

alphas
[ndarray or Series, default: np.logspace(-10, 2, 200)] An array of alphas to fit each model with

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a *(Stratified)KFold*,
- An object to be used as a cross-validation generator.
- An iterable yielding train, test splits.

This argument is passed to the `sklearn.model_selection.cross_val_score` method to produce the cross validated score for each alpha.

scoring
[string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

This argument is passed to the `sklearn.model_selection.cross_val_score` method to produce the cross validated score for each alpha.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

This class does not take advantage of estimator-specific searching and is therefore less optimal and more time consuming than the regular "RegressorCV" estimators.

Examples

```
>>> from yellowbrick.regressor import ManualAlphaSelection
>>> from sklearn.linear_model import Ridge
>>> model = ManualAlphaSelection(
...     Ridge(), cv=12, scoring='neg_mean_squared_error'
... )
...
>>> model.fit(X, y)
>>> model.show()
```

draw()

Draws the alphas values against their associated error in a similar fashion to the AlphaSelection visualizer.

fit(*X*, *y*, ***args*)

The fit method is the primary entry point for the manual alpha selection visualizer. It sets the alpha param for each alpha in the alphas list on the wrapped estimator, then scores the model using the passed in X and y data set. Those scores are then aggregated and drawn using matplotlib.

yellowbrick.regressor.alphas.**alphas**(*estimator*, *X*, *y=None*, *ax=None*, *is_fitted='auto'*, *show=True*, ***kwargs*)

Quick Method: The Alpha Selection Visualizer demonstrates how different values of alpha influence model selection during the regularization of linear models. Generally speaking, alpha increases the affect of regularization, e.g. if alpha is zero there is no regularization and the higher the alpha, the more the regularization parameter influences the final model.

Parameters

estimator
[a Scikit-Learn regressor] Should be an instance of a regressor, and specifically one whose name ends with "CV" otherwise a will raise a YellowbrickTypeError exception on instantiation. To use non-CV regressors see: `ManualAlphaSelection`. If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features.

y
[ndarray or Series of length n] An array or series of target values.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

visualizer
[AlphaSelection] Returns the alpha selection visualizer

yellowbrick.regressor.alphas.**manual_alphas**(*estimator*, *X*, *y=None*, *ax=None*, *alphas=None*, *cv=None*, *scoring=None*, *show=True*, ***kwargs*)

Quick Method: The Manual Alpha Selection Visualizer demonstrates how different values of alpha influence model selection during the regularization of linear models. Generally speaking, alpha increases the affect of regularization, e.g. if alpha is zero there is no regularization and the higher the alpha, the more the regularization parameter influences the final model.

Parameters

estimator
[an unfitted Scikit-Learn regressor] Should be an instance of an unfitted regressor, and specifically one whose name doesn't end with "CV". The regressor must support a call to `set_params(alpha=alpha)` and be fit multiple times. If the regressor name ends with "CV" a `YellowbrickValueError` is raised.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

alphas
[ndarray or Series, default: np.logspace(-10, 2, 200)] An array of alphas to fit each model with

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a *(Stratified)KFold*,
- An object to be used as a cross-validation generator.
- An iterable yielding train, test splits.

This argument is passed to the `sklearn.model_selection.cross_val_score` method to produce the cross validated score for each alpha.

scoring
[string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

This argument is passed to the `sklearn.model_selection.cross_val_score` method to produce the cross validated score for each alpha.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

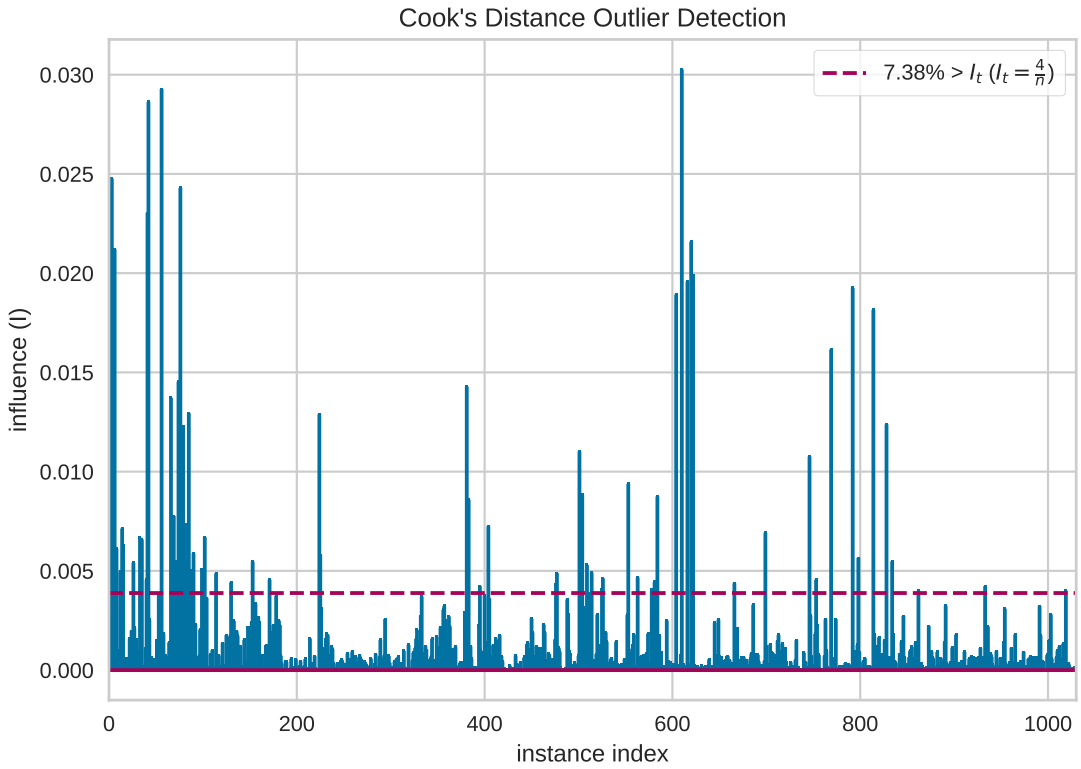
visualizer
[AlphaSelection] Returns the alpha selection visualizer

Cook's Distance

Cook's Distance is a measure of an observation or instances' influence on a linear regression. Instances with a large influence may be outliers, and datasets with a large number of highly influential points might not be suitable for linear regression without further processing such as outlier removal or imputation. The `CooksDistance` visualizer shows a stem plot of all instances by index and their associated distance score, along with a heuristic threshold to quickly show what percent of the dataset may be impacting OLS regression models.

Visualizer	*CooksDistance*
Quick Method	*cooks_distance()*
Models	General Linear Models
Workflow	Dataset/Sensitivity Analysis

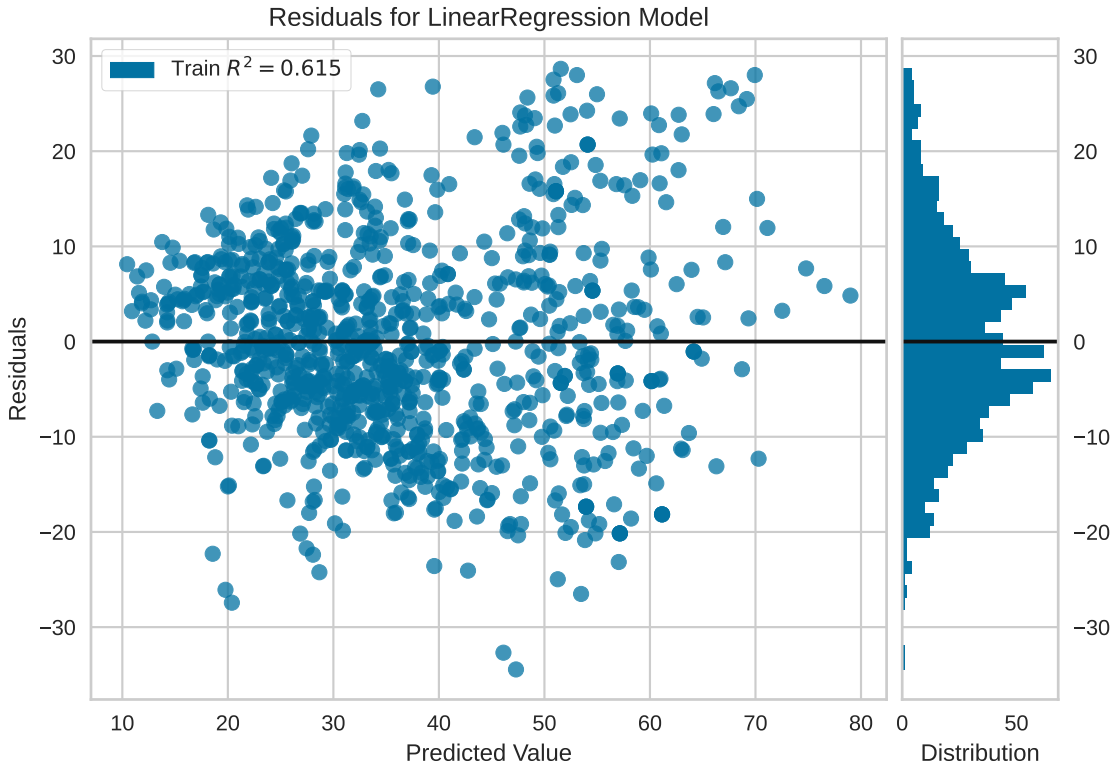
```
from yellowbrick.regressor import CooksDistance
from yellowbrick.datasets import load_concrete

# Load the regression dataset
X, y = load_concrete()

# Instantiate and fit the visualizer
visualizer = CooksDistance()
visualizer.fit(X, y)
visualizer.show()
```



The presence of so many highly influential points suggests that linear regression may not be suitable for this dataset.

One or more of the four assumptions behind linear regression might be being violated; namely one of: independence of observations, linearity of response, normality of residuals, or homogeneity of variance ("homoscedasticity"). We can check the latter three conditions using a residual plot:

```
from sklearn.linear_model import LinearRegression
from yellowbrick.regressor import ResidualsPlot

# Instantiate and fit the visualizer
model = LinearRegression()
visualizer_residuals = ResidualsPlot(model)
visualizer_residuals.fit(X, y)
visualizer_residuals.show()
```



The residuals appear to be normally distributed around 0, satisfying the linearity and normality conditions. However, they do skew slightly positive for larger predicted values, and also appear to increase in magnitude as the predicted value increases, suggesting a violation of the homoscedasticity condition.

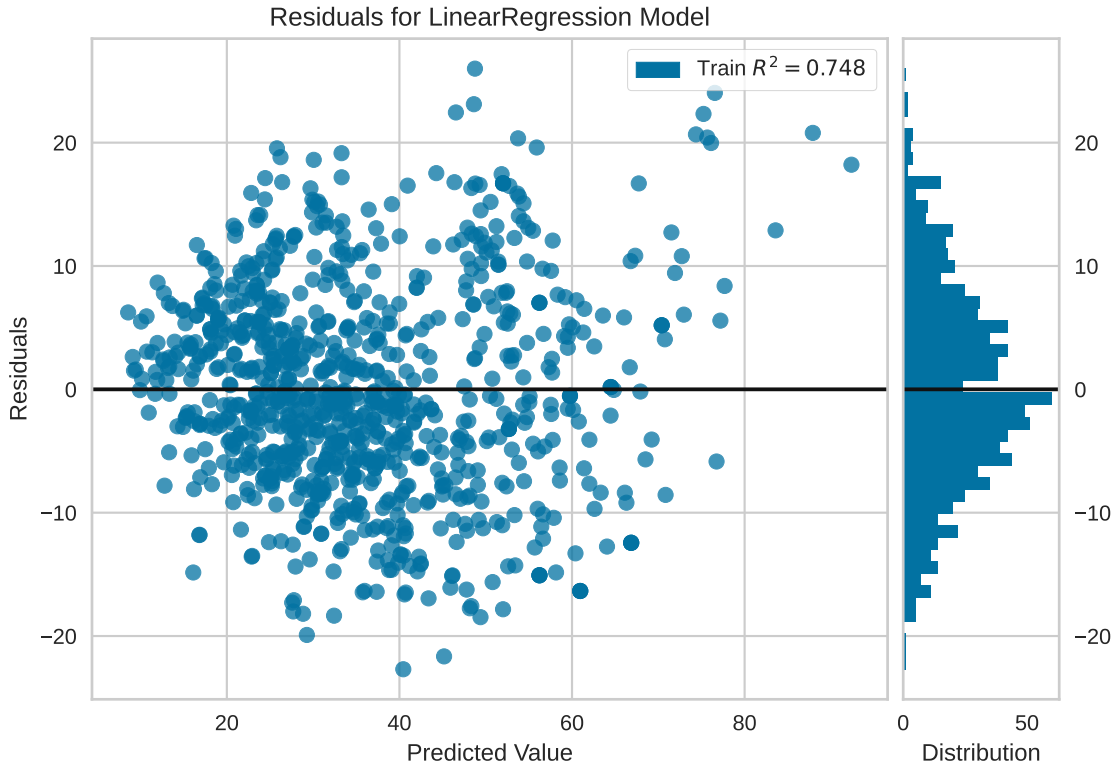
Given this information, we might consider one of the following options: (1) using a linear regression anyway, (2) using a linear regression after removing outliers, and (3) resorting to other regression models. For the sake of illustration, we will go with option (2) with the help of the Visualizer's public learned parameters `distance_` and `influence_threshold_`:

```
i_less_influential = (visualizer.distance_ <= visualizer.influence_threshold_)
X_li, y_li = X[i_less_influential], y[i_less_influential]

model = LinearRegression()
```

(continues on next page)

(continued from previous page)

```
visualizer_residuals = ResidualsPlot(model)
visualizer_residuals.fit(X_li, y_li)
visualizer_residuals.show()
```



The violations of the linear regression assumptions addressed earlier appear to be diminished. The goodness-of-fit measure has increased from 0.615 to 0.748, which is to be expected as there is less variance in the response variable after outlier removal.

Quick Method

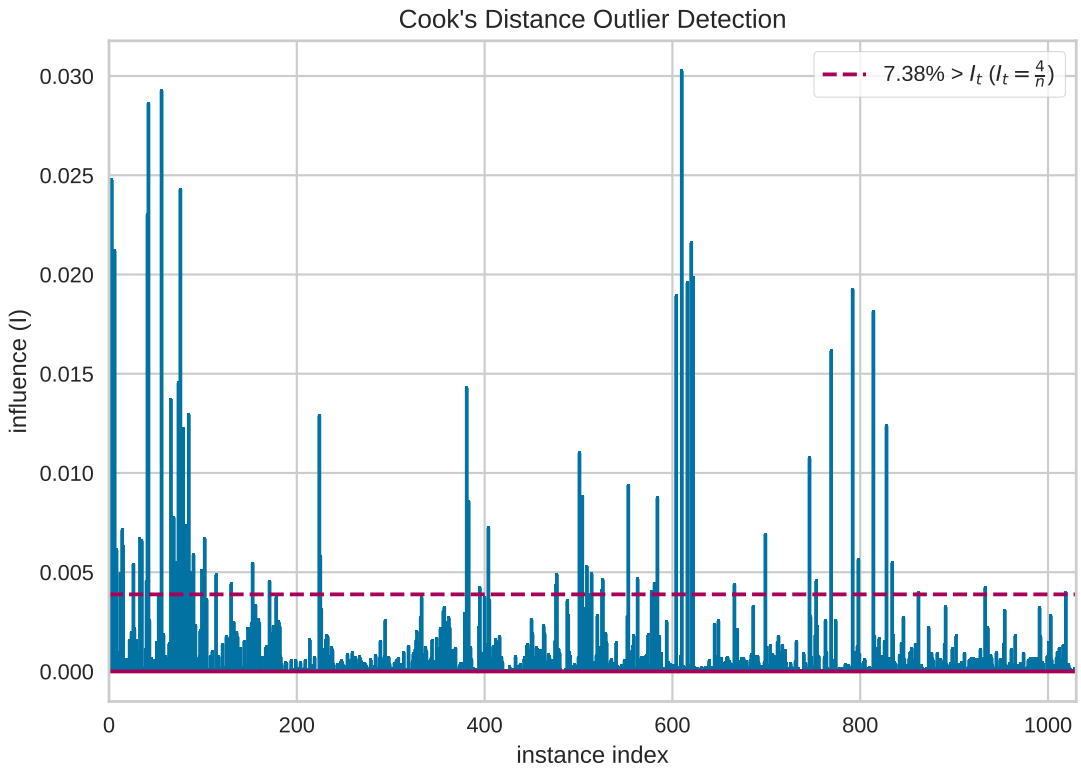
Similar functionality as above can be achieved in one line using the associated quick method, `cooks_distance`. This method will instantiate and fit a `CooksDistance` visualizer on the training data, then will score it on the optionally provided test data (or the training data if it is not provided).

```
from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import cooks_distance

# Load the regression dataset
X, y = load_concrete()

# Instantiate and fit the visualizer
cooks_distance(
    X, y,
```

(continues on next page)

(continued from previous page)

```
    draw_threshold=True,
    linefmt="C0-", markerfmt=","
)
```



API Reference

Visualize the influence and leverage of individual instances on a regression model.

class yellowbrick.regressor.influence.**CooksDistance**(*ax=None*, *draw_threshold=True*, *linefmt='C0-'*, *markerfmt=','*, ***kwargs*)

Bases: `Visualizer`

Cook's Distance is a measure of how influential an instance is to the computation of a regression, e.g. if the instance is removed would the estimated coeficients of the underlying model be substantially changed? Because of this, Cook's Distance is generally used to detect outliers in standard, OLS regression. In fact, a general rule of thumb is that D(i) > 4/n is a good threshold for determining highly influential points as outliers and this visualizer can report the percentage of data that is above that threshold.

This implementation of Cook's Distance assumes Ordinary Least Squares regression, and therefore embeds a `sklearn.linear_model.LinearRegression` under the hood. Distance is computed via the non-whitened leverage of the projection matrix, computed inside of `fit()`. The results of this visualizer are therefore similar to, but not as advanced, as a similar computation using statsmodels. Computing the influence for other regression models requires leave one out validation and can be expensive to compute.

See also:

For a longer discussion on detecting outliers in regression and computing leverage and influence, see linear regression in python, outliers/leverage detect by Huiming Song.

Parameters

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

draw_threshold
[bool, default: True] Draw a horizontal line at D(i) == 4/n to easily identify the most influential points on the final regression. This will also draw a legend that specifies the percentage of data points that are above the threshold.

linefmt
[str, default: 'C0-'] A string defining the properties of the vertical lines of the stem plot, usually this will be a color or a color and a line style. The default is simply a solid line with the first color of the color cycle.

markerfmt
[str, default: ','] A string defining the properties of the markers at the stem plot heads. The default is "pixel", e.g. basically no marker head at the top of the stem plot.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the final visualization (e.g. size or title parameters).

Notes

Cook's Distance is very similar to DFFITS, another diagnostic that is meant to show how influential a point is in a statistical regression. Although the computed values of Cook's and DFFITS are different, they are conceptually identical and there even exists a closed-form formula to convert one value to another. Because of this, we have chosen to implement Cook's distance rather than or in addition to DFFITS.

Attributes

distance_
[array, 1D] The Cook's distance value for each instance specified in `X`, e.g. an 1D array with shape `(X.shape[0],)`.

p_values_
[array, 1D] The p values associated with the F-test of Cook's distance distribution. A 1D array whose shape matches `distance_`.

influence_threshold_
[float] A rule of thumb influence threshold to determine outliers in the regression model, defined as It=4/n.

outlier_percentage_
[float] The percentage of instances whose Cook's distance is greater than the influnce threshold, the percentage is 0.0 <= p <= 100.0.

`draw()`
Draws a stem plot where each stem is the Cook's Distance of the instance at the index specified by the x axis. Optionaly draws a threshold line.

`finalize()`

Prepares the visualization for presentation and reporting.

`fit(X, y)`

Computes the leverage of X and uses the residuals of a `sklearn.linear_model.LinearRegression` to compute the Cook's Distance of each observation in X, their p-values and the number of outliers defined by the number of observations supplied.

Parameters

X
[array-like, 2D] The exogenous design matrix, e.g. training data.

y
[array-like, 1D] The endogenous response variable, e.g. target data.

Returns

self
[CooksDistance] Fit returns the visualizer instance.

`yellowbrick.regressor.influence.cooks_distance(X, y, ax=None, draw_threshold=True, linefmt='C0-', markerfmt=',', show=True, **kwargs)`

Cook's Distance is a measure of how influential an instance is to the computation of a regression, e.g. if the instance is removed would the estimated coeficients of the underlying model be substantially changed? Because of this, Cook's Distance is generally used to detect outliers in standard, OLS regression. In fact, a general rule of thumb is that D(i) > 4/n is a good threshold for determining highly influential points as outliers and this visualizer can report the percentage of data that is above that threshold.

This implementation of Cook's Distance assumes Ordinary Least Squares regression, and therefore embeds a `sklearn.linear_model.LinearRegression` under the hood. Distance is computed via the non-whitened leverage of the projection matrix, computed inside of `fit()`. The results of this visualizer are therefore similar to, but not as advanced, as a similar computation using statsmodels. Computing the influence for other regression models requires leave one out validation and can be expensive to compute.

See also:

For a longer discussion on detecting outliers in regression and computing leverage and influence, see linear regression in python, outliers/leverage detect by Huiming Song.

Parameters

X
[array-like, 2D] The exogenous design matrix, e.g. training data.

y
[array-like, 1D] The endogenous response variable, e.g. target data.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

draw_threshold
[bool, default: True] Draw a horizontal line at D(i) == 4/n to easily identify the most influential points on the final regression. This will also draw a legend that specifies the percentage of data points that are above the threshold.

linefmt
[str, default: 'C0-'] A string defining the properties of the vertical lines of the stem plot, usually this will be a color or a color and a line style. The default is simply a solid line with the first color of the color cycle.

markerfmt: str, default: ‘,’
A string defining the properties of the markers at the stem plot heads. The default is “pixel”, e.g. basically no marker head at the top of the stem plot.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the final visualization (e.g. size or title parameters).

8.3.6 Classification Visualizers

Classification models attempt to predict a target in a discrete space, that is assign an instance of dependent variables one or more categories. Classification score visualizers display the differences between classes as well as a number of classifier-specific visual evaluations. We currently have implemented the following classifier evaluations:

- *Classification Report*: A visual classification report that displays precision, recall, and F1 per-class as a heatmap.
- *Confusion Matrix*: A heatmap view of the confusion matrix of pairs of classes in multi-class classification.
- *ROCAUC*: Graphs the receiver operating characteristics and area under the curve.
- *Precision-Recall Curves*: Plots the precision and recall for different probability thresholds.
- *Class Balance*: Visual inspection of the target to show the support of each class to the final estimator.
- *Class Prediction Error*: An alternative to the confusion matrix that shows both support and the difference between actual and predicted classes.
- *Discrimination Threshold*: Shows precision, recall, f1, and queue rate over all thresholds for binary classifiers that use a discrimination probability or score.

Estimator score visualizers wrap scikit-learn estimators and expose the Estimator API such that they have `fit()`, `predict()`, and `score()` methods that call the appropriate estimator methods under the hood. Score visualizers can wrap an estimator and be passed in as the final step in a `Pipeline` or `VisualPipeline`.

```
# Classifier Evaluation Imports

from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

from yellowbrick.target import ClassBalance
from yellowbrick.classifier import ROCAUC
from yellowbrick.classifier import PrecisionRecallCurve
from yellowbrick.classifier import ClassificationReport
from yellowbrick.classifier import ClassPredictionError
from yellowbrick.classifier import DiscriminationThreshold
```

Classification Report

The classification report visualizer displays the precision, recall, F1, and support scores for the model. In order to support easier interpretation and problem detection, the report integrates numerical scores with a color-coded heatmap. All heatmaps are in the range `(0.0, 1.0)` to facilitate easy comparison of classification models across different classification reports.

Visualizer	*ClassificationReport*
Quick Method	*classification_report()*
Models	Classification
Workflow	Model evaluation

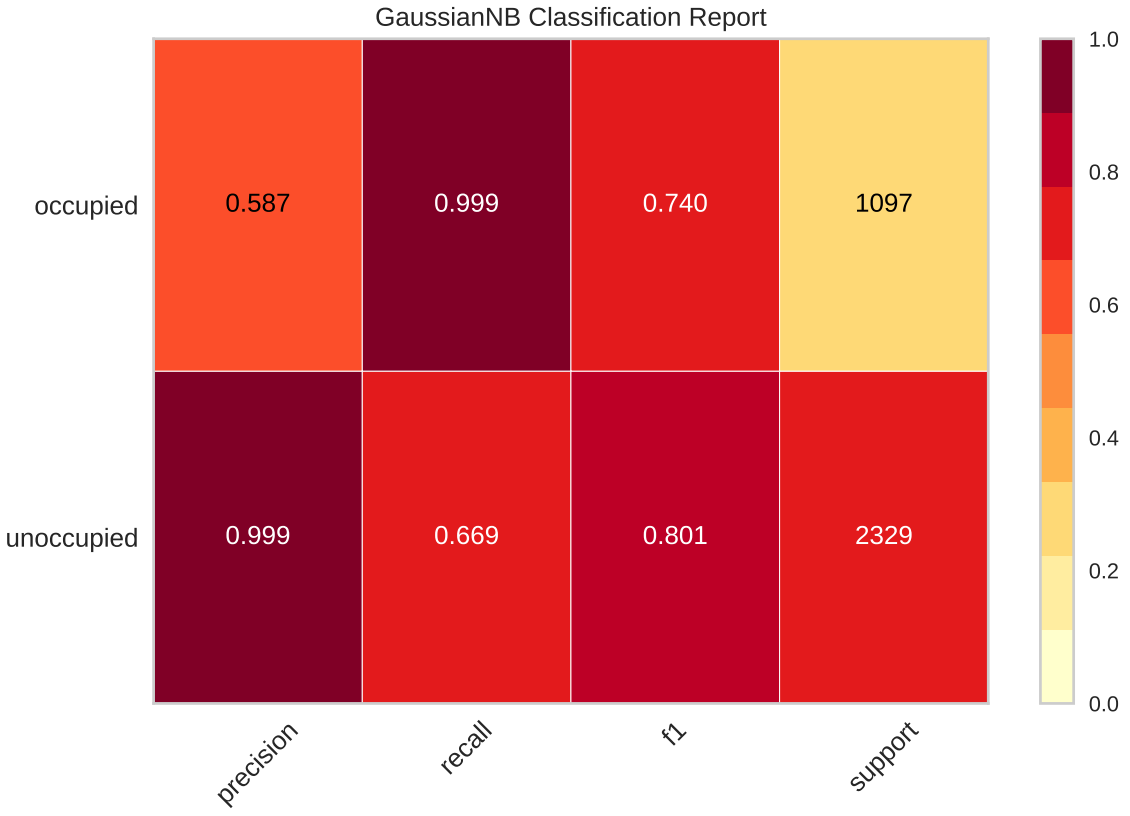
```
from sklearn.model_selection import TimeSeriesSplit
from sklearn.naive_bayes import GaussianNB

from yellowbrick.classifier import ClassificationReport
from yellowbrick.datasets import load_occupancy

# Load the classification dataset
X, y = load_occupancy()

# Specify the target classes
classes = ["unoccupied", "occupied"]

# Create the training and test data
tscv = TimeSeriesSplit()
for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

# Instantiate the classification model and visualizer
model = GaussianNB()
visualizer = ClassificationReport(model, classes=classes, support=True)

visualizer.fit(X_train, y_train)        # Fit the visualizer and the model
visualizer.score(X_test, y_test)        # Evaluate the model on the test data
visualizer.show()                       # Finalize and show the figure
```

The classification report shows a representation of the main classification metrics on a per-class basis. This gives a deeper intuition of the classifier behavior over global accuracy which can mask functional weaknesses in one class of a multiclass problem. Visual classification reports are used to compare classification models to select models that are "redder", e.g. have stronger classification metrics or that are more balanced.

The metrics are defined in terms of true and false positives, and true and false negatives. Positive and negative in this case are generic names for the classes of a binary classification problem. In the example above, we would consider true and false occupied and true and false unoccupied. Therefore a true positive is when the actual class is positive as is the estimated class. A false positive is when the actual class is negative but the estimated class is positive. Using this terminology the metrics are defined as follows:

precision

Precision can be seen as a measure of a classifier's exactness. For each class, it is defined as the ratio of true positives to the sum of true and false positives. Said another way, "for all instances classified positive, what percent was correct?"



recall
: Recall is a measure of the classifier's completeness; the ability of a classifier to correctly find all positive instances. For each class, it is defined as the ratio of true positives to the sum of true positives and false negatives. Said another way, "for all instances that were actually positive, what percent was classified correctly?"

f1 score
: The F_1 score is a weighted harmonic mean of precision and recall such that the best score is 1.0 and the worst is 0.0. Generally speaking, F_1 scores are lower than accuracy measures as they embed precision and recall into their computation. As a rule of thumb, the weighted average of F_1 should be used to compare classifier models, not global accuracy.

support
: Support is the number of actual occurrences of the class in the specified dataset. Imbalanced support in the training data may indicate structural weaknesses in the reported scores of the classifier and could indicate the need for stratified sampling or rebalancing. Support doesn't change between models but instead diagnoses the evaluation process.

Note: This example uses `TimeSeriesSplit` to split the data into the training and test sets. For more information on this cross-validation method, please refer to the scikit-learn documentation.

Quick Method

The same functionality above can be achieved with the associated quick method `classification_report`. This method will build the `ClassificationReport` object with the associated arguments, fit it, then (optionally) immediately show it.

```
from sklearn.model_selection import TimeSeriesSplit
from sklearn.naive_bayes import GaussianNB

from yellowbrick.datasets import load_occupancy
from yellowbrick.classifier import classification_report

# Load the classification data set
X, y = load_occupancy()

# Specify the target classes
classes = ["unoccupied", "occupied"]

# Create the training and test data
tscv = TimeSeriesSplit()
for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

# Instantiate the visualizer
visualizer = classification_report(
    GaussianNB(), X_train, y_train, X_test, y_test, classes=classes, support=True
)
```

API Reference

Visual classification report for classifier scoring.

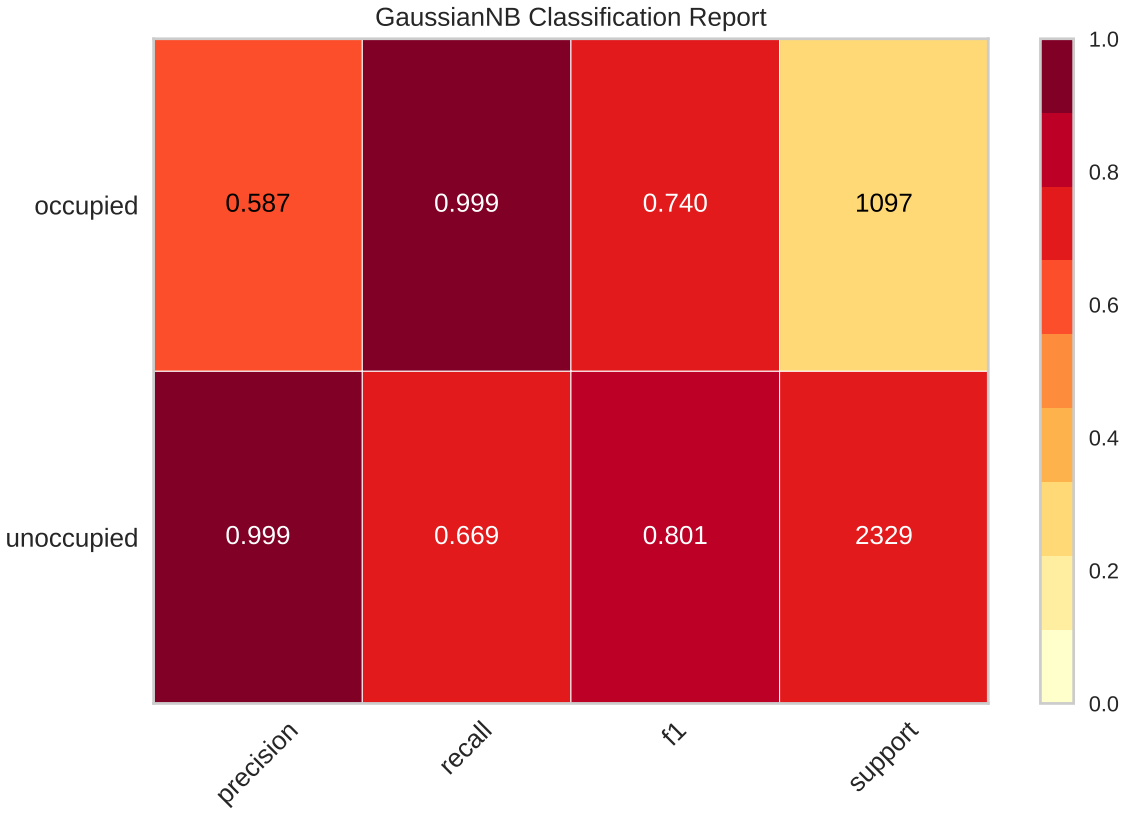
class yellowbrick.classifier.classification_report.**ClassificationReport**(*estimator*, *ax=None*, *classes=None*, *cmap='YlOrRd'*, *support=None*, *encoder=None*, *is_fitted='auto'*, *force_model=False*, *colorbar=True*, *fontsize=None*, ***kwargs*)

Bases: `ClassificationScoreVisualizer`

Classification report that shows the precision, recall, F1, and support scores for the model. Integrates numerical scores as well as a color-coded heatmap.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.



ax
: [matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

classes
: [list of str, defult: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

cmap
: [string, default: `'YlOrRd'`] Specify a colormap to define the heatmap of the predicted class against the actual class in the classification report.

support: {True, False, None, 'percent', 'count'}, default: None
: Specify if support will be displayed. It can be further defined by whether support should be reported as a raw count or percentage.

encoder
: [dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

is_fitted
: [bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified.

If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

colorbar
[bool, default: True] Specify if the color bar should be present

fontsize
[int or None, default: None] Specify the font size of the x and y labels

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Examples

```
>>> from yellowbrick.classifier import ClassificationReport
>>> from sklearn.linear_model import LogisticRegression
>>> viz = ClassificationReport(LogisticRegression())
>>> viz.fit(X_train, y_train)
>>> viz.score(X_test, y_test)
>>> viz.show()
```

Attributes

classes_
[ndarray of shape (n_classes,)] The class labels observed while fitting.

class_count_
[ndarray of shape (n_classes,)] Number of samples encountered for each class during fitting.

score_
[float] An evaluation metric of the classifier on test data produced when `score()` is called. This metric is between 0 and 1 – higher scores are generally better. For classifiers, this score is usually accuracy, but ensure you check the underlying model for more details about the score.

scores_
[dict of dicts] Outer dictionary composed of precision, recall, f1, and support scores with inner dictionaries specifiying the values for each class listed.

`draw()`

Renders the classification report across each axis.

`finalize`(***kwargs*)

Adds a title and sets the axis labels correctly. Also calls tight layout to ensure that no parts of the figure are cut off in the final visualization.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

score(*X*, *y*)

Generates the Scikit-Learn classification report.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

Returns

score_
[float] Global accuracy score

yellowbrick.classifier.classification_report.**classification_report**(*estimator*, *X_train*, *y_train*, *X_test=None*, *y_test=None*, *ax=None*, *classes=None*, *cmap='YlOrRd'*, *support=None*, *encoder=None*, *is_fitted='auto'*, *force_model=False*, *show=True*, *colorbar=True*, *fontsize=None*, ***kwargs*)

Classification Report

Displays precision, recall, F1, and support scores for the model. Integrates numerical scores as well as color-coded heatmap.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X_train
[ndarray or DataFrame of shape n x m] A feature array of n instances with m features the model is trained on. Used to fit the visualizer and also to score the visualizer if test splits are not directly specified.

y_train
[ndarray or Series of length n] An array or series of target or class values. Used to fit the visualizer and also to score the visualizer if test splits are not specified.

X_test
[ndarray or DataFrame of shape n x m, default: None] An optional feature array of n instances with m features that the model is scored on if specified, using X_train as the training data.

y_test
[ndarray or Series of length n, default: None] An optional array or series of target or class values that serve as actual labels for X_test for scoring purposes.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

classes
[list of str, defult: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

cmap
[string, default: `'YlOrRd'`] Specify a colormap to define the heatmap of the predicted class against the actual class in the classification report.

support: {True, False, None, 'percent', 'count'}, default: None
Specify if support will be displayed. It can be further defined by whether support should be reported as a raw count or percentage.

encoder
[dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

colorbar
[bool, default: True] Specify if the color bar should be present

fontsize
[int or None, default: None] Specify the font size of the x and y labels

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Returns

viz
[ClassificationReport] Returns the fitted, finalized visualizer

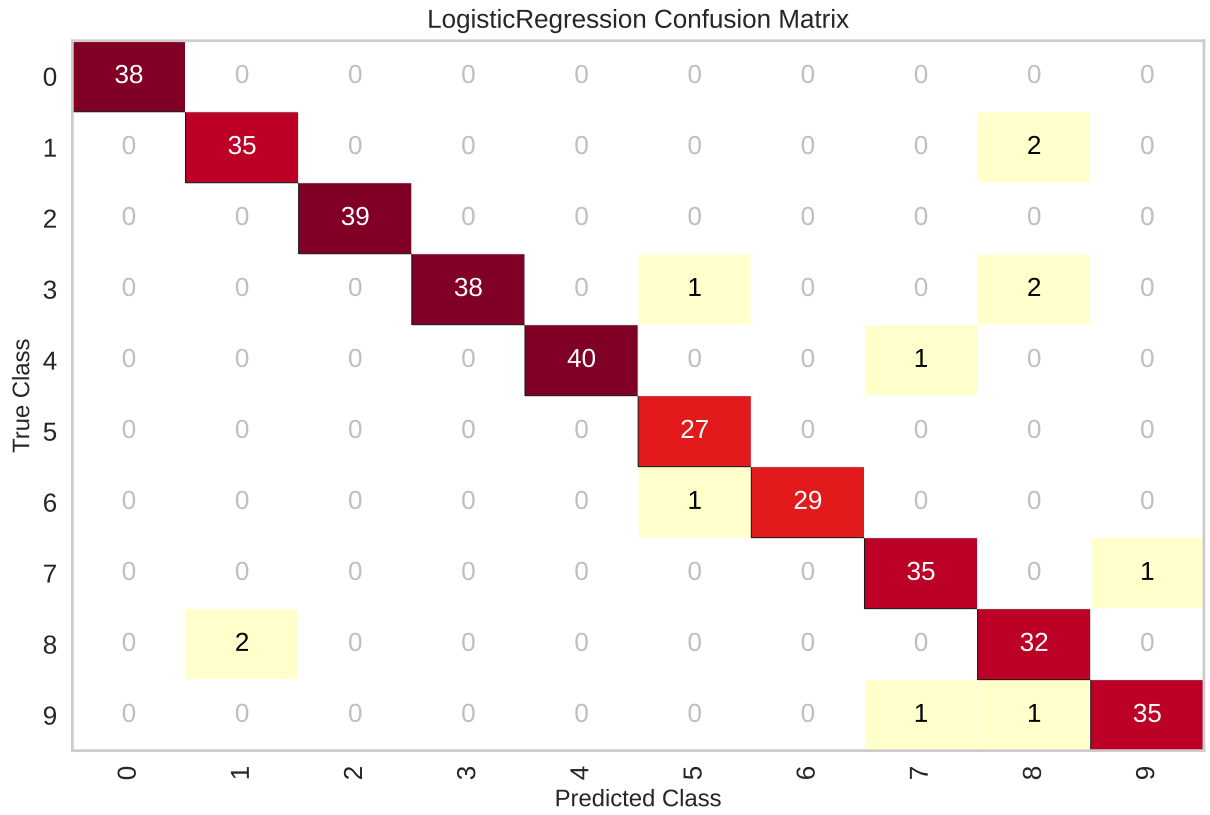
Confusion Matrix

The `ConfusionMatrix` visualizer is a `ScoreVisualizer` that takes a fitted scikit-learn classifier and a set of test `X` and `y` values and returns a report showing how each of the test values predicted classes compare to their actual classes. Data scientists use confusion matrices to understand which classes are most easily confused. These provide similar information as what is available in a `ClassificationReport`, but rather than top-level scores, they provide deeper insight into the classification of individual data points.

Below are a few examples of using the `ConfusionMatrix` visualizer; more information can be found by looking at the scikit-learn documentation on confusion matrices.

Visualizer	*ConfusionMatrix*
Quick Method	*confusion_matrix()*
Models	Classification
Workflow	Model evaluation

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split as tts
from sklearn.linear_model import LogisticRegression
from yellowbrick.classifier import ConfusionMatrix

# We'll use the handwritten digits data set from scikit-learn.
# Each feature of this dataset is an 8x8 pixel image of a handwritten number.
# Digits.data converts these 64 pixels into a single array of features
digits = load_digits()
X = digits.data
y = digits.target

X_train, X_test, y_train, y_test = tts(X, y, test_size =0.2, random_state=11)

model = LogisticRegression(multi_class="auto", solver="liblinear")

# The ConfusionMatrix visualizer taxes a model
cm = ConfusionMatrix(model, classes=[0,1,2,3,4,5,6,7,8,9])

# Fit fits the passed model. This is unnecessary if you pass the visualizer a pre-fitted␣
↪model
cm.fit(X_train, y_train)

# To create the ConfusionMatrix, we need some test data. Score runs predict() on the data
# and then creates the confusion_matrix from scikit-learn.
cm.score(X_test, y_test)

# How did we do?
cm.show()
```



Plotting with Class Names

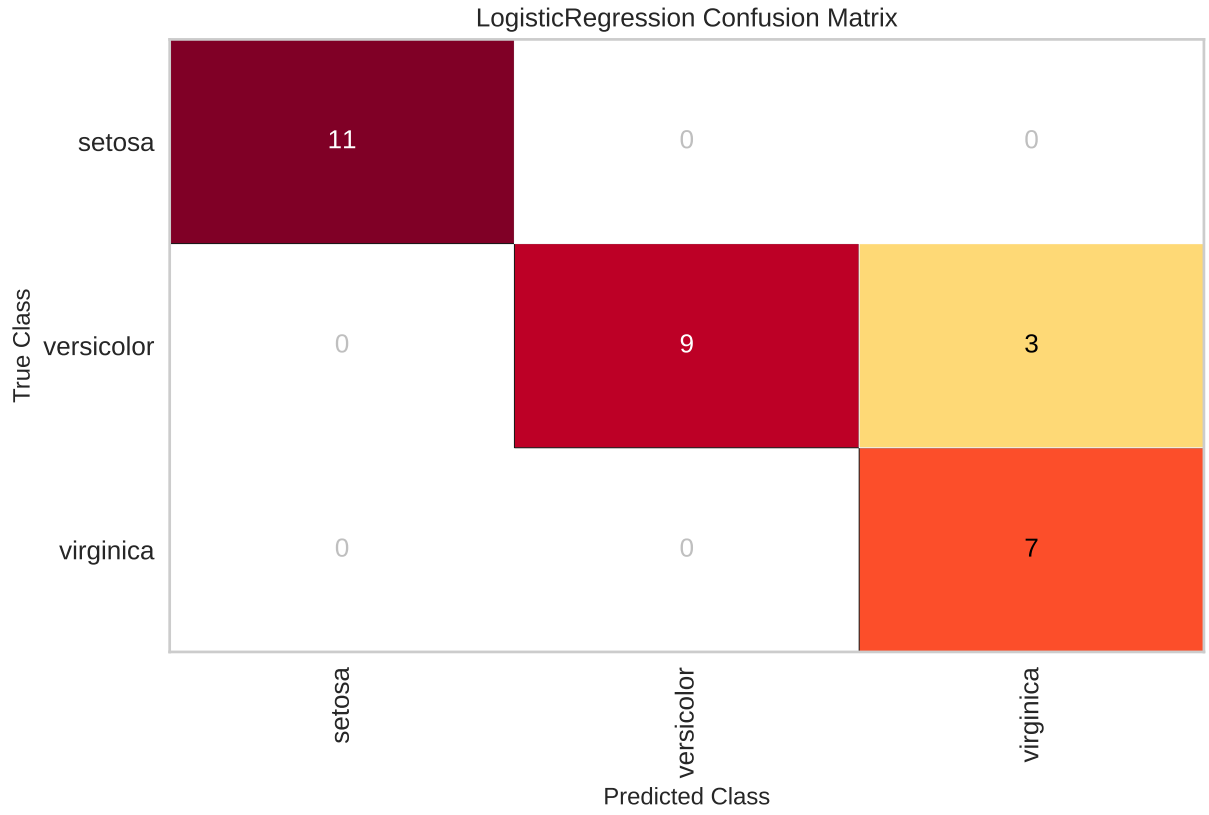
Class names can be added to a `ConfusionMatrix` plot using the `label_encoder` argument. The `label_encoder` can be a sklearn.preprocessing.LabelEncoder (or anything with an `inverse_transform` method that performs the mapping), or a `dict` with the encoding-to-string mapping as in the example below:

```
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split as tts

from yellowbrick.classifier import ConfusionMatrix

iris = load_iris()
X = iris.data
y = iris.target
classes = iris.target_names

X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2)

model = LogisticRegression(multi_class="auto", solver="liblinear")

iris_cm = ConfusionMatrix(
    model, classes=classes,
    label_encoder={0: 'setosa', 1: 'versicolor', 2: 'virginica'}
```

(continues on next page)

(continued from previous page)

```
)

iris_cm.fit(X_train, y_train)
iris_cm.score(X_test, y_test)
iris_cm.show()
```



Quick Method

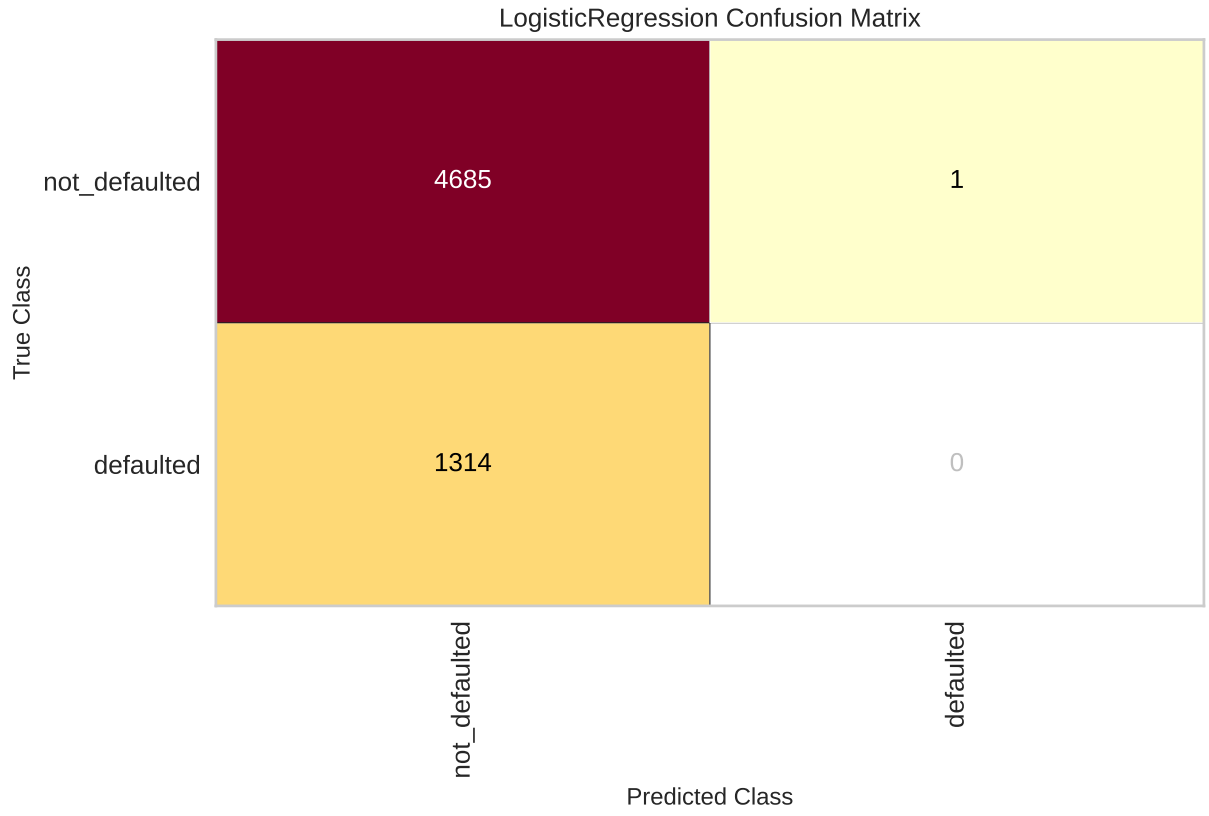
The same functionality above can be achieved with the associated quick method `confusion_matrix`. This method will build the `ConfusionMatrix` object with the associated arguments, fit it, then (optionally) immediately show it. In the below example we can see how a `LogisticRegression` struggles to effectively model the credit dataset (hint: check out *Rank2D* to examine for multicollinearity!).

```
from yellowbrick.datasets import load_credit
from yellowbrick.classifier import confusion_matrix
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split as tts

#Load the classification dataset
X, y = load_credit()

#Create the train and test data
```

(continues on next page)

(continued from previous page)

```
X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2)

# Instantiate the visualizer with the classification model
confusion_matrix(
    LogisticRegression(),
    X_train, y_train, X_test, y_test,
    classes=['not_defaulted', 'defaulted']
)
plt.tight_layout()
```



API Reference

Visual confusion matrix for classifier scoring.

class yellowbrick.classifier.confusion_matrix.**ConfusionMatrix**(*estimator*, *ax=None*, *sample_weight=None*, *percent=False*, *classes=None*, *encoder=None*, *cmap='YlOrRd'*, *fontsize=None*, *is_fitted='auto'*, *force_model=False*, ***kwargs*)

Bases: ClassificationScoreVisualizer

Creates a heatmap visualization of the sklearn.metrics.confusion_matrix(). A confusion matrix shows each combination of the true and predicted classes for a test data set.

The default color map uses a yellow/orange/red color scale. The user can choose between displaying values as the percent of true (cell value divided by sum of row) or as direct counts. If percent of true mode is selected, 100% accurate predictions are highlighted in green.

Requires a classification model.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

sample_weight: array-like of shape = [n_samples], optional
Passed to `confusion_matrix` to weight the samples.

percent: bool, default: False
Determines whether or not the confusion_matrix is displayed as counts or as a percent of true predictions. Note, if specifying a subset of classes, percent should be set to False or inaccurate figures will be displayed.

classes
[list of str, defult: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

encoder
[dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

cmap
[string, default: `'YlOrRd'`] Specify a colormap to define the heatmap of the predicted class against the actual class in the confusion matrix.

fontsize
[int, default: None] Specify the fontsize of the text in the grid and labels to make the matrix a bit easier to read. Uses rcParams font size by default.

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Examples

```
>>> from yellowbrick.classifier import ConfusionMatrix
>>> from sklearn.linear_model import LogisticRegression
>>> viz = ConfusionMatrix(LogisticRegression())
>>> viz.fit(X_train, y_train)
>>> viz.score(X_test, y_test)
>>> viz.show()
```

Attributes

classes_
[ndarray of shape (n_classes,)] The class labels observed while fitting.

class_counts_
[ndarray of shape (n_classes,)] Number of samples encountered for each class supporting the confusion matrix.

score_
[float] An evaluation metric of the classifier on test data produced when `score()` is called. This metric is between 0 and 1 – higher scores are generally better. For classifiers, this score is usually accuracy, but ensure you check the underlying model for more details about the metric.

confusion_matrix_
[array, shape = [n_classes, n_classes]] The numeric scores of the confusion matrix.

`draw()`

Renders the classification report; must be called after score.

`finalize`(***kwargs*)

Finalize executes any subclass-specific axes finalization steps.

Parameters

kwargs: dict
generic keyword arguments.

Notes

The user calls show and show calls finalize. Developers should implement visualizer-specific finalization methods like setting titles or axes labels, etc.

`score`(*X*, *y*)

Draws a confusion matrix based on the test data supplied by comparing predictions on instances X with the true values specified by the target vector y.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

Returns

score_
[float] Global accuracy score

show(*outpath=None*, ***kwargs*)

Makes the magic happen and a visualizer appear! You can pass in a path to save the figure to disk with various backends, or you can call it with no arguments to show the figure either in a notebook or in a GUI window that pops up on screen.

Parameters

outpath: string, default: None
path or None. Save figure to disk or if None show in window

clear_figure: boolean, default: False
When True, this flag clears the figure after saving to file or showing on screen. This is useful when making consecutive plots.

kwargs: dict
generic keyword arguments.

Notes

Developers of visualizers don't usually override show, as it is primarily called by the user to render the visualization.

yellowbrick.classifier.confusion_matrix.**confusion_matrix**(*estimator*, *X_train*, *y_train*, *X_test=None*, *y_test=None*, *ax=None*, *sample_weight=None*, *percent=False*, *classes=None*, *encoder=None*, *cmap='YlOrRd'*, *fontsize=None*, *is_fitted='auto'*, *force_model=False*, *show=True*, ***kwargs*)

Confusion Matrix

Creates a heatmap visualization of the sklearn.metrics.confusion_matrix(). A confusion matrix shows each combination of the true and predicted classes for a test data set.

The default color map uses a yellow/orange/red color scale. The user can choose between displaying values as the percent of true (cell value divided by sum of row) or as direct counts. If percent of true mode is selected, 100% accurate predictions are highlighted in green.

Requires a classification model.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X_train
[array-like, 2D] The table of instance data or independent variables that describe the outcome of the dependent variable, y. Used to fit the visualizer and also to score the visualizer if test splits are not specified.

y_train
[array-like, 2D] The vector of target data or the dependent variable predicted by X. Used to fit the visualizer and also to score the visualizer if test splits are not specified.

X_test: array-like, 2D, default: None
The table of instance data or independent variables that describe the outcome of the dependent variable, y. Used to score the visualizer if specified.

y_test: array-like, 1D, default: None
The vector of target data or the dependent variable predicted by X. Used to score the visualizer if specified.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

sample_weight: array-like of shape = [n_samples], optional
Passed to `confusion_matrix` to weight the samples.

percent: bool, default: False
Determines whether or not the confusion_matrix is displayed as counts or as a percent of true predictions. Note, if specifying a subset of classes, percent should be set to False or inaccurate figures will be displayed.

classes
[list of str, defult: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

encoder
[dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

cmap
[string, default: `'YlOrRd'`] Specify a colormap to define the heatmap of the predicted class against the actual class in the confusion matrix.

fontsize
[int, default: None] Specify the fontsize of the text in the grid and labels to make the matrix a bit easier to read. Uses rcParams font size by default.

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Returns

viz
[ConfusionMatrix] Returns the fitted, finalized visualizer

ROCAUC

A ROCAUC (Receiver Operating Characteristic/Area Under the Curve) plot allows the user to visualize the tradeoff between the classifier's sensitivity and specificity.

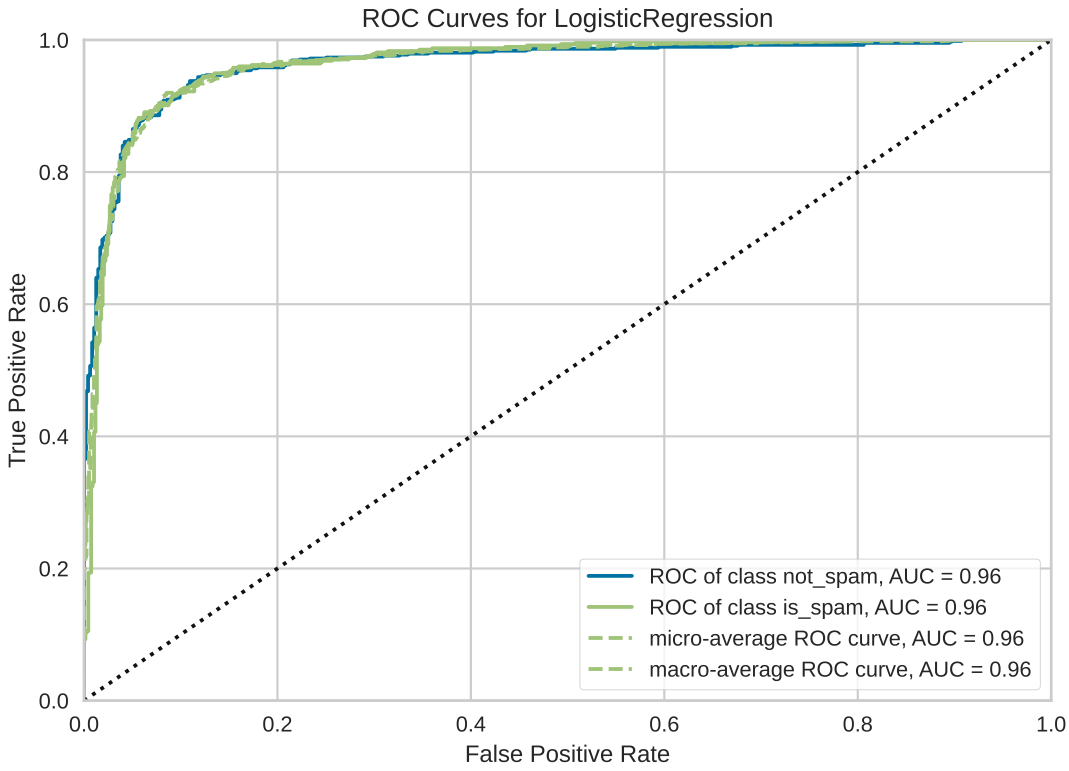
The Receiver Operating Characteristic (ROC) is a measure of a classifier's predictive quality that compares and visualizes the tradeoff between the model's sensitivity and specificity. When plotted, a ROC curve displays the true positive rate on the Y axis and the false positive rate on the X axis on both a global average and per-class basis. The ideal point is therefore the top-left corner of the plot: false positives are zero and true positives are one.

This leads to another metric, area under the curve (AUC), which is a computation of the relationship between false positives and true positives. The higher the AUC, the better the model generally is. However, it is also important to inspect the "steepness" of the curve, as this describes the maximization of the true positive rate while minimizing the false positive rate.

Visualizer	*ROCAUC*
Quick Method	*roc_auc()*
Models	Classification
Workflow	Model evaluation

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

from yellowbrick.classifier import ROCAUC
from yellowbrick.datasets import load_spam

# Load the classification dataset
X, y = load_spam()

# Create the training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# Instantiate the visualizer with the classification model
model = LogisticRegression(multi_class="auto", solver="liblinear")
visualizer = ROCAUC(model, classes=["not_spam", "is_spam"])

visualizer.fit(X_train, y_train)        # Fit the training data to the visualizer
visualizer.score(X_test, y_test)        # Evaluate the model on the test data
visualizer.show()                       # Finalize and show the figure
```

Warning: Versions of Yellowbrick =< v0.8 had a bug that triggered an `IndexError` when attempting binary classification using a Scikit-learn-style estimator with only a `decision_function`. This has been fixed as of v0.9, where the `micro`, `macro`, and `per-class` parameters of `ROCAUC` are set to `False` for such classifiers.



Multi-class ROCAUC Curves

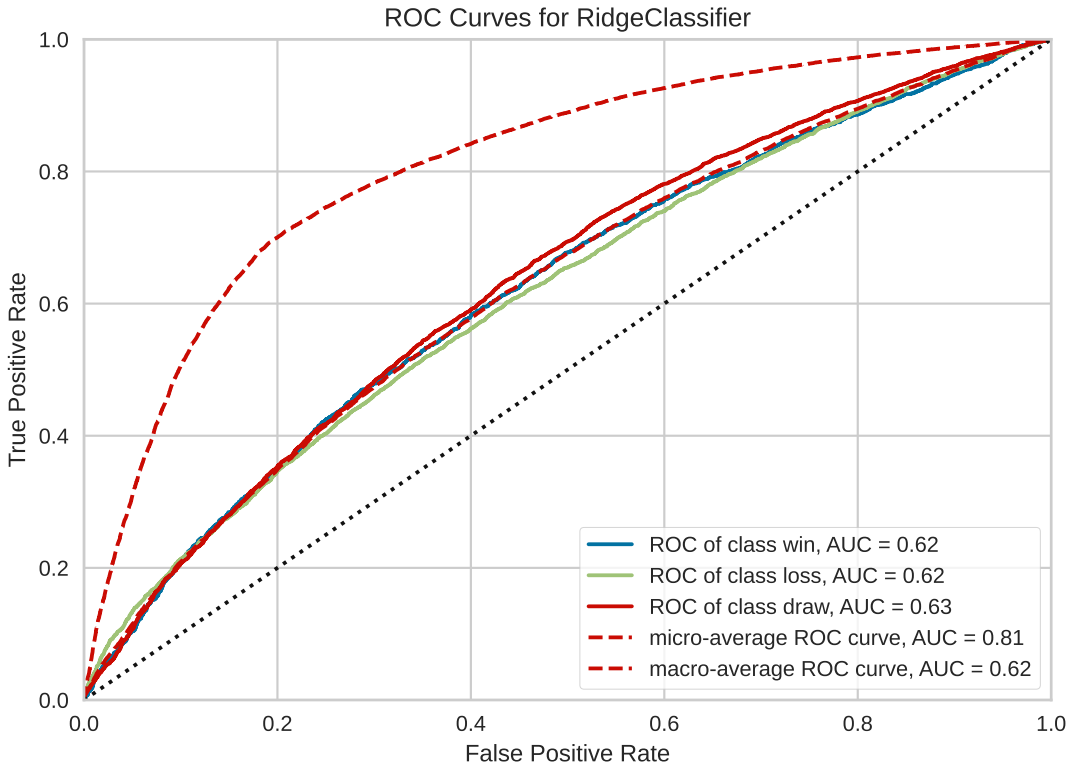
Yellowbrick's `ROCAUC` Visualizer does allow for plotting multiclass classification curves. ROC curves are typically used in binary classification, and in fact the Scikit-Learn `roc_curve` metric is only able to perform metrics for binary classifiers. Yellowbrick addresses this by binarizing the output (per-class) or to use one-vs-rest (micro score) or one-vs-all (macro score) strategies of classification.

```
from sklearn.linear_model import RidgeClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OrdinalEncoder, LabelEncoder

from yellowbrick.classifier import ROCAUC
from yellowbrick.datasets import load_game

# Load multi-class classification dataset
X, y = load_game()

# Encode the non-numeric columns
X = OrdinalEncoder().fit_transform(X)
y = LabelEncoder().fit_transform(y)

# Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

(continues on next page)

(continued from previous page)

```
# Instaniate the classification model and visualizer
model = RidgeClassifier()
visualizer = ROCAUC(model, classes=["win", "loss", "draw"])

visualizer.fit(X_train, y_train)        # Fit the training data to the visualizer
visualizer.score(X_test, y_test)        # Evaluate the model on the test data
visualizer.show()                       # Finalize and render the figure
```



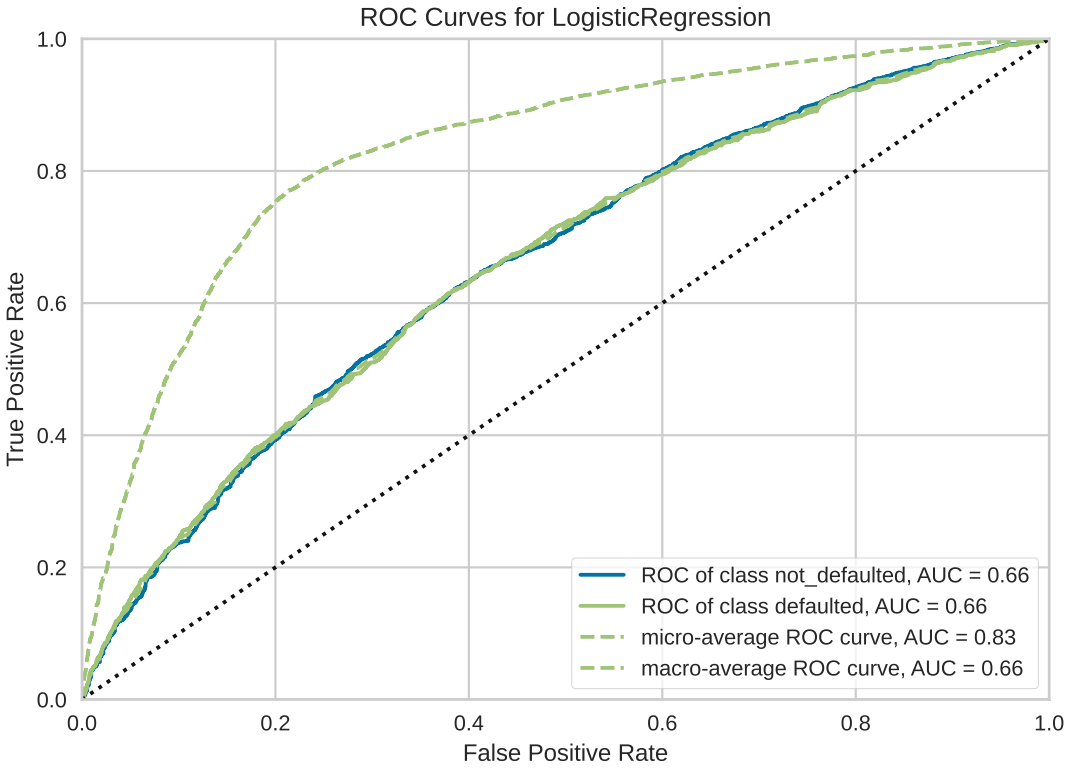
Warning: The target `y` must be numeric for this figure to work, or update to the latest version of sklearn.

By default with multi-class ROCAUC visualizations, a curve for each class is plotted, in addition to the micro- and macro-average curves for each class. This enables the user to inspect the tradeoff between sensitivity and specificity on a per-class basis. Note that for multi-class `ROCAUC`, at least one of the `micro`, `macro`, or `per_class` parameters must be set to `True` (by default, all are set to `True`).

Quick Method

The same functionality above can be achieved with the associated quick method `roc_auc`. This method will build the `ROCAUC` object with the associated arguments, fit it, then (optionally) immediately show it

```
from yellowbrick.classifier.rocauc import roc_auc
from yellowbrick.datasets import load_credit
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

#Load the classification dataset
X, y = load_credit()

#Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X,y)

# Instantiate the visualizer with the classification model
model = LogisticRegression()
roc_auc(model, X_train, y_train, X_test=X_test, y_test=y_test, classes=['not_defaulted',
↪'defaulted'])
```



API Reference

Implements visual ROC/AUC curves for classification evaluation.

class yellowbrick.classifier.rocauc.**ROCAUC**(*estimator*, *ax=None*, *micro=True*, *macro=True*, *per_class=True*, *binary=False*, *classes=None*, *encoder=None*, *is_fitted='auto'*, *force_model=False*, ***kwargs*)

Bases: `ClassificationScoreVisualizer`

Receiver Operating Characteristic (ROC) curves are a measure of a classifier's predictive quality that compares and visualizes the tradeoff between the models' sensitivity and specificity. The ROC curve displays the true positive rate on the Y axis and the false positive rate on the X axis on both a global average and per-class basis. The ideal point is therefore the top-left corner of the plot: false positives are zero and true positives are one.

This leads to another metric, area under the curve (AUC), a computation of the relationship between false positives and true positives. The higher the AUC, the better the model generally is. However, it is also important to inspect the "steepness" of the curve, as this describes the maximization of the true positive rate while minimizing the false positive rate. Generalizing "steepness" usually leads to discussions about convexity, which we do not get into here.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

micro
[bool, default: True] Plot the micro-averages ROC curve, computed from the sum of all true positives and false positives across all classes. Micro is not defined for binary classification problems with estimators with only a decision_function method.

macro
[bool, default: True] Plot the macro-averages ROC curve, which simply takes the average of curves across all classes. Macro is not defined for binary classification problems with estimators with only a decision_function method.

per_class
[bool, default: True] Plot the ROC curves for each individual class. This should be set to false if only the macro or micro average curves are required. For true binary classifiers, setting per_class=False will plot the positive class ROC curve, and per_class=True will use `1-P(1)` to compute the curve of the negative class if only a decision_function method exists on the estimator.

binary
[bool, default: False] This argument quickly resets the visualizer for true binary classification by updating the micro, macro, and per_class arguments to False (do not use in conjunction with those other arguments). Note that this is not a true hyperparameter to the visualizer, it just collects other parameters into a single, simpler argument.

classes
[list of str, defult: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some

visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

encoder
[dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Notes

ROC curves are typically used in binary classification, and in fact the Scikit-Learn `roc_curve` metric is only able to perform metrics for binary classifiers. As a result it is necessary to binarize the output or to use one-vs-rest or one-vs-all strategies of classification. The visualizer does its best to handle multiple situations, but exceptions can arise from unexpected models or outputs.

Another important point is the relationship of class labels specified on initialization to those drawn on the curves. The classes are not used to constrain ordering or filter curves; the ROC computation happens on the unique values specified in the target vector to the `score` method. To ensure the best quality visualization, do not use a LabelEncoder for this and do not pass in class labels.

See also:

http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

Examples

```
>>> from yellowbrick.classifier import ROCAUC
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.model_selection import train_test_split
>>> data = load_data("occupancy")
>>> features = ["temp", "relative humidity", "light", "C02", "humidity"]
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)
>>> oz = ROCAUC(LogisticRegression())
>>> oz.fit(X_train, y_train)
>>> oz.score(X_test, y_test)
>>> oz.show()
```

Attributes

classes_
[ndarray of shape (n_classes,)] The class labels observed while fitting.

class_count_
[ndarray of shape (n_classes,)] Number of samples encountered for each class during fitting.

score_
[float] An evaluation metric of the classifier on test data produced when `score()` is called. This metric is between 0 and 1 – higher scores are generally better. For classifiers, this score is usually accuracy, but if micro or macro is specified this returns an F1 score.

target_type_
[string] Specifies if the detected classification target was binary or multiclass.

`draw()`

Renders ROC-AUC plot. Called internally by score, possibly more than once

Returns

ax
[the axis with the plotted figure]

`finalize`(***kwargs*)

Sets a title and axis labels of the figures and ensures the axis limits are scaled between the valid ROCAUC score values.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

`fit`(*X*, *y=None*)

Fit the classification model.

`score`(*X*, *y=None*)

Generates the predicted target values using the Scikit-Learn estimator.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

Returns

score_
[float] Global accuracy unless micro or macro scores are requested.

`yellowbrick.classifier.rocauc.roc_auc`(*estimator*, *X_train*, *y_train*, *X_test=None*, *y_test=None*, *ax=None*, *micro=True*, *macro=True*, *per_class=True*, *binary=False*, *classes=None*, *encoder=None*, *is_fitted='auto'*, *force_model=False*, *show=True*, ***kwargs*)

ROCAUC

Receiver Operating Characteristic (ROC) curves are a measure of a classifier's predictive quality that compares and visualizes the tradeoff between the models' sensitivity and specificity. The ROC curve displays the true positive rate on the Y axis and the false positive rate on the X axis on both a global average and per-class basis. The ideal point is therefore the top-left corner of the plot: false positives are zero and true positives are one.

This leads to another metric, area under the curve (AUC), a computation of the relationship between false positives and true positives. The higher the AUC, the better the model generally is. However, it is also important to inspect the "steepness" of the curve, as this describes the maximization of the true positive rate while minimizing the false positive rate. Generalizing "steepness" usually leads to discussions about convexity, which we do not get into here.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X_train
[array-like, 2D] The table of instance data or independent variables that describe the outcome of the dependent variable, y. Used to fit the visualizer and also to score the visualizer if test splits are not specified.

y_train
[array-like, 2D] The vector of target data or the dependent variable predicted by X. Used to fit the visualizer and also to score the visualizer if test splits not specified.

X_test: array-like, 2D, default: None
The table of instance data or independent variables that describe the outcome of the dependent variable, y. Used to score the visualizer if specified.

y_test: array-like, 1D, default: None
The vector of target data or the dependent variable predicted by X. Used to score the visualizer if specified.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

test_size
[float, default=0.2] The percentage of the data to reserve as test data.

random_state
[int or None, default=None] The value to seed the random number generator for shuffling data.

micro
[bool, default: True] Plot the micro-averages ROC curve, computed from the sum of all true positives and false positives across all classes. Micro is not defined for binary classification problems with estimators with only a decision_function method.

macro
[bool, default: True] Plot the macro-averages ROC curve, which simply takes the average of curves across all classes. Macro is not defined for binary classification problems with estimators with only a decision_function method.

per_class
[bool, default: True] Plot the ROC curves for each individual class. This should be set to false if only the macro or micro average curves are required. For true binary classifiers, setting per_class=False will plot the positive class ROC curve, and per_class=True will use `1-P(1)` to compute the curve of the negative class if only a decision_function method exists on the estimator.

binary
[bool, default: False] This argument quickly resets the visualizer for true binary classification

by updating the micro, macro, and per_class arguments to False (do not use in conjunction with those other arguments). Note that this is not a true hyperparameter to the visualizer, it just collects other parameters into a single, simpler argument.

classes
[list of str, defult: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

encoder
[dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Returns

viz
[ROCAUC] Returns the fitted, finalized visualizer object

Notes

ROC curves are typically used in binary classification, and in fact the Scikit-Learn `roc_curve` metric is only able to perform metrics for binary classifiers. As a result it is necessary to binarize the output or to use one-vs-rest or one-vs-all strategies of classification. The visualizer does its best to handle multiple situations, but exceptions can arise from unexpected models or outputs.

Another important point is the relationship of class labels specified on initialization to those drawn on the curves. The classes are not used to constrain ordering or filter curves; the ROC computation happens on the unique values specified in the target vector to the `score` method. To ensure the best quality visualization, do not use a LabelEncoder for this and do not pass in class labels.

See also:

https://bit.ly/2IORWO2

Examples

```
>>> from yellowbrick.classifier import ROCAUC
>>> from sklearn.linear_model import LogisticRegression
>>> data = load_data("occupancy")
>>> features = ["temp", "relative humidity", "light", "C02", "humidity"]
>>> X = data[features].values
>>> y = data.occupancy.values
>>> roc_auc(LogisticRegression(), X, y)
```

Precision-Recall Curves

The `PrecisionRecallCurve` shows the tradeoff between a classifier's precision, a measure of result relevancy, and recall, a measure of completeness. For each class, precision is defined as the ratio of true positives to the sum of true and false positives, and recall is the ratio of true positives to the sum of true positives and false negatives.

Visualizer	*PrecisionRecallCurve*
Quick Method	*precision_recall_curve()*
Models	Classification
Workflow	Model evaluation

precision
: Precision can be seen as a measure of a classifier's exactness. For each class, it is defined as the ratio of true positives to the sum of true and false positives. Said another way, "for all instances classified positive, what percent was correct?"

recall
: Recall is a measure of the classifier's completeness; the ability of a classifier to correctly find all positive instances. For each class, it is defined as the ratio of true positives to the sum of true positives and false negatives. Said another way, "for all instances that were actually positive, what percent was classified correctly?"

average precision
: Average precision expresses the precision-recall curve in a single number, which represents the area under the curve. It is computed as the weighted average of precision achieved at each threshold, where the weights are the differences in recall from the previous thresholds.

Both precision and recall vary between 0 and 1, and in our efforts to select and tune machine learning models, our goal is often to try to maximize both precision and recall, i.e. a model that returns accurate results for the majority of classes it selects. This would result in a `PrecisionRecallCurve` visualization with a high area under the curve.

Binary Classification

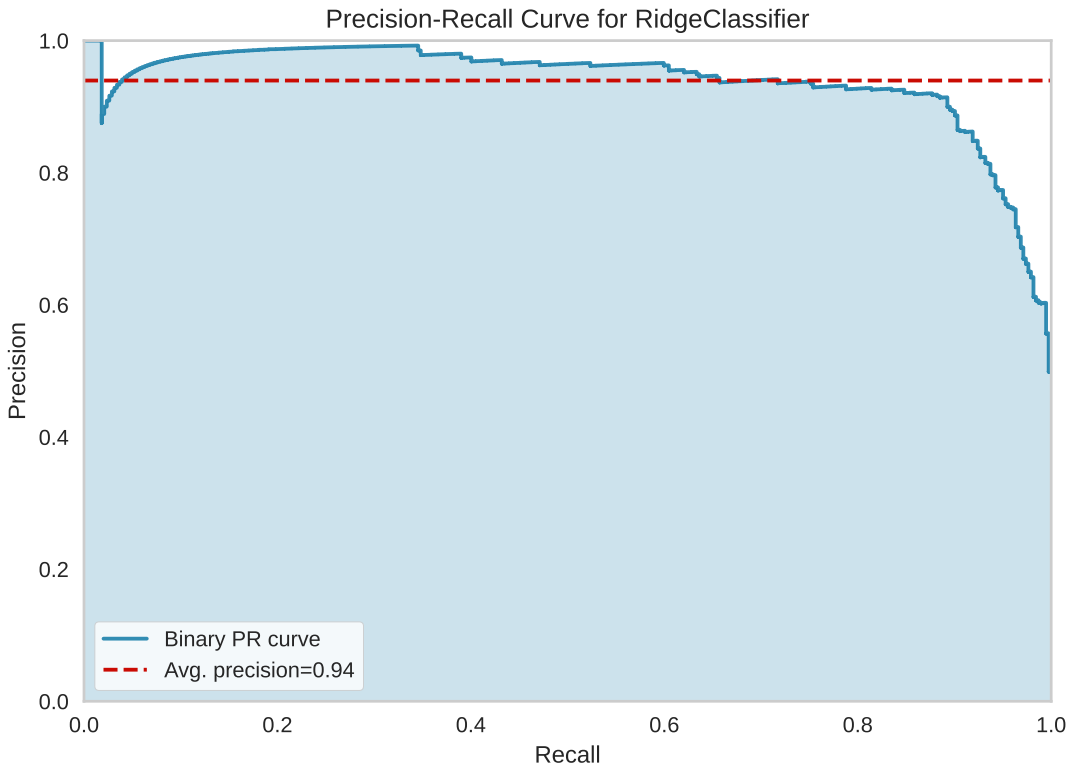
The base case for precision-recall curves is the binary classification case, and this case is also the most visually interpretable. In the figure below we can see the precision plotted on the y-axis against the recall on the x-axis. The larger the filled in area, the stronger the classifier. The red line annotates the average precision.

```
import matplotlib.pyplot as plt

from yellowbrick.datasets import load_spam
from sklearn.linear_model import RidgeClassifier
from yellowbrick.classifier import PrecisionRecallCurve
from sklearn.model_selection import train_test_split as tts
```

(continues on next page)

(continued from previous page)

```
# Load the dataset and split into train/test splits
X, y = load_spam()

X_train, X_test, y_train, y_test = tts(
    X, y, test_size=0.2, shuffle=True, random_state=0
)

# Create the visualizer, fit, score, and show it
viz = PrecisionRecallCurve(RidgeClassifier(random_state=0))
viz.fit(X_train, y_train)
viz.score(X_test, y_test)
viz.show()
```

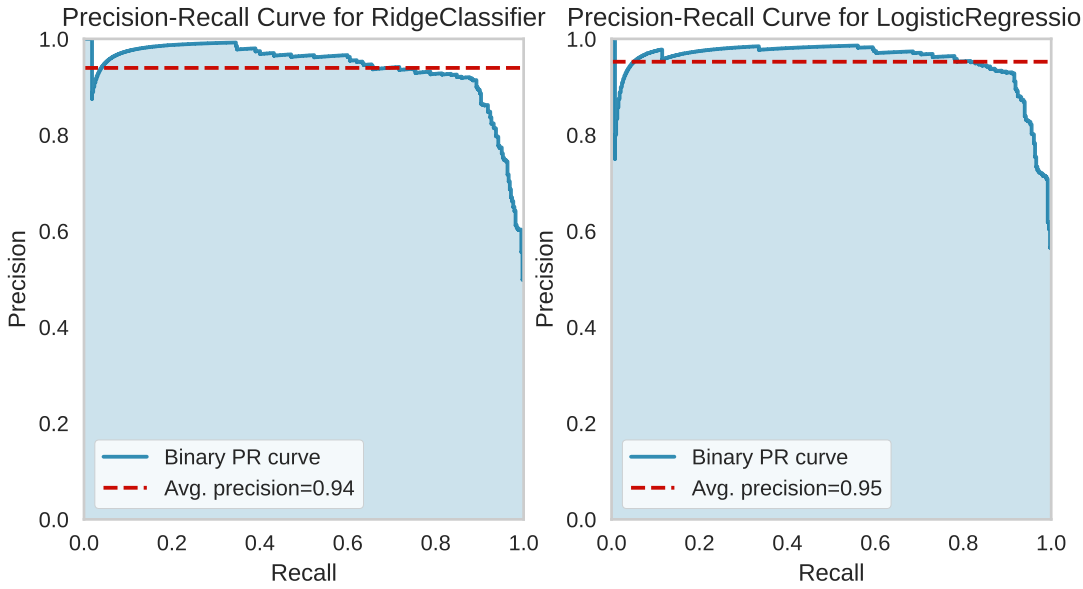


One way to use `PrecisionRecallCurves` is for model comparison, by examining which have the highest average precision. For instance, the below visualization suggest that a `LogisticRegression` model might be better than a `RidgeClassifier` for this particular dataset:

Precision-recall curves are one of the methods used to evaluate a classifier's quality, particularly when classes are very imbalanced. The below plot suggests that our classifier improves when we increase the weight of the "spam" case (which is 1), and decrease the weight for the "not spam" case (which is 0).

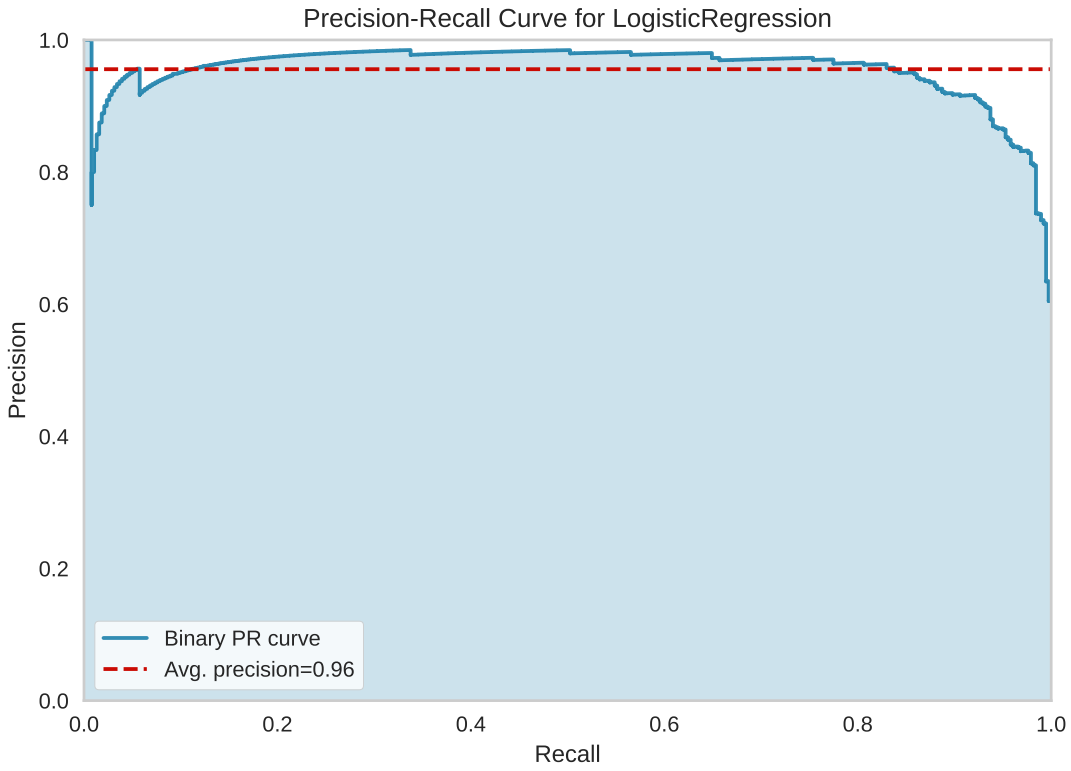
```
from yellowbrick.datasets import load_spam
from sklearn.linear_model import LogisticRegression
```

(continues on next page)



(continued from previous page)

```
from yellowbrick.classifier import PrecisionRecallCurve
from sklearn.model_selection import train_test_split as tts

# Load the dataset and split into train/test splits
X, y = load_spam()

X_train, X_test, y_train, y_test = tts(
    X, y, test_size=0.2, shuffle=True, random_state=0
)

# Specify class weights to shift the threshold towards spam classification
weights = {0:0.2, 1:0.8}

# Create the visualizer, fit, score, and show it
viz = PrecisionRecallCurve(
    LogisticRegression(class_weight=weights, random_state=0)
)
viz.fit(X_train, y_train)
viz.score(X_test, y_test)
viz.show()
```



Multi-Label Classification

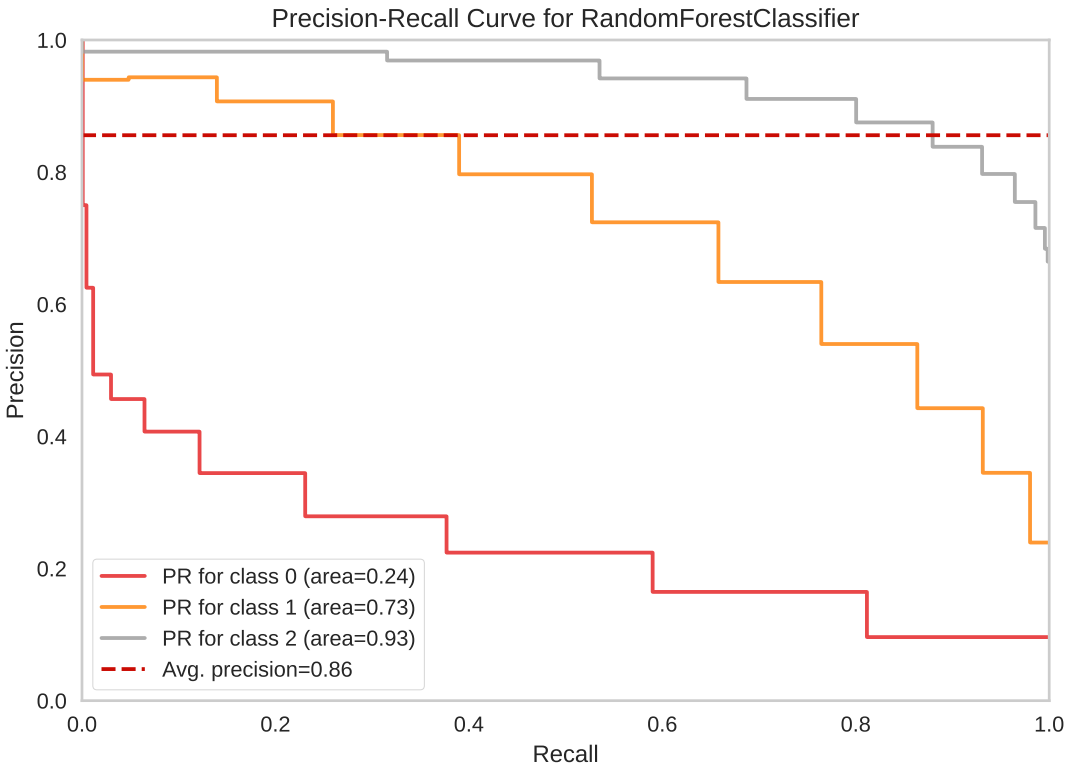
To support multi-label classification, the estimator is wrapped in a OneVsRestClassifier to produce binary comparisons for each class (e.g. the positive case is the class and the negative case is any other class). The precision-recall curve can then be computed as the micro-average of the precision and recall for all classes (by setting `micro=True`), or individual curves can be plotted for each class (by setting `per_class=True`):

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder
from sklearn.model_selection import train_test_split as tts
from yellowbrick.classifier import PrecisionRecallCurve
from yellowbrick.datasets import load_game

# Load dataset and encode categorical variables
X, y = load_game()
X = OrdinalEncoder().fit_transform(X)
y = LabelEncoder().fit_transform(y)

X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2, shuffle=True)

# Create the visualizer, fit, score, and show it
viz = PrecisionRecallCurve(
    RandomForestClassifier(n_estimators=10),
    per_class=True,
```

(continues on next page)

(continued from previous page)

```
    cmap="Set1"
)
viz.fit(X_train, y_train)
viz.score(X_test, y_test)
viz.show()
```



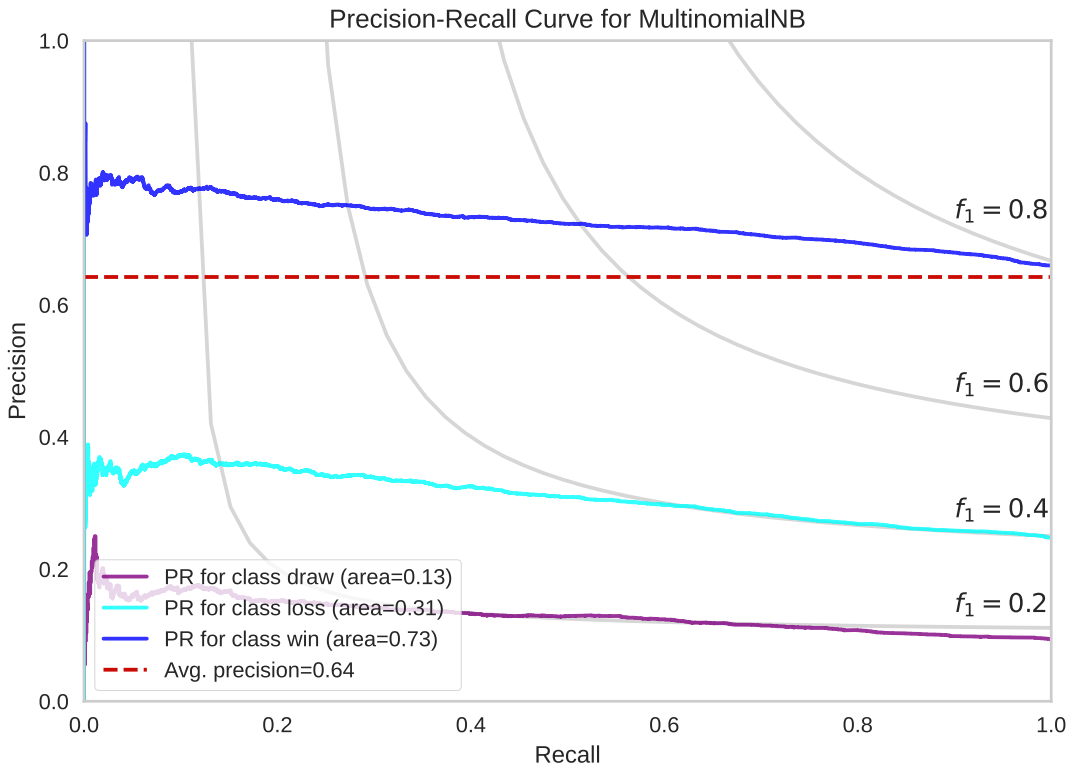
A more complex Precision-Recall curve can be computed, however, displaying the each curve individually, along with F1-score ISO curves (e.g. that show the relationship between precision and recall for various F1 scores).

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder
from sklearn.model_selection import train_test_split as tts
from yellowbrick.classifier import PrecisionRecallCurve
from yellowbrick.datasets import load_game

# Load dataset and encode categorical variables
X, y = load_game()
X = OrdinalEncoder().fit_transform(X)

# Encode the target (we'll use the encoder to retrieve the class labels)
encoder = LabelEncoder()
y = encoder.fit_transform(y)

X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2, shuffle=True)
```

(continues on next page)

(continued from previous page)

```
# Create the visualizer, fit, score, and show it
viz = PrecisionRecallCurve(
    MultinomialNB(),
    classes=encoder.classes_,
    colors=["purple", "cyan", "blue"],
    iso_f1_curves=True,
    per_class=True,
    micro=False
)
viz.fit(X_train, y_train)
viz.score(X_test, y_test)
viz.show()
```



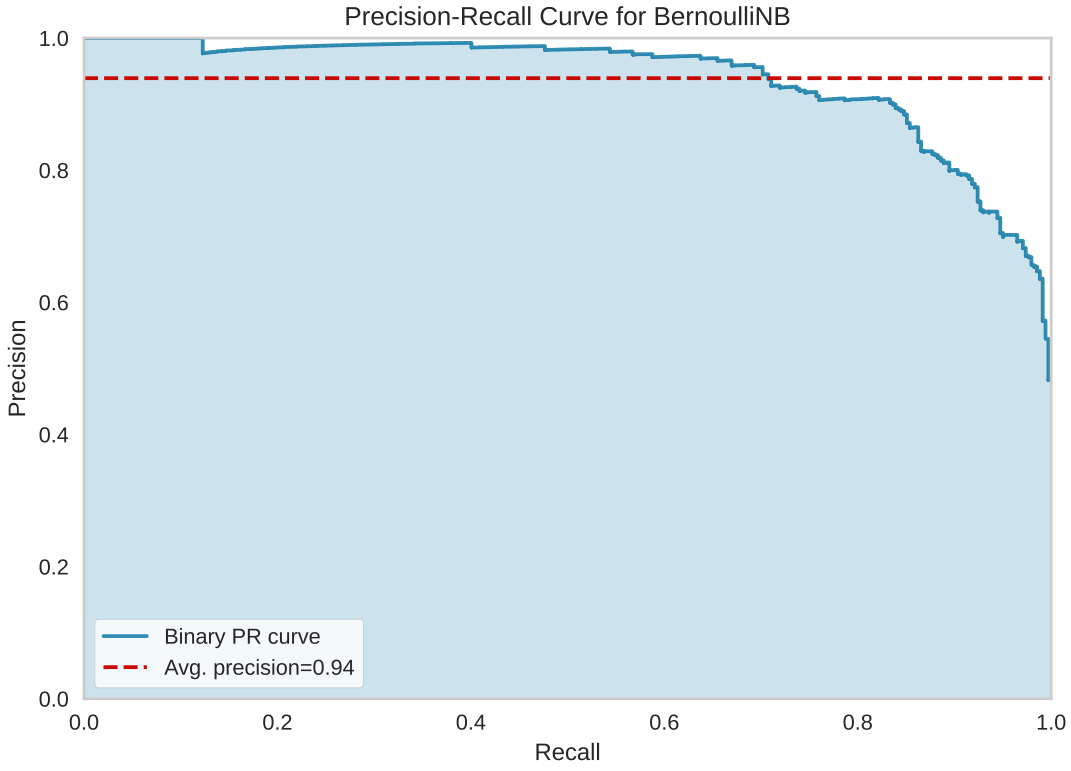
See also:

Scikit-Learn: Model Selection with Precision Recall Curves

Quick Method

Similar functionality as above can be achieved in one line using the associated quick method, `precision_recall_curve`. This method will instantiate and fit a `PrecisionRecallCurve` visualizer on the training data, then will score it on the optionally provided test data (or the training data if it is not provided).

```
from sklearn.naive_bayes import BernoulliNB
from sklearn.model_selection import train_test_split as tts
from yellowbrick.classifier import precision_recall_curve
from yellowbrick.datasets import load_spam

# Load the dataset and split into train/test splits
X, y = load_spam()

X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2, shuffle=True)

# Create the visualizer, fit, score, and show it
viz = precision_recall_curve(BernoulliNB(), X_train, y_train, X_test, y_test)
```



API Reference

Implements Precision-Recall curves for classification models.

class yellowbrick.classifier.prcurve.**PrecisionRecallCurve**(*estimator*, *ax=None*, *classes=None*, *colors=None*, *cmap=None*, *encoder=None*, *fill_area=True*, *ap_score=True*, *micro=True*, *iso_f1_curves=False*, *iso_f1_values=(0.2, 0.4, 0.6, 0.8)*, *per_class=False*, *fill_opacity=0.2*, *line_opacity=0.8*, *is_fitted='auto'*, *force_model=False*, ***kwargs*)

Bases: `ClassificationScoreVisualizer`

Precision-Recall curves are a metric used to evaluate a classifier's quality, particularly when classes are very imbalanced. The precision-recall curve shows the tradeoff between precision, a measure of result relevancy, and recall, a measure of completeness. For each class, precision is defined as the ratio of true positives to the sum of true and false positives, and recall is the ratio of true positives to the sum of true positives and false negatives.

A large area under the curve represents both high recall and precision, the best case scenario for a classifier, showing a model that returns accurate results for the majority of classes it selects.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

classes
[list of str, default: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

colors
[list of strings, default: None] An optional list or tuple of colors to colorize the curves when `per_class=True`. If `per_class=False`, this parameter will be ignored. If both `colors` and `cmap` are provided, `cmap` will be ignored.

cmap
[string or Matplotlib colormap, default: None] An optional string or Matplotlib colormap to colorize the curves when `per_class=True`. If `per_class=False`, this parameter will be ignored. If both `colors` and `cmap` are provided, `cmap` will be ignored.

encoder
[dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

fill_area
[bool, default: True] Fill the area under the curve (or curves) with the curve color.

ap_score
[bool, default: True] Annotate the graph with the average precision score, a summary of the plot that is computed as the weighted mean of precisions at each threshold, with the increase in recall from the previous threshold used as the weight.

micro
[bool, default: True] If multi-class classification, draw the precision-recall curve for the micro-average of all classes. In the multi-class case, either micro or per-class must be set to True. Ignored in the binary case.

iso_f1_curves
[bool, default: False] Draw ISO F1-Curves on the plot to show how close the precision-recall curves are to different F1 scores.

iso_f1_values
[tuple , default: (0.2, 0.4, 0.6, 0.8)] Values of f1 score for which to draw ISO F1-Curves

per_class
[bool, default: False] If multi-class classification, draw the precision-recall curve for each class using a OneVsRestClassifier to compute the recall on a per-class basis. In the multi-class case, either micro or per-class must be set to True. Ignored in the binary case.

fill_opacity
[float, default: 0.2] Specify the alpha or opacity of the fill area (0 being transparent, and 1.0 being completly opaque).

line_opacity
[float, default: 0.8] Specify the alpha or opacity of the lines (0 being transparent, and 1.0 being completly opaque).

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Notes

To support multi-label classification, the estimator is wrapped in a `OneVsRestClassifier` to produce binary comparisons for each class (e.g. the positive case is the class and the negative case is any other class). The precision-recall curve can then be computed as the micro-average of the precision and recall for all classes (by setting micro=True), or individual curves can be plotted for each class (by setting per_class=True).

Note also that some parameters of this visualizer are learned on the `score` method, not only on `fit`.

See also:

https://bit.ly/2kOIeCC

Examples

```
>>> from yellowbrick.classifier import PrecisionRecallCurve
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.svm import LinearSVC
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)
>>> viz = PrecisionRecallCurve(LinearSVC())
>>> viz.fit(X_train, y_train)
>>> viz.score(X_test, y_test)
>>> viz.show()
```

Attributes

target_type_
[str] Either "binary" or "multiclass" depending on the type of target fit to the visualizer. If "multiclass" then the estimator is wrapped in a OneVsRestClassifier classification strategy.

score_
[float or dict of floats] Average precision, a summary of the plot as a weighted mean of precision at each threshold, weighted by the increase in recall from the previous threshold. In the multiclass case, a mapping of class/metric to the average precision score.

precision_
[array or dict of array with shape=[n_thresholds + 1]] Precision values such that element i is the precision of predictions with score >= thresholds[i] and the last element is 1. In the multiclass case, a mapping of class/metric to precision array.

recall_
[array or dict of array with shape=[n_thresholds + 1]] Decreasing recall values such that element i is the recall of predictions with score >= thresholds[i] and the last element is 0. In the multiclass case, a mapping of class/metric to recall array.

classes_
[ndarray of shape (n_classes,)] The class labels observed while fitting.

class_count_
[ndarray of shape (n_classes,)] Number of samples encountered for each class during fitting.

`draw()`

Draws the precision-recall curves computed in score on the axes.

`finalize()`

Finalize the figure by adding titles, labels, and limits.

`fit(`*X*, *y=None*`)`

Fit the classification model; if y is multi-class, then the estimator is adapted with a `OneVsRestClassifier` strategy, otherwise the estimator is fit directly.

`score(`*X*, *y*`)`

Generates the Precision-Recall curve on the specified test data.

Returns

score_
[float] Average precision, a summary of the plot as a weighted mean of precision at each threshold, weighted by the increase in recall from the previous threshold.

yellowbrick.classifier.prcurve.**precision_recall_curve**(*estimator*, *X_train*, *y_train*, *X_test=None*, *y_test=None*, *ax=None*, *classes=None*, *colors=None*, *cmap=None*, *encoder=None*, *fill_area=True*, *ap_score=True*, *micro=True*, *iso_f1_curves=False*, *iso_f1_values=(0.2, 0.4, 0.6, 0.8)*, *per_class=False*, *fill_opacity=0.2*, *line_opacity=0.8*, *is_fitted='auto'*, *force_model=False*, *show=True*, ***kwargs*)

Precision-Recall Curve

Precision-Recall curves are a metric used to evaluate a classifier's quality, particularly when classes are very imbalanced. The precision-recall curve shows the tradeoff between precision, a measure of result relevancy, and recall, a measure of completeness. For each class, precision is defined as the ratio of true positives to the sum of true and false positives, and recall is the ratio of true positives to the sum of true positives and false negatives.

A large area under the curve represents both high recall and precision, the best case scenario for a classifier, showing a model that returns accurate results for the majority of classes it selects.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X_train
[ndarray or DataFrame of shape n x m] A feature array of n instances with m features the model is trained on. Used to fit the visualizer and also to score the visualizer if test splits are not directly specified.

y_train
[ndarray or Series of length n] An array or series of target or class values. Used to fit the visualizer and also to score the visualizer if test splits are not specified.

X_test
[ndarray or DataFrame of shape n x m, default: None] An optional feature array of n instances with m features that the model is scored on if specified, using X_train as the training data.

y_test
[ndarray or Series of length n, default: None] An optional array or series of target or class values that serve as actual labels for X_test for scoring purposes.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

classes
[list of str, default: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

colors
[list of strings, default: None] An optional list or tuple of colors to colorize the curves when `per_class=True`. If `per_class=False`, this parameter will be ignored. If both `colors` and `cmap` are provided, `cmap` will be ignored.

cmap
[string or Matplotlib colormap, default: None] An optional string or Matplotlib colormap to

colorize the curves when `per_class=True`. If `per_class=False`, this parameter will be ignored. If both `colors` and `cmap` are provided, `cmap` will be ignored.

encoder
[dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

fill_area
[bool, default: True] Fill the area under the curve (or curves) with the curve color.

ap_score
[bool, default: True] Annotate the graph with the average precision score, a summary of the plot that is computed as the weighted mean of precisions at each threshold, with the increase in recall from the previous threshold used as the weight.

micro
[bool, default: True] If multi-class classification, draw the precision-recall curve for the micro-average of all classes. In the multi-class case, either micro or per-class must be set to True. Ignored in the binary case.

iso_f1_curves
[bool, default: False] Draw ISO F1-Curves on the plot to show how close the precision-recall curves are to different F1 scores.

iso_f1_values
[tuple , default: (0.2, 0.4, 0.6, 0.8)] Values of f1 score for which to draw ISO F1-Curves

per_class
[bool, default: False] If multi-class classification, draw the precision-recall curve for each class using a OneVsRestClassifier to compute the recall on a per-class basis. In the multi-class case, either micro or per-class must be set to True. Ignored in the binary case.

fill_opacity
[float, default: 0.2] Specify the alpha or opacity of the fill area (0 being transparent, and 1.0 being completly opaque).

line_opacity
[float, default: 0.8] Specify the alpha or opacity of the lines (0 being transparent, and 1.0 being completly opaque).

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Returns

viz
[PrecisionRecallCurve] Returns the visualizer that generates the curve visualization.

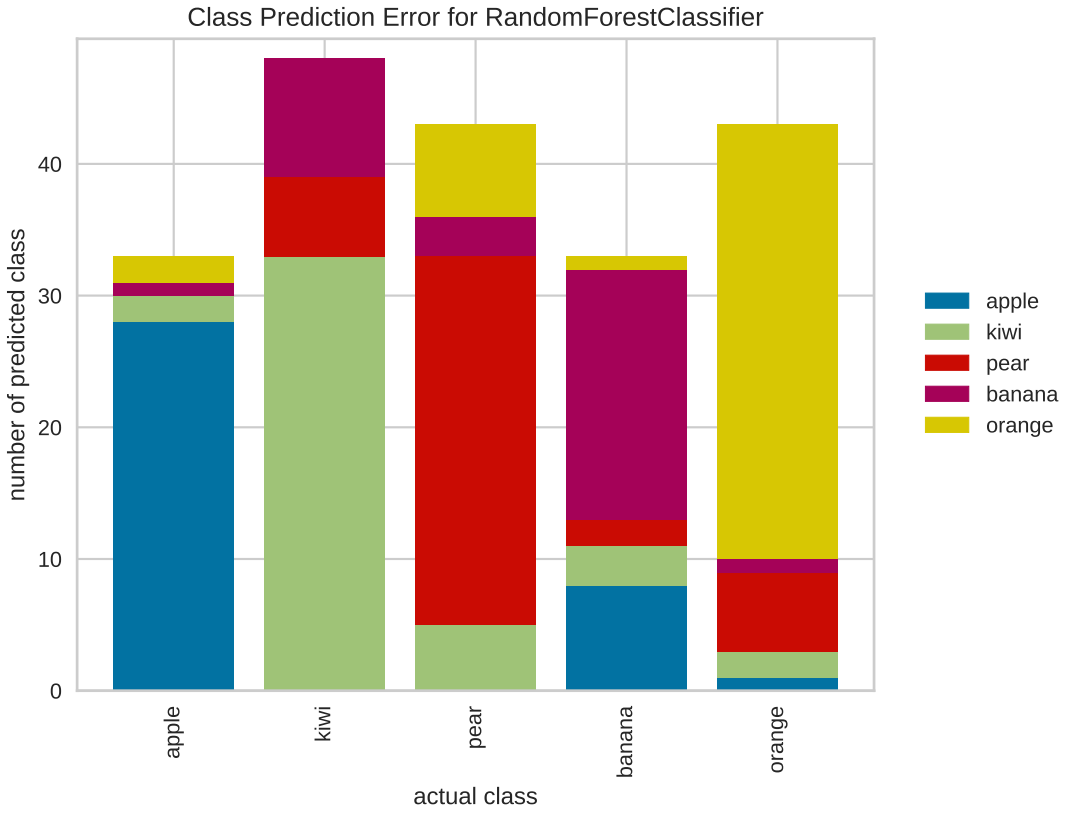
Class Prediction Error

The Yellowbrick `ClassPredictionError` plot is a twist on other and sometimes more familiar classification model diagnostic tools like the *Confusion Matrix* and *Classification Report*. Like the *Classification Report*, this plot shows the support (number of training samples) for each class in the fitted classification model as a stacked bar chart. Each bar is segmented to show the proportion of predictions (including false negatives and false positives, like a *Confusion Matrix*) for each class. You can use a `ClassPredictionError` to visualize which classes your classifier is having a particularly difficult time with, and more importantly, what incorrect answers it is giving on a per-class basis. This can often enable you to better understand strengths and weaknesses of different models and particular challenges unique to your dataset.

The class prediction error chart provides a way to quickly understand how good your classifier is at predicting the right classes.

Visualizer	*ClassPredictionError*
Quick Method	*class_prediction_error()*
Models	Classification
Workflow	Model evaluation

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from yellowbrick.classifier import ClassPredictionError


# Create classification dataset
X, y = make_classification(
    n_samples=1000, n_classes=5, n_informative=3, n_clusters_per_class=1,
    random_state=36,
)

classes = ["apple", "kiwi", "pear", "banana", "orange"]

# Perform 80/20 training/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
                                                    random_state=42)
# Instantiate the classification model and visualizer
visualizer = ClassPredictionError(
    RandomForestClassifier(random_state=42, n_estimators=10), classes=classes
)

# Fit the training data to the visualizer
visualizer.fit(X_train, y_train)

# Evaluate the model on the test data
visualizer.score(X_test, y_test)

# Draw visualization
visualizer.show()
```



In the above example, while the `RandomForestClassifier` appears to be fairly good at correctly predicting apples based on the features of the fruit, it often incorrectly labels pears as kiwis and mistakes kiwis for bananas.

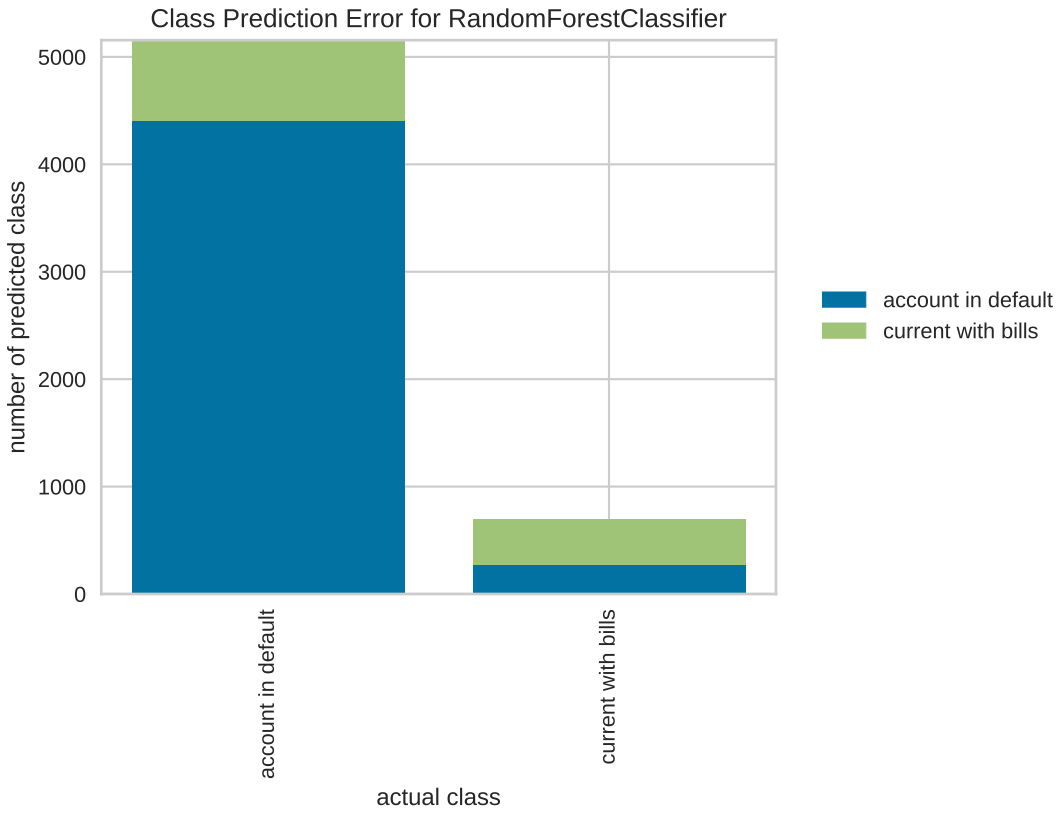
By contrast, in the following example, the `RandomForestClassifier` does a great job at correctly predicting accounts in default, but it is a bit of a coin toss in predicting account holders who stayed current on bills.

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from yellowbrick.classifier import ClassPredictionError
from yellowbrick.datasets import load_credit

X, y = load_credit()

classes = ['account in default', 'current with bills']

# Perform 80/20 training/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
                                                    random_state=42)

# Instantiate the classification model and visualizer
visualizer = ClassPredictionError(
    RandomForestClassifier(n_estimators=10), classes=classes
)

# Fit the training data to the visualizer
```

(continues on next page)

(continued from previous page)

```
visualizer.fit(X_train, y_train)

# Evaluate the model on the test data
visualizer.score(X_test, y_test)

# Draw visualization
visualizer.show()
```



Quick Method

Similar functionality as above can be achieved in one line using the associated quick method, `class_prediction_error`. This method will instantiate and fit a `ClassPredictionError` visualizer on the training data, then will score it on the optionally provided test data (or the training data if it is not provided).

```
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split as tts
from yellowbrick.classifier import class_prediction_error
from yellowbrick.datasets import load_occupancy

# Load the dataset and split into train/test splits
X, y = load_occupancy()
X_train, X_test, y_train, y_test = tts(
```

(continues on next page)

(continued from previous page)

```
    X, y, test_size=0.2, shuffle=True
)

class_prediction_error(
    LinearSVC(random_state=42),
    X_train, y_train, X_test, y_test,
    classes=["vacant", "occupied"]
)
```

API Reference

Shows the balance of classes and their associated predictions.

class yellowbrick.classifier.class_prediction_error.**ClassPredictionError**(*estimator*, *ax=None*, *classes=None*, *encoder=None*, *is_fitted='auto'*, *force_model=False*, ***kwargs*)

Bases: ClassificationScoreVisualizer

Class Prediction Error chart that shows the support for each class in the fitted classification model displayed as a stacked bar. Each bar is segmented to show the distribution of predicted classes for each class. It is initialized with a fitted model and generates a class prediction error chart on draw.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

classes
[list of str, defult: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

encoder
[dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Attributes

classes_
[ndarray of shape (n_classes,)] The class labels observed while fitting.

class_count_
[ndarray of shape (n_classes,)] Number of samples encountered for each class during fitting.

score_
[float] An evaluation metric of the classifier on test data produced when `score()` is called. This metric is between 0 and 1 – higher scores are generally better. For classifiers, this score is usually accuracy, but ensure you check the underlying model for more details about the score.

predictions_
[ndarray] An ndarray of predictions whose rows are the true classes and whose columns are the predicted classes

`draw()`

Renders the class prediction error across the axis.

Returns

ax
[Matplotlib Axes] The axes on which the figure is plotted

`finalize`(***kwargs*)

Adds a title and axis labels to the visualizer, ensuring that the y limits zoom the visualization in to the area of interest. Finalize also calls tight layout to ensure that no parts of the figure are cut off.

Notes

Generally this method is called from show and not directly by the user.

`score`(*X*, *y*)

Generates a 2D array where each row is the count of the predicted classes and each column is the true class

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

Returns

score_
[float] Global accuracy score

yellowbrick.classifier.class_prediction_error.`class_prediction_error`(*estimator*, *X_train*, *y_train*, *X_test=None*, *y_test=None*, *ax=None*, *classes=None*, *encoder=None*, *is_fitted='auto'*, *force_model=False*, *show=True*, ***kwargs*)

Class Prediction Error

Divides the dataset X and y into train and test splits, fits the model on the train split, then scores the model on the test split. The visualizer displays the support for each class in the fitted classification model displayed as a stacked bar plot. Each bar is segmented to show the distribution of predicted classes for each class.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X_train
[ndarray or DataFrame of shape n x m] A feature array of n instances with m features the model is trained on. Used to fit the visualizer and also to score the visualizer if test splits are not directly specified.

y_train
[ndarray or Series of length n] An array or series of target or class values. Used to fit the visualizer and also to score the visualizer if test splits are not specified.

X_test
[ndarray or DataFrame of shape n x m, default: None] An optional feature array of n instances with m features that the model is scored on if specified, using X_train as the training data.

y_test
[ndarray or Series of length n, default: None] An optional array or series of target or class values that serve as actual labels for X_test for scoring purposes.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

classes
[list of str, defult: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

encoder
[dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs: dict
Keyword arguments passed to the visualizer base classes.

Returns

viz
[ClassPredictionError] Returns the fitted, finalized visualizer

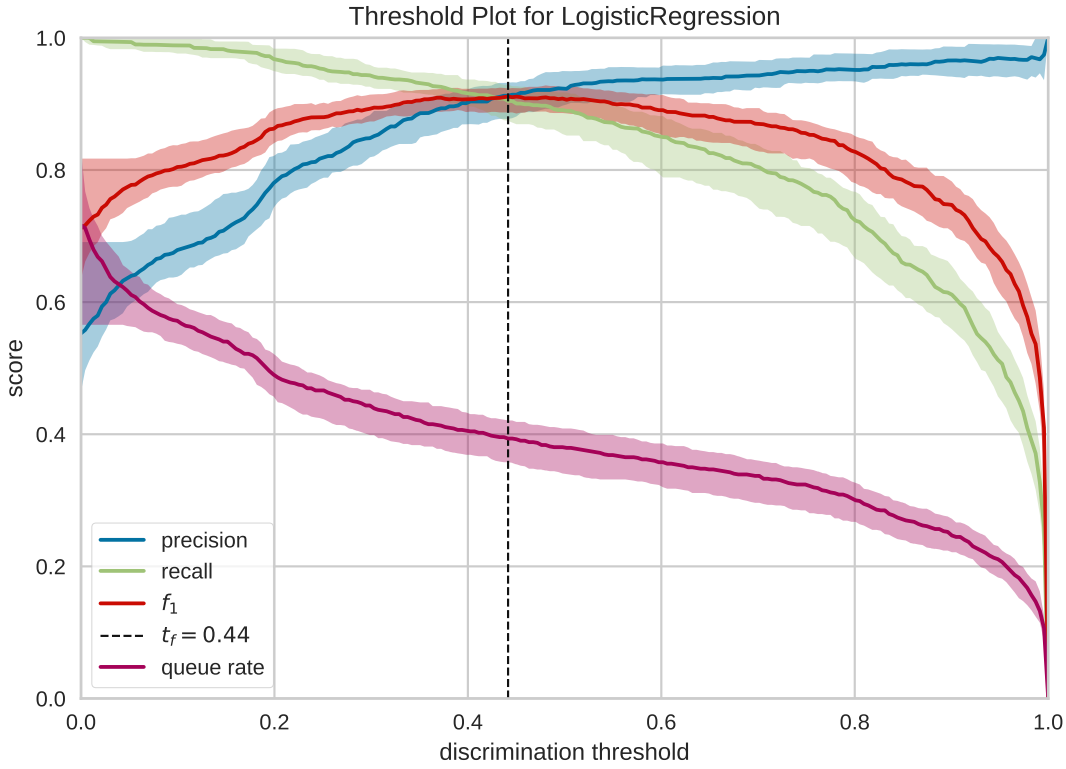
Discrimination Threshold

Caution: This visualizer only works for *binary* classification.

A visualization of precision, recall, f1 score, and queue rate with respect to the discrimination threshold of a binary classifier. The *discrimination threshold* is the probability or score at which the positive class is chosen over the negative class. Generally, this is set to 50% but the threshold can be adjusted to increase or decrease the sensitivity to false positives or to other application factors.

Visualizer	*DiscriminationThreshold*
Quick Method	*discrimination_threshold()*
Models	Classification
Workflow	Model evaluation

```
from sklearn.linear_model import LogisticRegression

from yellowbrick.classifier import DiscriminationThreshold
from yellowbrick.datasets import load_spam

# Load a binary classification dataset
X, y = load_spam()

# Instantiate the classification model and visualizer
model = LogisticRegression(multi_class="auto", solver="liblinear")
visualizer = DiscriminationThreshold(model)

visualizer.fit(X, y)        # Fit the data to the visualizer
visualizer.show()           # Finalize and render the figure
```



One common use of binary classification algorithms is to use the score or probability they produce to determine cases that require special treatment. For example, a fraud prevention application might use a classification algorithm to determine if a transaction is likely fraudulent and needs to be investigated in detail. In the figure above, we present an example where a binary classifier determines if an email is "spam" (the positive case) or "not spam" (the negative case). Emails that are detected as spam are moved to a hidden folder and eventually deleted.

Many classifiers use either a `decision_function` to score the positive class or a `predict_proba` function to compute the probability of the positive class. If the score or probability is greater than some discrimination threshold then the positive class is selected, otherwise, the negative class is.

Generally speaking, the threshold is balanced between cases and set to 0.5 or 50% probability. However, this threshold

may not be the optimal threshold: often there is an inverse relationship between precision and recall with respect to a discrimination threshold. By adjusting the threshold of the classifier, it is possible to tune the F1 score (the harmonic mean of precision and recall) to the best possible fit or to adjust the classifier to behave optimally for the specific application. Classifiers are tuned by considering the following metrics:

- **Precision**: An increase in precision is a reduction in the number of false positives; this metric should be optimized when the cost of special treatment is high (e.g. wasted time in fraud preventing or missing an important email).
- **Recall**: An increase in recall decrease the likelihood that the positive class is missed; this metric should be optimized when it is vital to catch the case even at the cost of more false positives.
- **F1 Score**: The F1 score is the harmonic mean between precision and recall. The `fbeta` parameter determines the relative weight of precision and recall when computing this metric, by default set to 1 or F1. Optimizing this metric produces the best balance between precision and recall.
- **Queue Rate**: The "queue" is the spam folder or the inbox of the fraud investigation desk. This metric describes the percentage of instances that must be reviewed. If review has a high cost (e.g. fraud prevention) then this must be minimized with respect to business requirements; if it doesn't (e.g. spam filter), this could be optimized to ensure the inbox stays clean.

In the figure above we see the visualizer tuned to look for the optimal F1 score, which is annotated as a threshold of 0.43. The model is run multiple times over multiple train/test splits in order to account for the variability of the model with respect to the metrics (shown as the fill area around the median curve).

Quick Method

The same functionality above can be achieved with the associated quick method `discrimination_threshold`. This method will build the `DiscriminationThreshold` object with the associated arguments, fit it, then (optionally) immediately show it

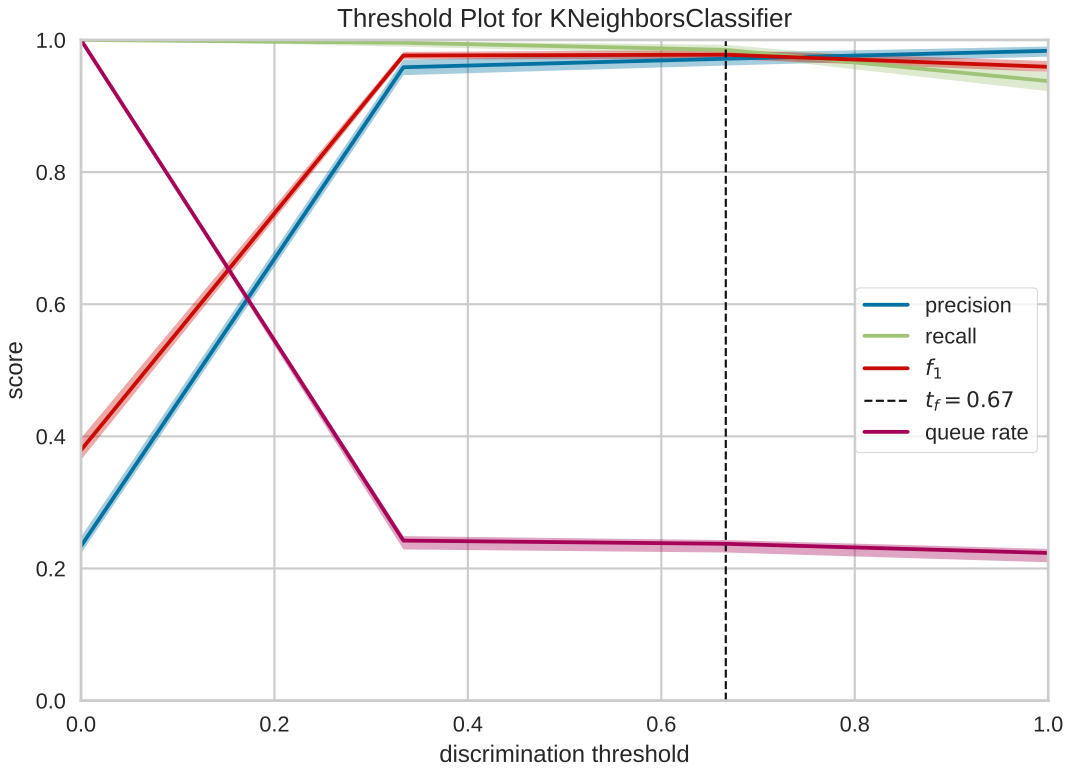
```
from yellowbrick.classifier.threshold import discrimination_threshold
from yellowbrick.datasets import load_occupancy
from sklearn.neighbors import KNeighborsClassifier

#Load the classification dataset
X, y = load_occupancy()

# Instantiate the visualizer with the classification model
model = KNeighborsClassifier(3)

discrimination_threshold(model, X, y)
```

API Reference

DiscriminationThreshold visualizer for probabilistic classifiers.

class yellowbrick.classifier.threshold.**DiscriminationThreshold**(*estimator*, *ax=None*, *n_trials=50*, *cv=0.1*, *fbeta=1.0*, *argmax='fscore'*, *exclude=None*, *quantiles=array([0.1, 0.5, 0.9])*, *random_state=None*, *is_fitted='auto'*, *force_model=False*, ***kwargs*)

Bases: `ModelVisualizer`



Visualizes how precision, recall, f1 score, and queue rate change as the discrimination threshold increases. For probabilistic, binary classifiers, the discrimination threshold is the probability at which you choose the positive class over the negative. Generally this is set to 50%, but adjusting the discrimination threshold will adjust sensitivity to false positives which is described by the inverse relationship of precision and recall with respect to the threshold.

The visualizer also accounts for variability in the model by running multiple trials with different train and test splits of the data. The variability is visualized using a band such that the curve is drawn as the median score of each trial and the band is from the 10th to 90th percentile.

The visualizer is intended to help users determine an appropriate threshold for decision making (e.g. at what threshold do we have a human review the data), given a tolerance for precision and recall or limiting the number of records to check (the queue rate).

Parameters

estimator
: [estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
: [matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

n_trials
: [integer, default: 50] Number of times to shuffle and split the dataset to account for noise in the threshold metrics curves. Note if cv provides > 1 splits, the number of trials will be

n_trials * cv.get_n_splits()

cv
[float or cross-validation generator, default: 0.1] Determines the splitting strategy for each trial. Possible inputs are:

- float, to specify the percent of the test split
- object to be used as cross-validation generator

This attribute is meant to give flexibility with stratified splitting but if a splitter is provided, it should only return one split and have shuffle set to True.

fbeta
[float, 1.0 by default] The strength of recall versus precision in the F-score.

argmax
[str or None, default: 'fscore'] Annotate the threshold maximized by the supplied metric (see exclude for the possible metrics to use). If None or passed to exclude, will not annotate the graph.

exclude
[str or list, optional] Specify metrics to omit from the graph, can include:

- `"precision"`
- `"recall"`
- `"queue_rate"`
- `"fscore"`

Excluded metrics will not be displayed in the graph, nor will they be available in `thresholds_`; however, they will be computed on fit.

quantiles
[sequence, default: np.array([0.1, 0.5, 0.9])] Specify the quantiles to view model variability across a number of trials. Must be monotonic and have three elements such that the first element is the lower bound, the second is the drawn curve, and the third is the upper bound. By default the curve is drawn at the median, and the bounds from the 10th percentile to the 90th percentile.

random_state
[int, optional] Used to seed the random state for shuffling the data while composing different train and test splits. If supplied, the random state is incremented in a deterministic fashion for each split.

Note that if a splitter is provided, it's random state will also be updated with this random state, even if it was previously set.

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Notes

The term "discrimination threshold" is rare in the literature. Here, we use it to mean the probability at which the positive class is selected over the negative class in binary classification.

Classification models must implement either a `decision_function` or `predict_proba` method in order to be used with this class. A `YellowbrickTypeError` is raised otherwise.

Caution: This method only works for binary, probabilistic classifiers.

See also:

For a thorough explanation of discrimination thresholds, see: Visualizing Machine Learning Thresholds to Make Better Business Decisions by Insight Data.

Attributes

thresholds_
[array] The uniform thresholds identified by each of the trial runs.

cv_scores_
[dict of arrays of `len(thresholds_)`] The values for all included metrics including the upper and lower bounds of the metrics defined by quantiles.

`draw()`

Draws the cv scores as a line chart on the current axes.

`finalize`(***kwargs*)

Sets a title and axis labels on the visualizer and ensures that the axis limits are scaled to valid threshold values.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

`fit`(*X*, *y*, ***kwargs*)

Fit is the entry point for the visualizer. Given instances described by X and binary classes described in the target y, fit performs n trials by shuffling and splitting the dataset then computing the precision, recall, f1, and queue rate scores for each trial. The scores are aggregated by the quantiles expressed then drawn.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values. The target y must be a binary classification target.

kwargs: dict
keyword arguments passed to Scikit-Learn API.

Returns

self
[instance] Returns the instance of the visualizer

raises: YellowbrickValueError
If the target y is not a binary classification target.

yellowbrick.classifier.threshold.**discrimination_threshold**(*estimator*, *X*, *y*, *ax=None*, *n_trials=50*, *cv=0.1*, *fbeta=1.0*, *argmax='fscore'*, *exclude=None*, *quantiles=array([0.1, 0.5, 0.9])*, *random_state=None*, *is_fitted='auto'*, *force_model=False*, *show=True*, ***kwargs*)

Discrimination Threshold

Visualizes how precision, recall, f1 score, and queue rate change as the discrimination threshold increases. For probabilistic, binary classifiers, the discrimination threshold is the probability at which you choose the positive class over the negative. Generally this is set to 50%, but adjusting the discrimination threshold will adjust sensitivity to false positives which is described by the inverse relationship of precision and recall with respect to the threshold.

See also:

See DiscriminationThreshold for more details.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values. The target y must be a binary classification target.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

n_trials
[integer, default: 50] Number of times to shuffle and split the dataset to account for noise in the threshold metrics curves. Note if cv provides > 1 splits, the number of trials will be n_trials * cv.get_n_splits()

cv
[float or cross-validation generator, default: 0.1] Determines the splitting strategy for each trial. Possible inputs are:

- float, to specify the percent of the test split
- object to be used as cross-validation generator

This attribute is meant to give flexibility with stratified splitting but if a splitter is provided, it should only return one split and have shuffle set to True.

fbeta
[float, 1.0 by default] The strength of recall versus precision in the F-score.

argmax
[str or None, default: 'fscore'] Annotate the threshold maximized by the supplied metric (see exclude for the possible metrics to use). If None or passed to exclude, will not annotate the graph.

exclude
[str or list, optional] Specify metrics to omit from the graph, can include:

- `"precision"`
- `"recall"`
- `"queue_rate"`
- `"fscore"`

Excluded metrics will not be displayed in the graph, nor will they be available in `thresholds_`; however, they will be computed on fit.

quantiles
[sequence, default: np.array([0.1, 0.5, 0.9])] Specify the quantiles to view model variability across a number of trials. Must be monotonic and have three elements such that the first element is the lower bound, the second is the drawn curve, and the third is the upper bound. By default the curve is drawn at the median, and the bounds from the 10th percentile to the 90th percentile.

random_state
[int, optional] Used to seed the random state for shuffling the data while composing different train and test splits. If supplied, the random state is incremented in a deterministic fashion for each split.

Note that if a splitter is provided, it's random state will also be updated with this random state, even if it was previously set.

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

Returns

viz
[DiscriminationThreshold] Returns the fitted and finalized visualizer object.

Notes

The term "discrimination threshold" is rare in the literature. Here, we use it to mean the probability at which the positive class is selected over the negative class in binary classification.

Classification models must implement either a `decision_function` or `predict_proba` method in order to be used with this class. A `YellowbrickTypeError` is raised otherwise.

See also:

For a thorough explanation of discrimination thresholds, see: Visualizing Machine Learning Thresholds to Make Better Business Decisions by Insight Data.

Examples

```
>>> from yellowbrick.classifier.threshold import discrimination_threshold
>>> from sklearn.linear_model import LogisticRegression
>>> from yellowbrick.datasets import load_occupancy
>>> X, y = load_occupancy()
>>> model = LogisticRegression(multi_class="auto", solver="liblinear")
>>> discrimination_threshold(model, X, y)
```

8.3.7 Clustering Visualizers

Clustering models are unsupervised methods that attempt to detect patterns in unlabeled data. There are two primary classes of clustering algorithm: *agglomerative* clustering links similar data points together, whereas *centroidal* clustering attempts to find centers or partitions in the data. Yellowbrick provides the `yellowbrick.cluster` module to visualize and evaluate clustering behavior. Currently we provide several visualizers to evaluate centroidal mechanisms, particularly K-Means clustering, that help us to discover an optimal K parameter in the clustering metric:

- *Elbow Method*: visualize the clusters according to some scoring function, look for an "elbow" in the curve.
- *Silhouette Visualizer*: visualize the silhouette scores of each cluster in a single model.
- *Intercluster Distance Maps*: visualize the relative distance and size of clusters.

Because it is very difficult to `score` a clustering model, Yellowbrick visualizers wrap scikit-learn clusterer estimators via their `fit()` method. Once the clustering model is trained, then the visualizer can call `show()` to display the clustering evaluation metric.

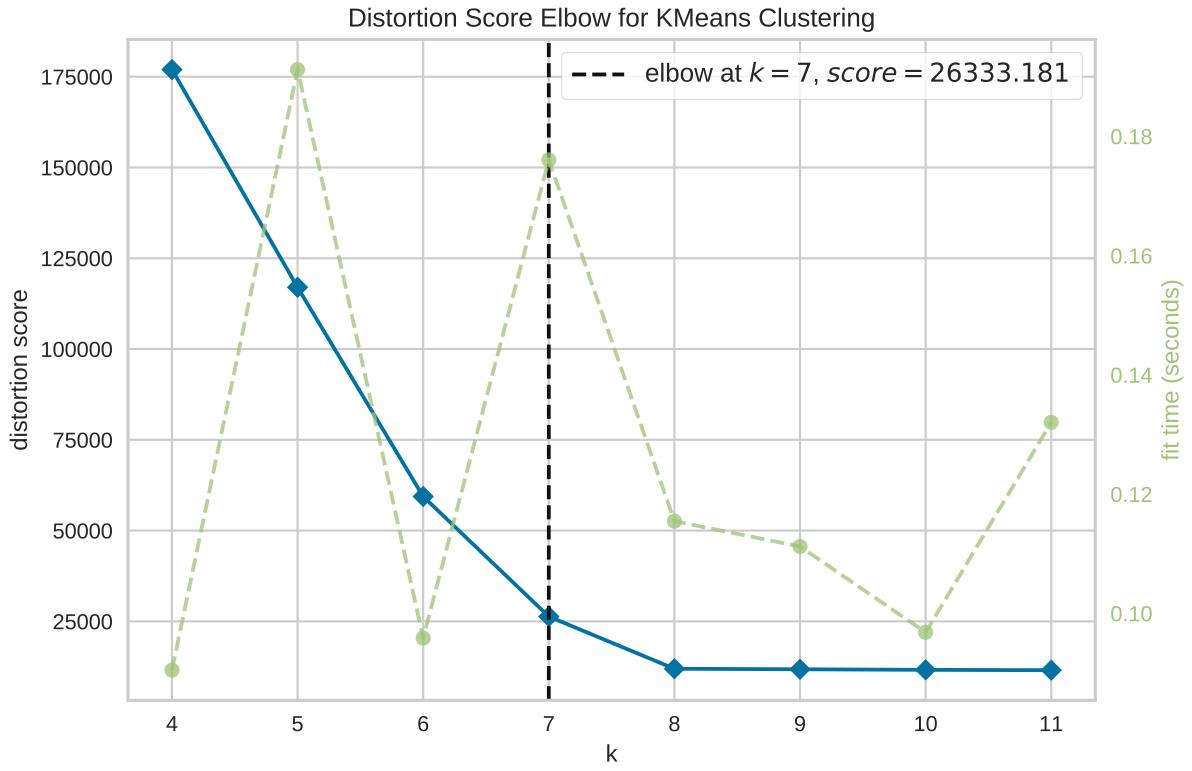
Elbow Method

The `KElbowVisualizer` implements the "elbow" method to help data scientists select the optimal number of clusters by fitting the model with a range of values for K. If the line chart resembles an arm, then the "elbow" (the point of inflection on the curve) is a good indication that the underlying model fits best at that point. In the visualizer "elbow" will be annotated with a dashed line.

To demonstrate, in the following example the `KElbowVisualizer` fits the `KMeans` model for a range of K values from 4 to 11 on a sample two-dimensional dataset with 8 random clusters of points. When the model is fit with 8 clusters, we can see a line annotating the "elbow" in the graph, which in this case we know to be the optimal number.

Visualizer	*KElbowVisualizer*
Quick Method	*kelbow_visualizer()*
Models	Clustering
Workflow	Model evaluation

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

from yellowbrick.cluster import KElbowVisualizer

# Generate synthetic dataset with 8 random clusters
X, y = make_blobs(n_samples=1000, n_features=12, centers=8, random_state=42)

# Instantiate the clustering model and visualizer
model = KMeans()
visualizer = KElbowVisualizer(model, k=(4,12))

visualizer.fit(X)        # Fit the data to the visualizer
visualizer.show()        # Finalize and render the figure
```



By default, the scoring parameter `metric` is set to `distortion`, which computes the sum of squared distances from each point to its assigned center. However, two other metrics can also be used with the `KElbowVisualizer` – `silhouette` and `calinski_harabasz`. The `silhouette` score calculates the mean Silhouette Coefficient of all samples, while the `calinski_harabasz` score computes the ratio of dispersion between and within clusters.

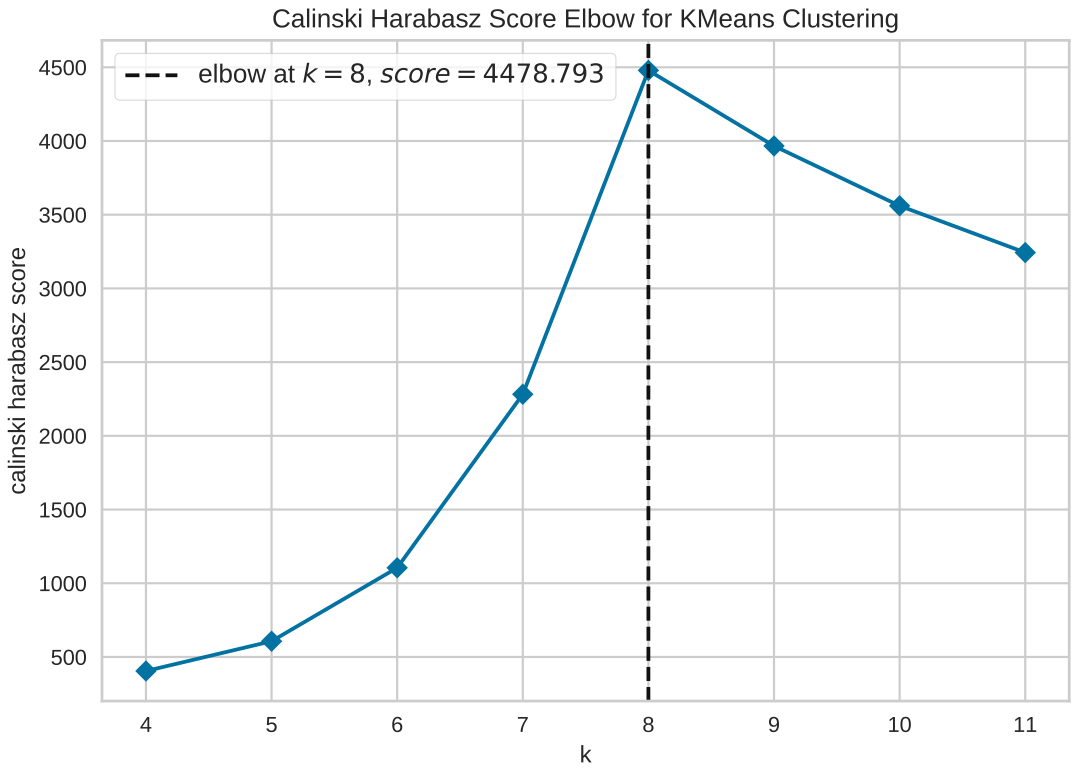
The `KElbowVisualizer` also displays the amount of time to train the clustering model per K as a dashed green line, but is can be hidden by setting `timings=False`. In the following example, we'll use the `calinski_harabasz` score and hide the time to fit the model.

```
from sklearn.cluster import KMeans
```

(continues on next page)

(continued from previous page)

```
from sklearn.datasets import make_blobs

from yellowbrick.cluster import KElbowVisualizer

# Generate synthetic dataset with 8 random clusters
X, y = make_blobs(n_samples=1000, n_features=12, centers=8, random_state=42)

# Instantiate the clustering model and visualizer
model = KMeans()
visualizer = KElbowVisualizer(
    model, k=(4,12), metric='calinski_harabasz', timings=False
)

visualizer.fit(X)        # Fit the data to the visualizer
visualizer.show()        # Finalize and render the figure
```



By default, the parameter `locate_elbow` is set to `True`, which automatically find the "elbow" which likely corresponds to the optimal value of k using the "knee point detection algorithm". However, users can turn off the feature by setting `locate_elbow=False`. You can read about the implementation of this algorithm at "Knee point detection in Python" by Kevin Arvai.

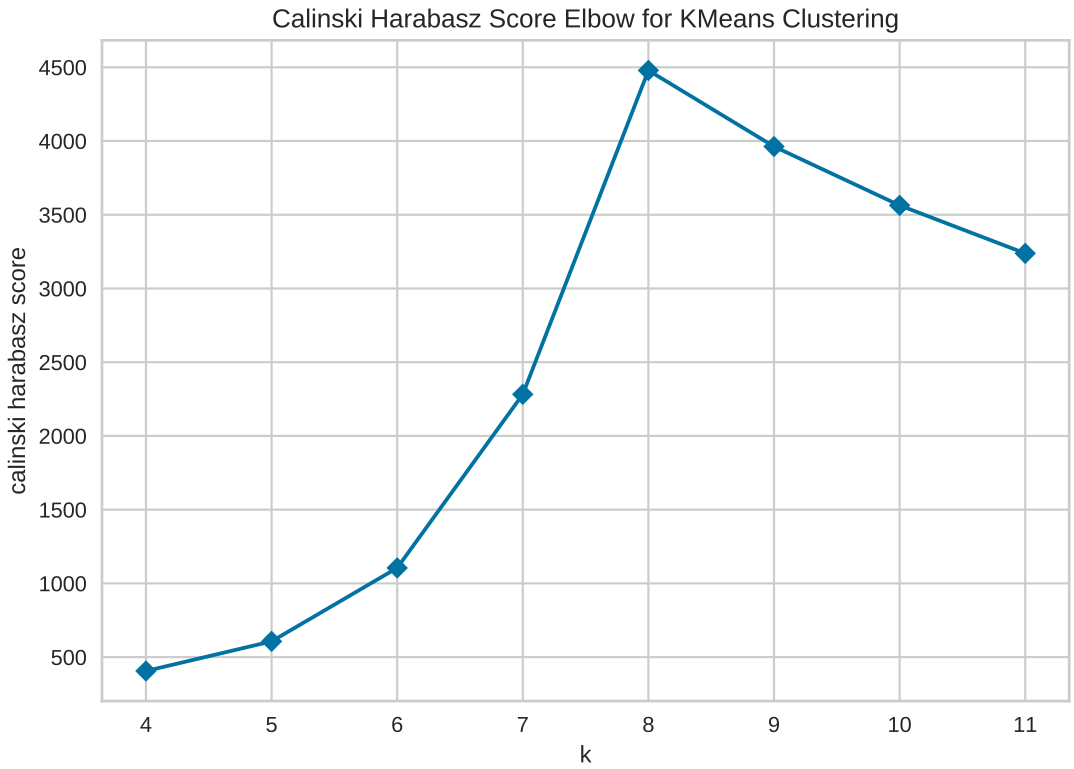
In the following example, we'll use the `calinski_harabasz` score and turn off `locate_elbow` feature.

```
from sklearn.cluster import KMeans
```

(continues on next page)

(continued from previous page)

```
from sklearn.datasets import make_blobs

from yellowbrick.cluster import KElbowVisualizer

# Generate synthetic dataset with 8 random clusters
X, y = make_blobs(n_samples=1000, n_features=12, centers=8, random_state=42)

# Instantiate the clustering model and visualizer
model = KMeans()
visualizer = KElbowVisualizer(
    model, k=(4,12), metric='calinski_harabasz', timings=False, locate_elbow=False
)

visualizer.fit(X)        # Fit the data to the visualizer
visualizer.show()        # Finalize and render the figure
```

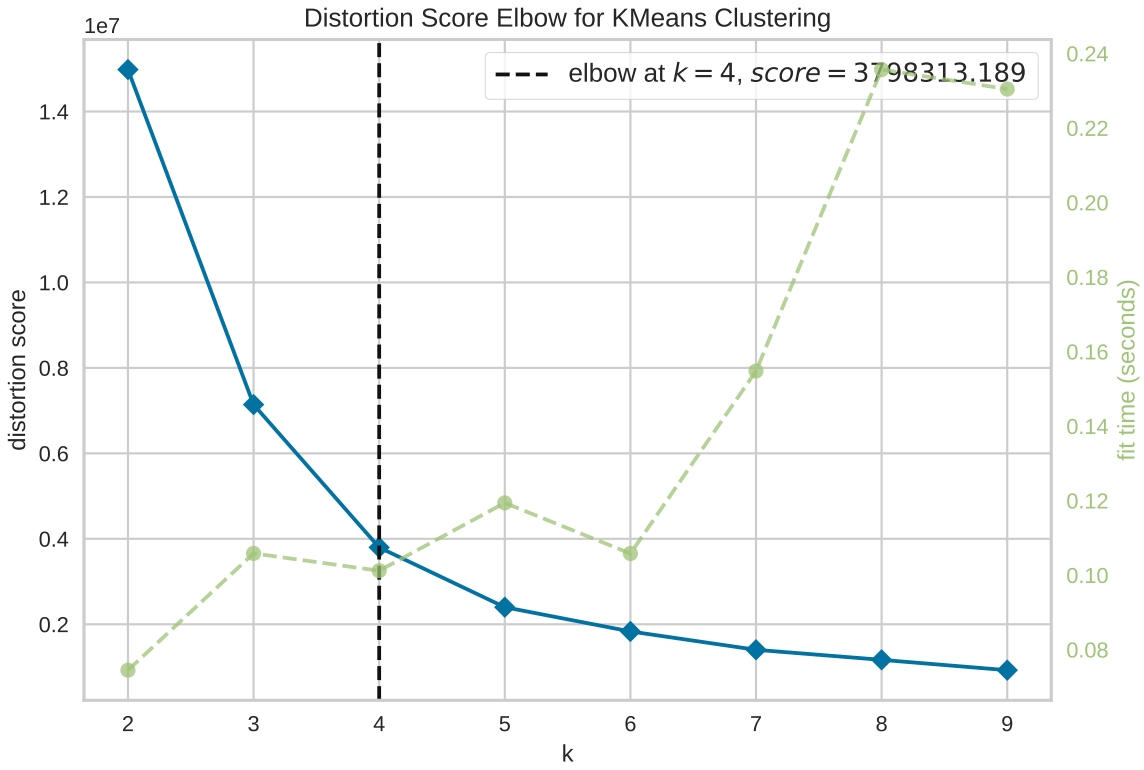


It is important to remember that the "elbow" method does not work well if the data is not very clustered. In this case, you might see a smooth curve and the optimal value of K will be unclear.

Quick Method

The same functionality above can be achieved with the associated quick method `kelbow_visualizer`. This method will build the `KElbowVisualizer` object with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from sklearn.cluster import KMeans
from yellowbrick.cluster.elbow import kelbow_visualizer
from yellowbrick.datasets.loaders import load_nfl

X, y = load_nfl()

# Use the quick method and immediately show the figure
kelbow_visualizer(KMeans(random_state=4), X, k=(2,10))
```



API Reference

Implements the elbow method for determining the optimal number of clusters. https://bl.ocks.org/rpgove/0060ff3b656618e9136b

class yellowbrick.cluster.elbow.**KElbowVisualizer**(*estimator*, *ax=None*, *k=10*, *metric='distortion'*, *distance_metric='euclidean'*, *timings=True*, *locate_elbow=True*, ***kwargs*)

Bases: `ClusteringScoreVisualizer`

The K-Elbow Visualizer implements the "elbow" method of selecting the optimal number of clusters for K-means clustering. K-means is a simple unsupervised machine learning algorithm that groups data into a specified number (k) of clusters. Because the user must specify in advance what k to choose, the algorithm is somewhat naive – it assigns all members to k clusters even if that is not the right k for the dataset.

The elbow method runs k-means clustering on the dataset for a range of values for k (say from 1-10) and then for each value of k computes an average score for all clusters. By default, the `distortion` score is computed, the sum of square distances from each point to its assigned center. Other metrics can also be used such as the `silhouette` score, the mean silhouette coefficient for all samples or the `calinski_harabasz` score, which computes the ratio of dispersion between and within clusters.

When these overall metrics for each model are plotted, it is possible to visually determine the best value for k. If the line chart looks like an arm, then the "elbow" (the point of inflection on the curve) is the best value of k. The "arm" can be either up or down, but if there is a strong inflection point, it is a good indication that the underlying model fits best at that point.

Parameters

estimator
[a scikit-learn clusterer] Should be an instance of an unfitted clusterer, specifically `KMeans` or `MiniBatchKMeans`. If it is not a clusterer, an exception is raised.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

k
[integer, tuple, or iterable] The k values to compute silhouette scores for. If a single integer is specified, then will compute the range (2,k). If a tuple of 2 integers is specified, then k will be in np.arange(k[0], k[1]). Otherwise, specify an iterable of integers to use as values for k.

metric
[string, default: `"distortion"`] Select the scoring metric to evaluate the clusters. The default is the mean distortion, defined by the sum of squared distances between each observation and its closest centroid. Other metrics include:

- **distortion**: mean sum of squared distances to centers
- **silhouette**: mean ratio of intra-cluster and nearest-cluster distance
- **calinski_harabasz**: ratio of within to between cluster dispersion

distance_metric
[str or callable, default='euclidean'] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by sklearn's metrics.pairwise.pairwise_distances. If X is the distance array itself, use metric="precomputed".

timings
[bool, default: True] Display the fitting time per k to evaluate the amount of time required to

train the clustering model.

locate_elbow
[bool, default: True] Automatically find the "elbow" or "knee" which likely corresponds to the optimal value of k using the "knee point detection algorithm". The knee point detection algorithm finds the point of maximum curvature, which in a well-behaved clustering problem also represents the pivot of the elbow curve. The point is labeled with a dashed line and annotated with the score and k values.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

If you get a visualizer that doesn't have an elbow or inflection point, then this method may not be working. The elbow method does not work well if the data is not very clustered; in this case, you might see a smooth curve and the value of k is unclear. Other scoring methods, such as BIC or SSE, also can be used to explore if clustering is a correct choice.

For a discussion on the Elbow method, read more at Robert Gove's Block website. For more on the knee point detection algorithm see the paper "Finding a "kneedle" in a Haystack".

See also:

The scikit-learn documentation for the silhouette_score and calinski_harabasz_score. The default, `distortion_score`, is implemented in `yellowbrick.cluster.elbow`.

Examples

```
>>> from yellowbrick.cluster import KElbowVisualizer
>>> from sklearn.cluster import KMeans
>>> model = KElbowVisualizer(KMeans(), k=10)
>>> model.fit(X)
>>> model.show()
```

Attributes

k_scores_
[array of shape (n,) where n is no. of k values] The silhouette score corresponding to each k value.

k_timers_
[array of shape (n,) where n is no. of k values] The time taken to fit n KMeans model corresponding to each k value.

elbow_value_
[integer] The optimal value of k.

elbow_score_
[float] The silhouette score corresponding to the optimal value of k.

`draw()`
Draw the elbow curve for the specified scores and values of K.

finalize()

Prepare the figure for rendering by setting the title as well as the X and Y axis labels and adding the legend.

fit(*X*, *y=None*, ***kwargs*)

Fits n KMeans models where n is the length of `self.k_values_`, storing the silhouette scores in the `self.k_scores_` attribute. The "elbow" and silhouette score corresponding to it are stored in `self.elbow_value` and `self.elbow_score` respectively. This method finishes up by calling draw to create the plot.

property metric_color

property timing_color

property vline_color

yellowbrick.cluster.elbow.**kelbow_visualizer**(*model*, *X*, *y=None*, *ax=None*, *k=10*, *metric='distortion'*, *distance_metric='euclidean'*, *timings=True*, *locate_elbow=True*, *show=True*, ***kwargs*)

Quick Method:

model

[a Scikit-Learn clusterer] Should be an instance of an unfitted clusterer, specifically `KMeans` or `MiniBatchKMeans`. If it is not a clusterer, an exception is raised.

X

[array-like of shape (n, m)] A matrix or data frame with n instances and m features

y

[array-like of shape (n,), optional] A vector or series representing the target for each instance

ax

[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

k

[integer, tuple, or iterable] The k values to compute silhouette scores for. If a single integer is specified, then will compute the range (2,k). If a tuple of 2 integers is specified, then k will be in np.arange(k[0], k[1]). Otherwise, specify an iterable of integers to use as values for k.

metric

[string, default: `"distortion"`] Select the scoring metric to evaluate the clusters. The default is the mean distortion, defined by the sum of squared distances between each observation and its closest centroid. Other metrics include:

- **distortion**: mean sum of squared distances to centers
- **silhouette: mean ratio of intra-cluster and nearest-cluster distance**
- **calinski_harabasz**: ratio of within to between cluster dispersion

distance_metric

[str or callable, default='euclidean'] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by sklearn's metrics.pairwise.pairwise_distances. If X is the distance array itself, use metric="precomputed".

timings

[bool, default: True] Display the fitting time per k to evaluate the amount of time required to train the clustering model.

locate_elbow
[bool, default: True] Automatically find the "elbow" or "knee" which likely corresponds to the optimal value of k using the "knee point detection algorithm". The knee point detection algorithm finds the point of maximum curvature, which in a well-behaved clustering problem also represents the pivot of the elbow curve. The point is labeled with a dashed line and annotated with the score and k values.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

viz
[KElbowVisualizer] The kelbow visualizer, fitted and finalized.

Silhouette Visualizer

The Silhouette Coefficient is used when the ground-truth about the dataset is unknown and computes the density of clusters computed by the model. The score is computed by averaging the silhouette coefficient for each sample, computed as the difference between the average intra-cluster distance and the mean nearest-cluster distance for each sample, normalized by the maximum value. This produces a score between 1 and -1, where 1 is highly dense clusters and -1 is completely incorrect clustering.

The Silhouette Visualizer displays the silhouette coefficient for each sample on a per-cluster basis, visualizing which clusters are dense and which are not. This is particularly useful for determining cluster imbalance, or for selecting a value for K by comparing multiple visualizers.

Visualizer	*SilhouetteVisualizer*
Quick Method	*silhouette_visualizer()*
Models	Clustering
Workflow	Model evaluation

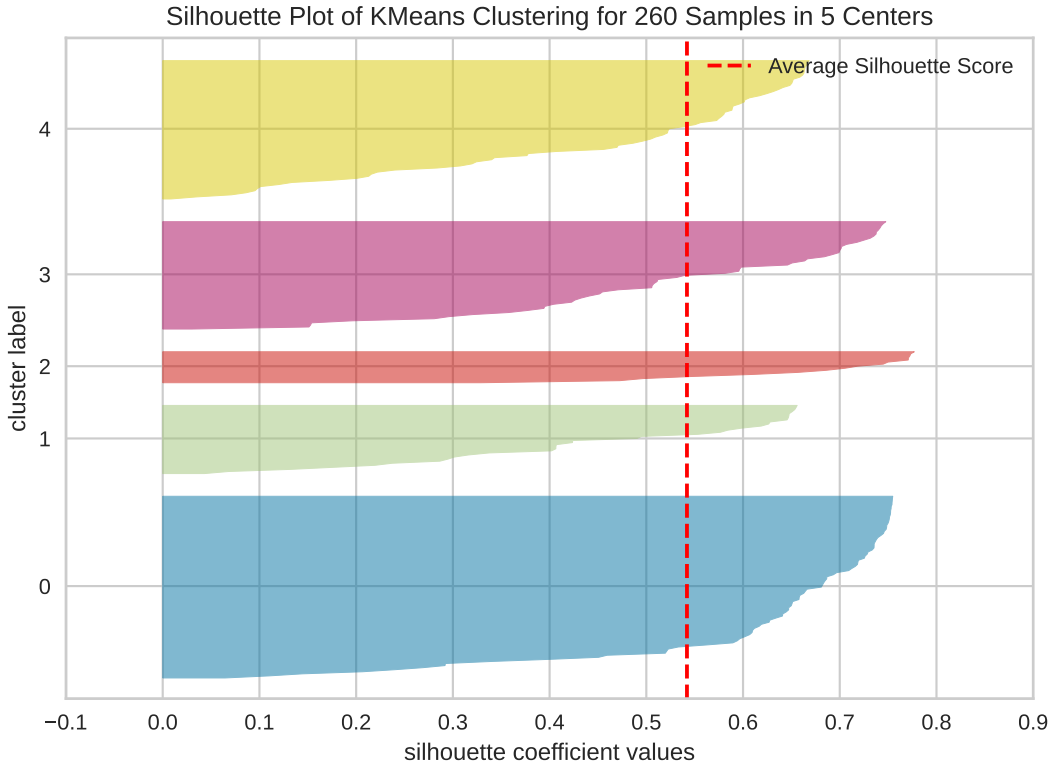
Examples and demo

```
from sklearn.cluster import KMeans

from yellowbrick.cluster import SilhouetteVisualizer
from yellowbrick.datasets import load_nfl

# Load a clustering dataset
X, y = load_nfl()

# Specify the features to use for clustering
features = ['Rec', 'Yds', 'TD', 'Fmb', 'Ctch_Rate']
X = X.query('Tgt >= 20')[features]

# Instantiate the clustering model and visualizer
model = KMeans(5, random_state=42)
visualizer = SilhouetteVisualizer(model, colors='yellowbrick')
```

(continues on next page)

(continued from previous page)

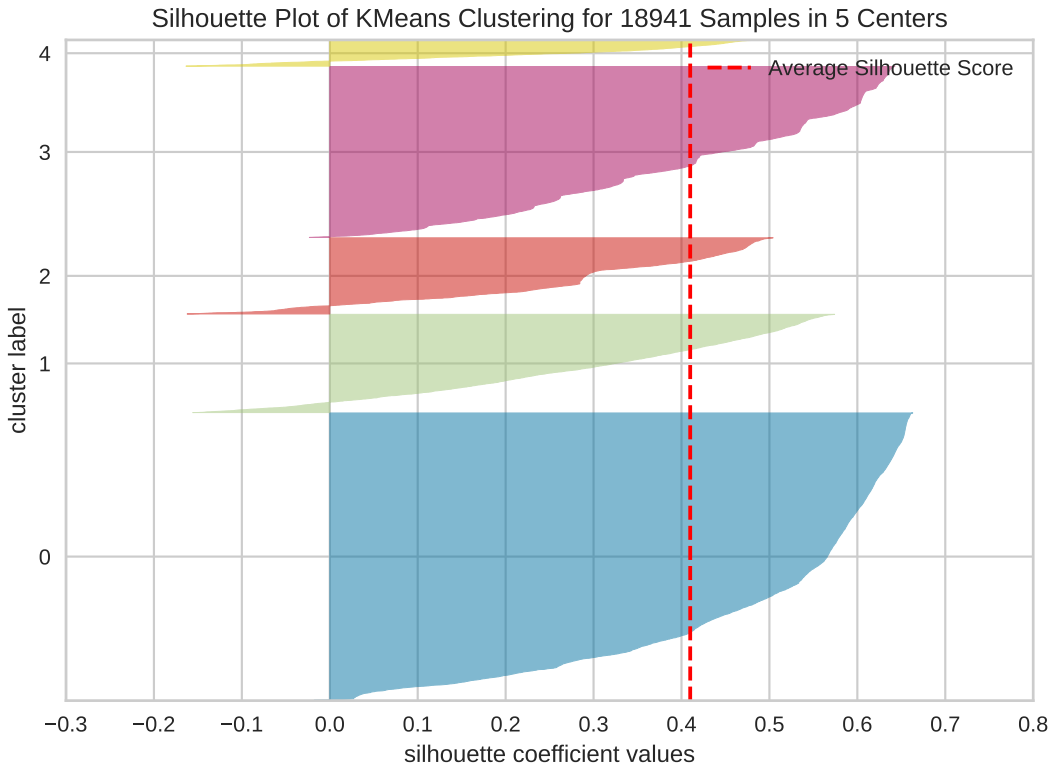
```
visualizer.fit(X)        # Fit the data to the visualizer
visualizer.show()        # Finalize and render the figure
```



Quick Method

The same functionality above can be achieved with the associated quick method *silhouette_visualizer*. This method will build the Silhouette Visualizer object with the associated arguments, fit it, then (optionally) immediately show it.

```
from sklearn.cluster import KMeans

from yellowbrick.cluster import silhouette_visualizer
from yellowbrick.datasets import load_credit

# Load a clustering dataset
X, y = load_credit()

# Specify rows to cluster: under 40 y/o and have either graduate or university education
X = X[(X['age'] <= 40) & (X['edu'].isin([1,2]))]

# Use the quick method and immediately show the figure
silhouette_visualizer(KMeans(5, random_state=42), X, colors='yellowbrick')
```



API Reference

Implements visualizers that use the silhouette metric for cluster evaluation.

class yellowbrick.cluster.silhouette.**SilhouetteVisualizer**(*estimator*, *ax=None*, *colors=None*, *is_fitted='auto'*, ***kwargs*)

Bases: ClusteringScoreVisualizer

The Silhouette Visualizer displays the silhouette coefficient for each sample on a per-cluster basis, visually evaluating the density and separation between clusters. The score is calculated by averaging the silhouette coefficient for each sample, computed as the difference between the average intra-cluster distance and the mean nearest-cluster distance for each sample, normalized by the maximum value. This produces a score between -1 and +1, where scores near +1 indicate high separation and scores near -1 indicate that the samples may have been assigned to the wrong cluster.

In SilhouetteVisualizer plots, clusters with higher scores have wider silhouettes, but clusters that are less cohesive will fall short of the average score across all clusters, which is plotted as a vertical dotted red line.

This is particularly useful for determining cluster imbalance, or for selecting a value for K by comparing multiple visualizers.

Parameters

estimator
[a Scikit-Learn clusterer] Should be an instance of a centroidal clustering algorithm (`KMeans` or `MiniBatchKMeans`). If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

colors
[iterable or string, default: None] A collection of colors to use for each cluster group. If there are fewer colors than cluster groups, colors will repeat. May also be a Yellowbrick or matplotlib colormap string.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> from yellowbrick.cluster import SilhouetteVisualizer
>>> from sklearn.cluster import KMeans
>>> model = SilhouetteVisualizer(KMeans(10))
>>> model.fit(X)
>>> model.show()
```

Attributes

silhouette_score_
[float] Mean Silhouette Coefficient for all samples. Computed via scikit-learn *sklearn.metrics.silhouette_score*.

silhouette_samples_
[array, shape = [n_samples]] Silhouette Coefficient for each samples. Computed via scikit-learn *sklearn.metrics.silhouette_samples*.

n_samples_
[integer] Number of total samples in the dataset (X.shape[0])

n_clusters_
[integer] Number of clusters (e.g. n_clusters or k value) passed to internal scikit-learn model.

y_tick_pos_
[array of shape (n_clusters,)] The computed center positions of each cluster on the y-axis

`draw`(*labels*)

Draw the silhouettes for each sample and the average score.

Parameters

labels
[array-like] An array with the cluster label for each silhouette sample, usually computed with `predict()`. Labels are not stored on the visualizer so that the figure can be redrawn with new data.

`finalize`()

Prepare the figure for rendering by setting the title and adjusting the limits on the axes, adding labels and a legend.

fit(*X*, *y=None*, ***kwargs*)

Fits the model and generates the silhouette visualization.

yellowbrick.cluster.silhouette.**silhouette_visualizer**(*estimator*, *X*, *y=None*, *ax=None*, *colors=None*, *is_fitted='auto'*, *show=True*, ***kwargs*)

Quick Method: The Silhouette Visualizer displays the silhouette coefficient for each sample on a per-cluster basis, visually evaluating the density and separation between clusters. The score is calculated by averaging the silhouette coefficient for each sample, computed as the difference between the average intra-cluster distance and the mean nearest-cluster distance for each sample, normalized by the maximum value. This produces a score between -1 and +1, where scores near +1 indicate high separation and scores near -1 indicate that the samples may have been assigned to the wrong cluster.

Parameters

estimator
[a Scikit-Learn clusterer] Should be an instance of a centroidal clustering algorithm (`KMeans` or `MiniBatchKMeans`). If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X
[array-like of shape (n, m)] A matrix or data frame with n instances and m features

y
[array-like of shape (n,), optional] A vector or series representing the target for each instance

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

colors
[iterable or string, default: None] A collection of colors to use for each cluster group. If there are fewer colors than cluster groups, colors will repeat. May also be a Yellowbrick or matplotlib colormap string.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

viz
[SilhouetteVisualizer] The silhouette visualizer, fitted and finalized.

Intercluster Distance Maps

Intercluster distance maps display an embedding of the cluster centers in 2 dimensions with the distance to other centers preserved. E.g. the closer to centers are in the visualization, the closer they are in the original feature space. The clusters are sized according to a scoring metric. By default, they are sized by membership, e.g. the number of instances that belong to each center. This gives a sense of the relative importance of clusters. Note however, that because two clusters overlap in the 2D space, it does not imply that they overlap in the original feature space.

Visualizer	*InterclusterDistance*
Quick Method	*intercluster_distance()*
Models	Clustering
Workflow	Model evaluation

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

from yellowbrick.cluster import InterclusterDistance

# Generate synthetic dataset with 12 random clusters
X, y = make_blobs(n_samples=1000, n_features=12, centers=12, random_state=42)

# Instantiate the clustering model and visualizer
model = KMeans(6)
visualizer = InterclusterDistance(model)

visualizer.fit(X)        # Fit the data to the visualizer
visualizer.show()        # Finalize and render the figure
```

Quick Method

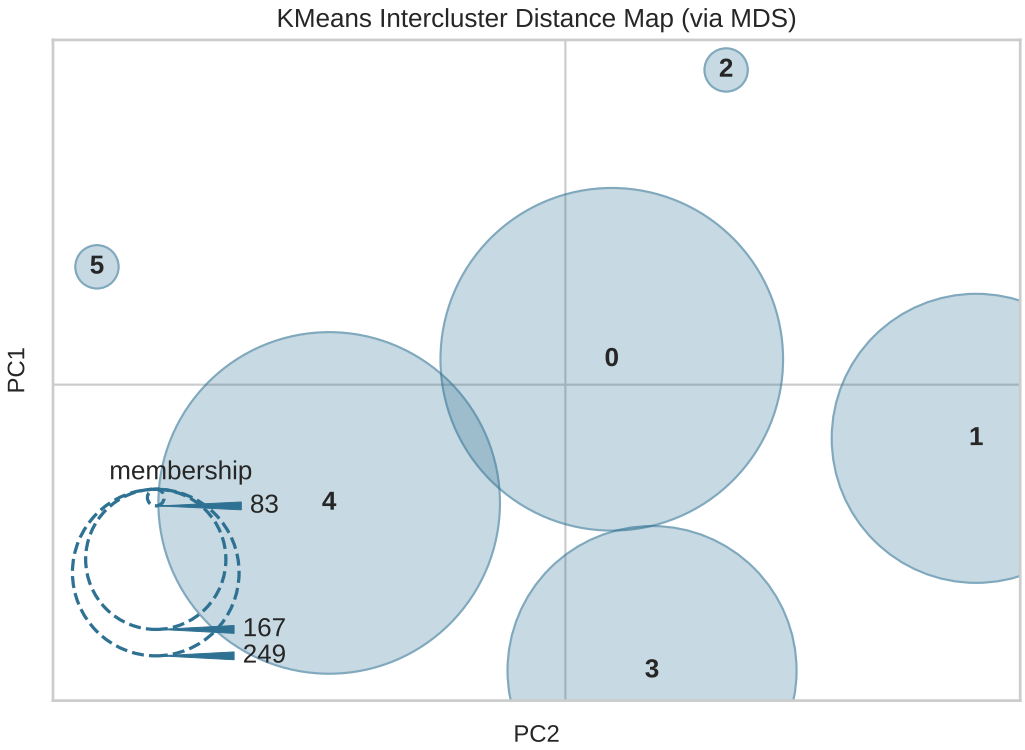
The same functionality above can be achieved with the associated quick method *intercluster_distance*. This method will build the `InterclusterDistance` object with the associated arguments, fit it, then (optionally) immediately show it.

```
from yellowbrick.datasets import load_nfl
from sklearn.cluster import MiniBatchKMeans
from yellowbrick.cluster import intercluster_distance


X, _ = load_nfl()
intercluster_distance(MiniBatchKMeans(5, random_state=777), X)
```

API Reference

Implements Intercluster Distance Map visualizations.

class yellowbrick.cluster.icdm.**InterclusterDistance**(*estimator*, *ax=None*, *min_size=400*, *max_size=25000*, *embedding='mds'*, *scoring='membership'*, *legend=True*, *legend_loc='lower left'*, *legend_size=1.5*, *random_state=None*, *is_fitted='auto'*, ***kwargs*)



Bases: `ClusteringScoreVisualizer`

Intercluster distance maps display an embedding of the cluster centers in 2 dimensions with the distance to other centers preserved. E.g. the closer to centers are in the visualization, the closer they are in the original feature space. The clusters are sized according to a scoring metric. By default, they are sized by membership, e.g. the number of instances that belong to each center. This gives a sense of the relative importance of clusters. Note however, that because two clusters overlap in the 2D space, it does not imply that they overlap in the original feature space.

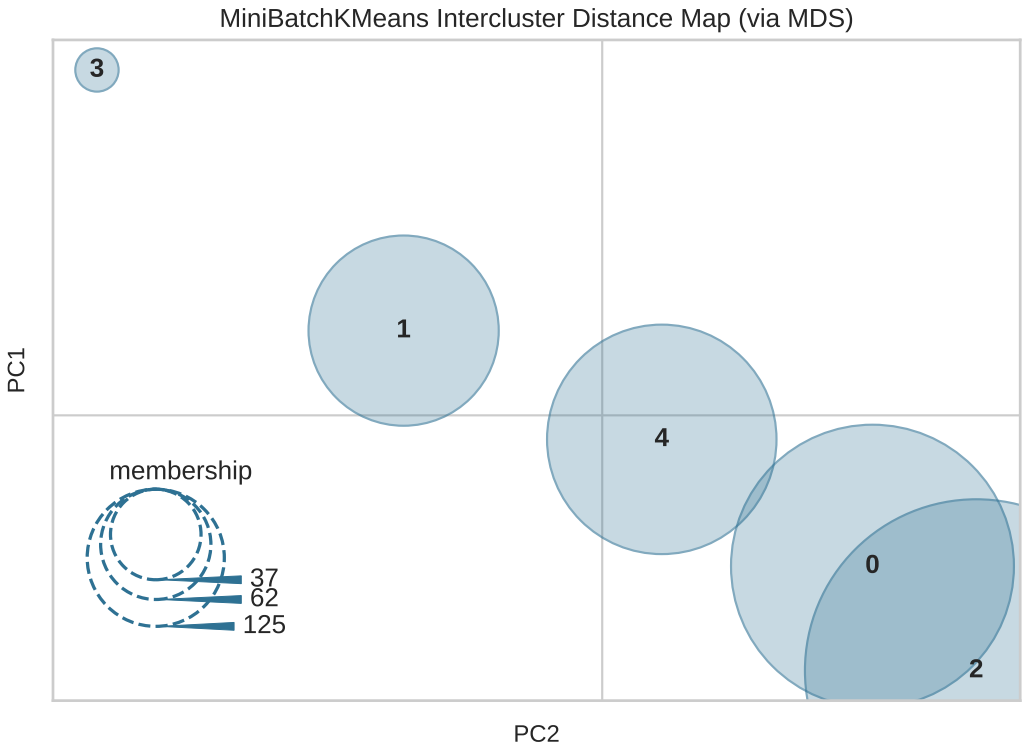
Parameters

estimator
[a Scikit-Learn clusterer] Should be an instance of a centroidal clustering algorithm (or a hierarchical algorithm with a specified number of clusters). Also accepts some other models like LDA for text clustering. If it is not a clusterer, an exception is raised. If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

min_size
[int, default: 400] The size, in points, of the smallest cluster drawn on the graph. Cluster sizes will be scaled between the min and max sizes.

max_size
[int, default: 25000] The size, in points, of the largest cluster drawn on the graph. Cluster



sizes will be scaled between the min and max sizes.

embedding
[default: 'mds'] The algorithm used to embed the cluster centers in 2 dimensional space so that the distance between clusters is represented equivalently to their relationship in feature spaceself. Embedding algorithm options include:

- **mds**: multidimensional scaling
- **tsne**: stochastic neighbor embedding

scoring
[default: 'membership'] The scoring method used to determine the size of the clusters drawn on the graph so that the relative importance of clusters can be viewed. Scoring method options include:

- **membership**: number of instances belonging to each cluster

legend
[bool, default: True] Whether or not to draw the size legend onto the graph, omit the legend to more easily see clusters that overlap.

legend_loc
[str, default: "lower left"] The location of the legend on the graph, used to move the legend out of the way of clusters into open space. The same legend location options for matplotlib are used here.

See also:

https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.legend

legend_size
[float, default: 1.5] The size, in inches, of the size legend to inset into the graph.

random_state
[int or RandomState, default: None] Fixes the random state for stochastic embedding algorithms.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

kwargs
[dict] Keyword arguments passed to the base class and may influence the feature visualization properties.

Notes

Currently the only two embeddings supported are MDS and TSNE. Soon to follow will be PCoA and a customized version of PCoA for LDA. The only supported scoring metric is membership, but in the future, silhouette scores and cluster diameter will be added.

In terms of algorithm support, right now any clustering algorithm that has a learned `cluster_centers_` and `labels_` attribute will work with the visualizer. In the future, we will update this to work with hierarchical clusterers that have `n_components` and LDA.

Attributes

*cluster_centers_*
[array of shape (n_clusters, n_features)] Searches for or creates cluster centers for the specified clustering algorithm.

embedded_centers_
[array of shape (n_clusters, 2)] The positions of all the cluster centers on the graph.

scores_
[array of shape (n_clusters,)] The scores of each cluster that determine its size on the graph.

fit_time_
[Timer] The time it took to fit the clustering model and perform the embedding.

property cluster_centers_

Searches for or creates cluster centers for the specified clustering algorithm. This algorithm ensures that that the centers are appropriately drawn and scaled so that distance between clusters are maintained.

draw()

Draw the embedded centers with their sizes on the visualization.

finalize()

Finalize the visualization to create an "origin grid" feel instead of the default matplotlib feel. Set the title, remove spines, and label the grid with components. This function also adds a legend from the sizes if required.

fit(*X*, *y=None*)

Fit the clustering model, computing the centers then embeds the centers into 2D space using the embedding method specified.

property lax

Returns the legend axes, creating it only on demand by creating a 2" by 2" inset axes that has no grid, ticks, spines or face frame (e.g is mostly invisible). The legend can then be drawn on this axes.

property transformer

Creates the internal transformer that maps the cluster center's high dimensional space to its two dimensional space.

yellowbrick.cluster.icdm.**intercluster_distance**(*estimator*, *X*, *y=None*, *ax=None*, *min_size=400*, *max_size=25000*, *embedding='mds'*, *scoring='membership'*, *legend=True*, *legend_loc='lower left'*, *legend_size=1.5*, *random_state=None*, *is_fitted='auto'*, *show=True*, ***kwargs*)

Quick Method: Intercluster distance maps display an embedding of the cluster centers in 2 dimensions with the distance to other centers preserved. E.g. the closer to centers are in the visualization, the closer they are in the original feature space. The clusters are sized according to a scoring metric. By default, they are sized by membership, e.g. the number of instances that belong to each center. This gives a sense of the relative importance of clusters. Note however, that because two clusters overlap in the 2D space, it does not imply that they overlap in the original feature space.

Parameters

estimator
[a Scikit-Learn clusterer] Should be an instance of a centroidal clustering algorithm (or a hierarchical algorithm with a specified number of clusters). Also accepts some other models like LDA for text clustering. If it is not a clusterer, an exception is raised. If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X
[array-like of shape (n, m)] A matrix or data frame with n instances and m features

y
[array-like of shape (n,), optional] A vector or series representing the target for each instance

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).

min_size
[int, default: 400] The size, in points, of the smallest cluster drawn on the graph. Cluster sizes will be scaled between the min and max sizes.

max_size
[int, default: 25000] The size, in points, of the largest cluster drawn on the graph. Cluster sizes will be scaled between the min and max sizes.

embedding
[default: 'mds'] The algorithm used to embed the cluster centers in 2 dimensional space so that the distance between clusters is represented equivalently to their relationship in feature spaceself. Embedding algorithm options include:

- **mds**: multidimensional scaling
- **tsne**: stochastic neighbor embedding

scoring
[default: 'membership'] The scoring method used to determine the size of the clusters drawn on the graph so that the relative importance of clusters can be viewed. Scoring method options include:

- **membership**: number of instances belonging to each cluster

legend
[bool, default: True] Whether or not to draw the size legend onto the graph, omit the legend to more easily see clusters that overlap.

legend_loc
[str, default: "lower left"] The location of the legend on the graph, used to move the legend out of the way of clusters into open space. The same legend location options for matplotlib are used here.

See also:

https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.legend

legend_size
[float, default: 1.5] The size, in inches, of the size legend to inset into the graph.

random_state
[int or RandomState, default: None] Fixes the random state for stochastic embedding algorithms.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments passed to the base class and may influence the feature visualization properties.

Returns

viz
[InterclusterDistance] The intercluster distance visualizer, fitted and finalized.

8.3.8 Model Selection Visualizers

Yellowbrick visualizers are intended to steer the model selection process. Generally, model selection is a search problem defined as follows: given N instances described by numeric properties and (optionally) a target for estimation, find a model described by a triple composed of features, an algorithm and hyperparameters that best fits the data. For most purposes the "best" triple refers to the triple that receives the best cross-validated score for the model type.

The `yellowbrick.model_selection` package provides visualizers for inspecting the performance of cross validation and hyper parameter tuning. Many visualizers wrap functionality found in `sklearn.model_selection` and others build upon it for performing multi-model comparisons.

The currently implemented model selection visualizers are as follows:

- *Validation Curve*: visualizes how the adjustment of a hyperparameter influences training and test scores to tune the bias/variance trade-off.
- *Learning Curve*: shows how the size of training data influences the model to diagnose if a model suffers more from variance error vs. bias error.
- *Cross Validation Scores*: displays cross-validated scores as a bar chart with average as a horizontal line.

- *Feature Importances*: rank features by relative importance in a model
- *Recursive Feature Elimination*: select a subset of features by importance
- *Feature Dropping Curve*: select subsets of features randomly

Model selection makes heavy use of cross validation to measure the performance of an estimator. Cross validation splits a dataset into a training data set and a test data set; the model is fit on the training data and evaluated on the test data. This helps avoid a common pitfall, overfitting, where the model simply memorizes the training data and does not generalize well to new or unknown input.

There are many ways to define how to split a dataset for cross validation. For more information on how scikit-learn implements these mechanisms, please review Cross-validation: evaluating estimator performance in the scikit-learn documentation.

Validation Curve

Model validation is used to determine how effective an estimator is on data that it has been trained on as well as how generalizable it is to new input. To measure a model's performance we first split the dataset into training and test splits, fitting the model on the training data and scoring it on the reserved test data.

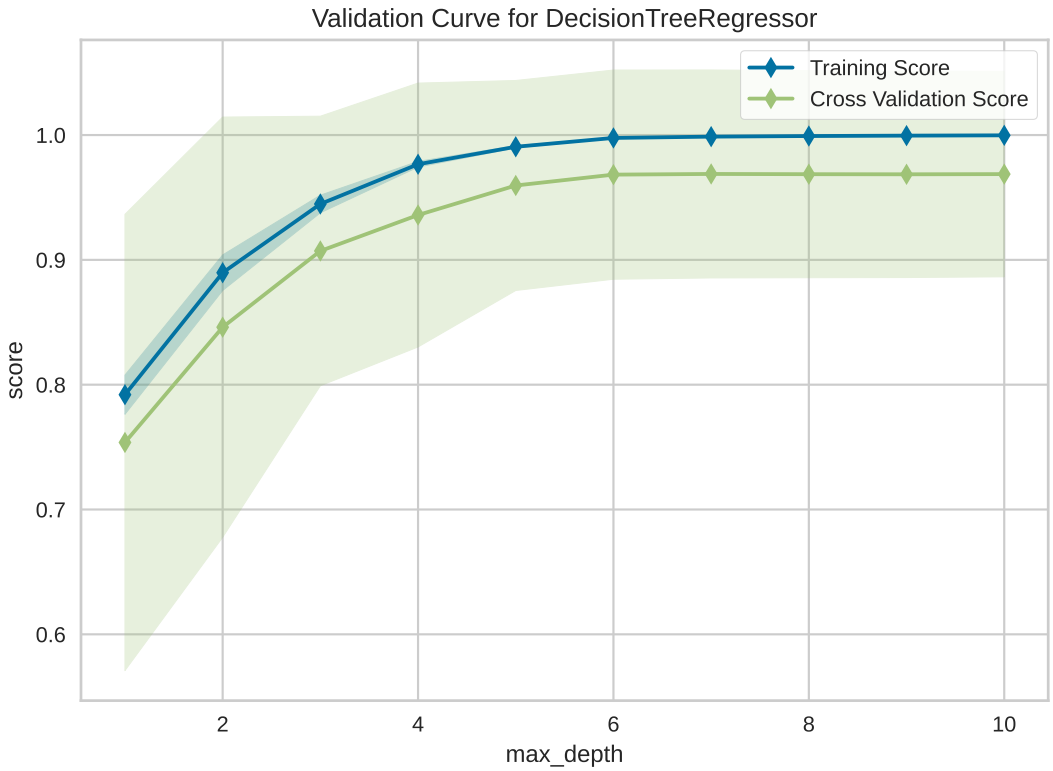
In order to maximize the score, the hyperparameters of the model must be selected which best allow the model to operate in the specified feature space. Most models have multiple hyperparameters and the best way to choose a combination of those parameters is with a grid search. However, it is sometimes useful to plot the influence of a single hyperparameter on the training and test data to determine if the estimator is underfitting or overfitting for some hyperparameter values.

Visualizer	*ValidationCurve*
Quick Method	*validation_curve()*
Models	Classification and Regression
Workflow	Model Selection

In our first example, we'll explore using the `ValidationCurve` visualizer with a regression dataset and in the second, a classification dataset. Note that any estimator that implements `fit()` and `predict()` and has an appropriate scoring mechanism can be used with this visualizer.

```
import numpy as np

from yellowbrick.datasets import load_energy
from yellowbrick.model_selection import ValidationCurve

from sklearn.tree import DecisionTreeRegressor

# Load a regression dataset
X, y = load_energy()

viz = ValidationCurve(
    DecisionTreeRegressor(), param_name="max_depth",
    param_range=np.arange(1, 11), cv=10, scoring="r2"
)

# Fit and show the visualizer
viz.fit(X, y)
viz.show()
```

To further customize this plot, the visualizer also supports a `markers` parameter that changes the marker style.



After loading and wrangling the data, we initialize the `ValidationCurve` with a `DecisionTreeRegressor`. Decision trees become more overfit the deeper they are because at each level of the tree the partitions are dealing with a smaller subset of data. One way to deal with this overfitting process is to limit the depth of the tree. The validation curve explores the relationship of the `"max_depth"` parameter to the R2 score with 10 shuffle split cross-validation. The `param_range` argument specifies the values of `max_depth`, here from 1 to 10 inclusive.

We can see in the resulting visualization that a depth limit of less than 5 levels severely underfits the model on this data set because the training score and testing score climb together in this parameter range, and because of the high variability of cross validation on the test scores. After a depth of 7, the training and test scores diverge, this is because deeper trees are beginning to overfit the training data, providing no generalizability to the model. However, because the cross validation score does not necessarily decrease, the model is not suffering from high error due to variance.

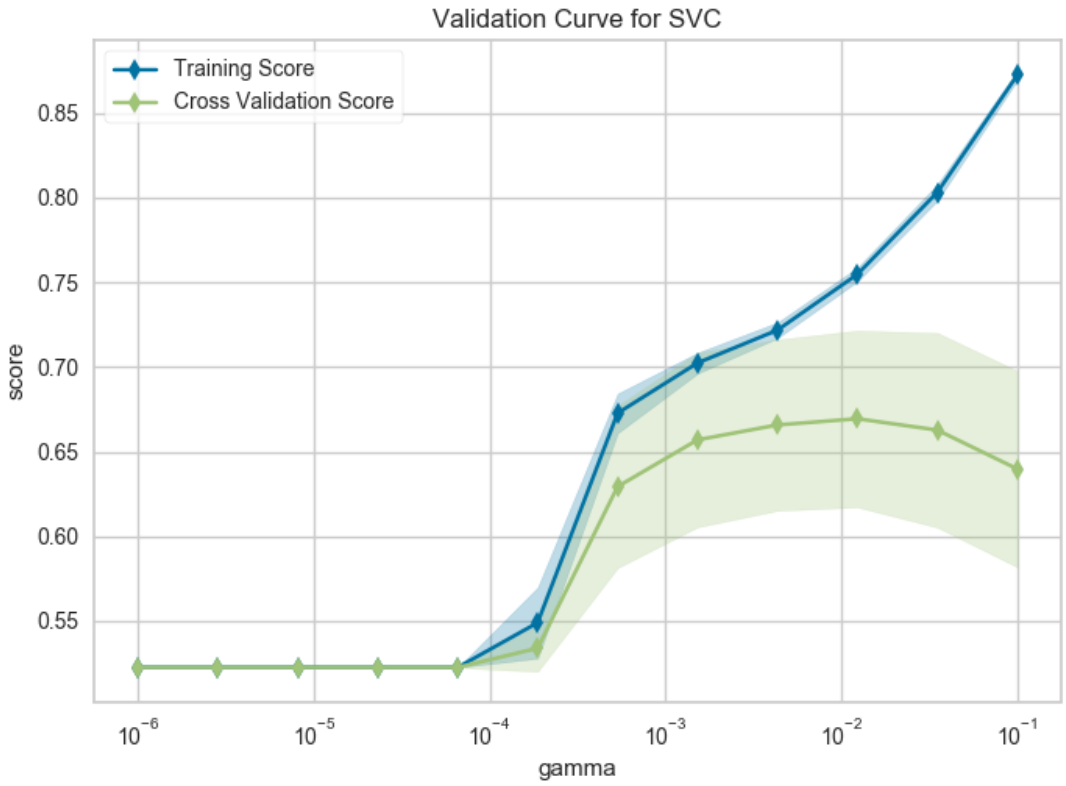
In the next visualizer, we will see an example that more dramatically visualizes the bias/variance tradeoff.

```
from sklearn.svm import SVC
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import StratifiedKFold

# Load a classification data set
X, y = load_game()

# Encode the categorical data with one-hot encoding
X = OneHotEncoder().fit_transform(X)

# Create the validation curve visualizer
cv = StratifiedKFold(12)
```

(continues on next page)

(continued from previous page)

```
param_range = np.logspace(-6, -1, 12)

viz = ValidationCurve(
    SVC(), param_name="gamma", param_range=param_range,
    logx=True, cv=cv, scoring="f1_weighted", n_jobs=8,
)

viz.fit(X, y)
viz.show()
```



After loading data and one-hot encoding it using the Pandas `get_dummies` function, we create a stratified k-folds cross-validation strategy. The hyperparameter of interest is the gamma of a support vector classifier, the coefficient of the RBF kernel. Gamma controls how much influence a single example has, the larger gamma is, the tighter the support vector is around single points (overfitting the model).

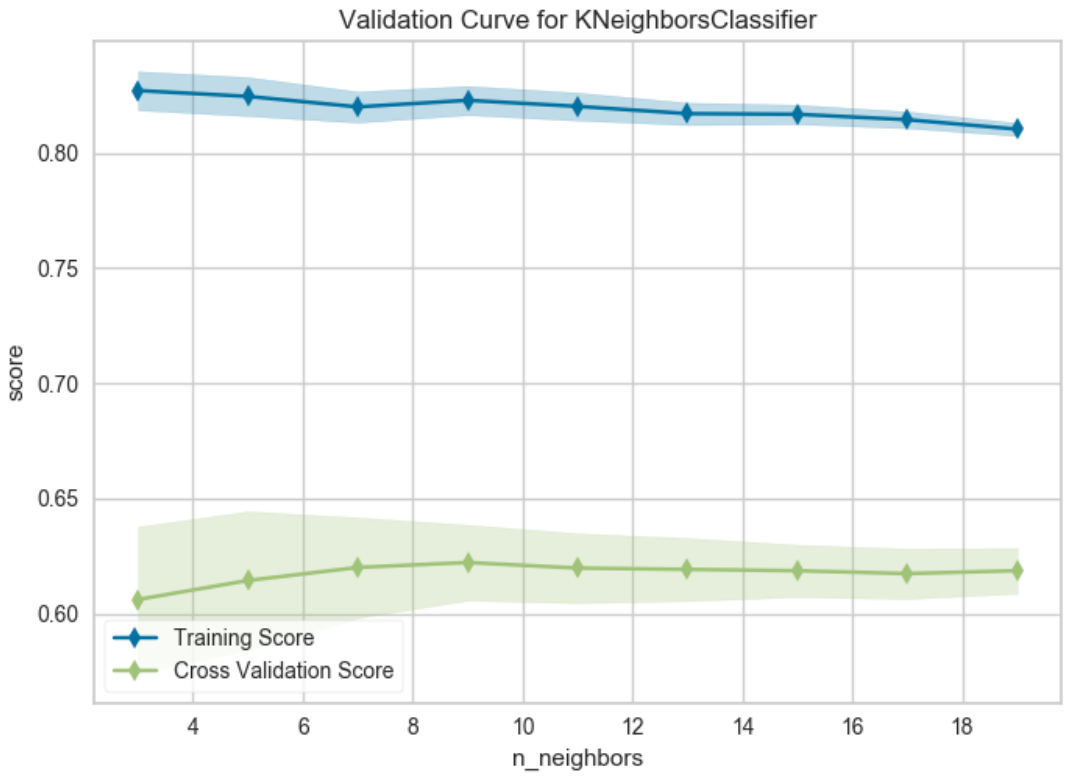
In this visualization we see a definite inflection point around `gamma=0.1`. At this point the training score climbs rapidly as the SVC memorizes the data, while the cross-validation score begins to decrease as the model cannot generalize to unseen data.

Warning: Note that running this and the next example may take a long time. Even with parallelism using n_jobs=8, it can take several hours to go through all the combinations. Reducing the parameter range and minimizing the amount of cross-validation can speed up the validation curve visualization.

Validation curves can be performance intensive since they are training `n_params * n_splits` models and scoring

them. It is critically important to ensure that the specified hyperparameter range is correct, as we will see in the next example.

```
from sklearn.neighbors import KNeighborsClassifier

cv = StratifiedKFold(4)
param_range = np.arange(3, 20, 2)

oz = ValidationCurve(
    KNeighborsClassifier(), param_name="n_neighbors",
    param_range=param_range, cv=cv, scoring="f1_weighted", n_jobs=4,
)

# Using the same game dataset as in the SVC example
oz.fit(X, y)
oz.show()
```



The k nearest neighbors (kNN) model is commonly used when similarity is important to the interpretation of the model. Choosing k is difficult, the higher k is the more data is included in a classification, creating more complex decision topologies, whereas the lower k is, the simpler the model is and the less it may generalize. Using a validation curve seems like an excellent strategy for choosing k, and often it is. However in the example above, all we can see is a decreasing variability in the cross-validated scores.

This validation curve poses two possibilities: first, that we do not have the correct `param_range` to find the best k and need to expand our search to larger values. The second is that other hyperparameters (such as uniform or distance based weighting, or even the distance metric) may have more influence on the default model than k by itself does. Although validation curves can give us some intuition about the performance of a model to a single hyperparameter, grid search

is required to understand the performance of a model with respect to multiple hyperparameters.

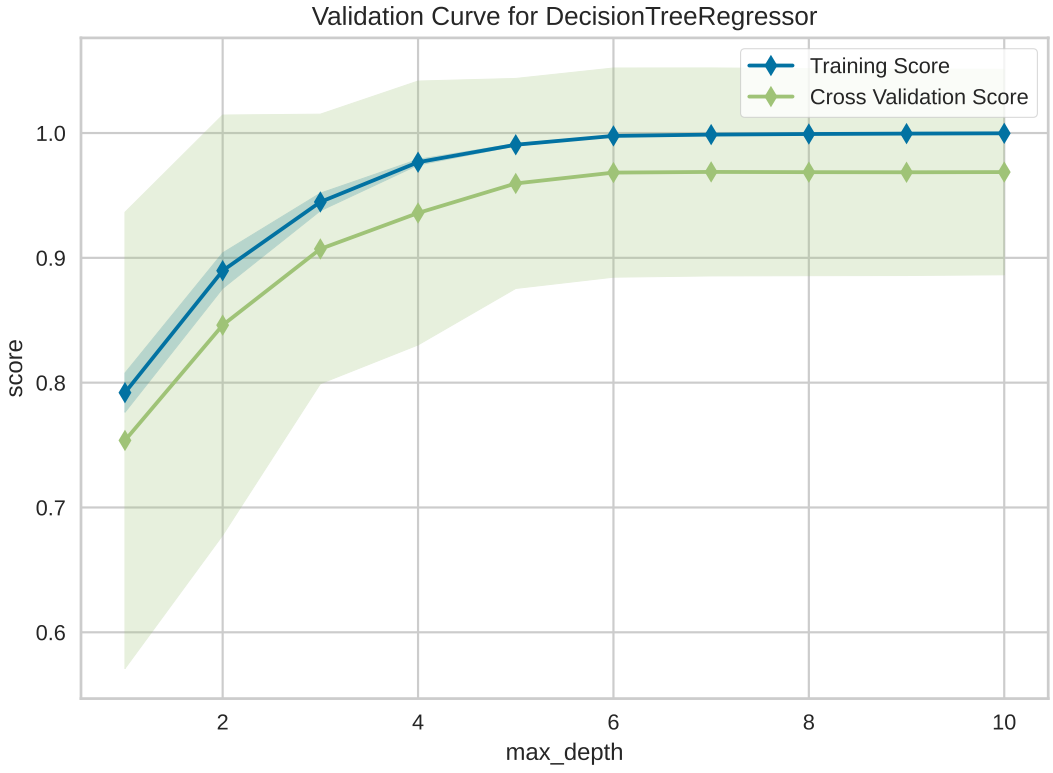
See also:

This visualizer is based on the validation curve described in the scikit-learn documentation: Validation Curves. The visualizer wraps the validation_curve function and most of the arguments are passed directly to it.

Quick Method

Similar functionality as above can be achieved in one line using the associated quick method, `validation_curve`. This method will instantiate and fit a `ValidationCurve` visualizer.

```
import numpy as np

from yellowbrick.datasets import load_energy
from yellowbrick.model_selection import validation_curve

from sklearn.tree import DecisionTreeRegressor

# Load a regression dataset
X, y = load_energy()

viz = validation_curve(
    DecisionTreeRegressor(), X, y, param_name="max_depth",
    param_range=np.arange(1, 11), cv=10, scoring="r2",
)
```



API Reference

Implements a visual validation curve for a hyperparameter.

class yellowbrick.model_selection.validation_curve.**ValidationCurve**(*estimator*, *param_name*, *param_range*, *ax=None*, *logx=False*, *groups=None*, *cv=None*, *scoring=None*, *n_jobs=1*, *pre_dispatch='all'*, *markers='-d'*, ***kwargs*)

Bases: `ModelVisualizer`

Visualizes the validation curve for both test and training data for a range of values for a single hyperparameter of the model. Adjusting the value of a hyperparameter adjusts the complexity of a model. Less complex models suffer from increased error due to bias, while more complex models suffer from increased error due to variance. By inspecting the training and cross-validated test score error, it is possible to estimate a good value for a hyperparameter that balances the bias/variance trade-off.

The visualizer evaluates cross-validated training and test scores for the different hyperparameters supplied. The curve is plotted so that the x-axis is the value of the hyperparameter and the y-axis is the model score. This is similar to a grid search with a single hyperparameter.

The cross-validation generator splits the dataset k times, and scores are averaged over all k runs for the training and test subsets. The curve plots the mean score, and the filled in area suggests the variability of cross-validation by plotting one standard deviation above and below the mean for each split.

Parameters

estimator
[a scikit-learn estimator] An object that implements `fit` and `predict`, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric.

Note that the object is cloned for each validation.

param_name
[string] Name of the parameter that will be varied.

param_range
[array-like, shape (n_values,)] The values of the parameter that will be evaluated.

ax
[matplotlib.Axes object, optional] The axes object to plot the figure on.

logx
[boolean, optional] If True, plots the x-axis with a logarithmic scale.

groups
[array-like, with shape (n_samples,)] Optional group labels for the samples used while splitting the dataset into train/test sets.

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

scoring
[string, callable or None, optional, default: None] A string or scorer callable object / function with signature `scorer(estimator, X, y)`. See scikit-learn model evaluation documentation for names of possible metrics.

n_jobs
[integer, optional] Number of jobs to run in parallel (default 1).

pre_dispatch
[integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like '2*n_jobs'.

markers
[string, default: '-d'] Matplotlib style markers for points on the plot points Options: '-,', '-+', '-o', '-*', '-v', '-h', '-d'

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

This visualizer is essentially a wrapper for the `sklearn.model_selection.learning_curve utility`, discussed in the validation curves documentation.

See also:

The documentation for the learning_curve function, which this visualizer wraps.

Examples

```
>>> import numpy as np
>>> from yellowbrick.model_selection import ValidationCurve
>>> from sklearn.svm import SVC
>>> pr = np.logspace(-6,-1,5)
>>> model = ValidationCurve(SVC(), param_name="gamma", param_range=pr)
>>> model.fit(X, y)
>>> model.show()
```

Attributes

train_scores_
[array, shape (n_ticks, n_cv_folds)] Scores on training sets.

train_scores_mean_
[array, shape (n_ticks,)] Mean training data scores for each training split

train_scores_std_
[array, shape (n_ticks,)] Standard deviation of training data scores for each training split

test_scores_
[array, shape (n_ticks, n_cv_folds)] Scores on test set.

test_scores_mean_
[array, shape (n_ticks,)] Mean test data scores for each test split

test_scores_std_
[array, shape (n_ticks,)] Standard deviation of test data scores for each test split

`draw`(***kwargs*)

Renders the training and test curves.

`finalize`(***kwargs*)

Add the title, legend, and other visual final touches to the plot.

`fit`(*X*, *y=None*)

Fits the validation curve with the wrapped estimator and parameter array to the specified data. Draws training and test score curves and saves the scores to the visualizer.

Parameters

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y
[array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

self
[instance] Returns the instance of the validation curve visualizer for use in pipelines and other sequential transformers.

`yellowbrick.model_selection.validation_curve.validation_curve`(*estimator*, *X*, *y*, *param_name*, *param_range*, *ax=None*, *logx=False*, *groups=None*, *cv=None*, *scoring=None*, *n_jobs=1*, *pre_dispatch='all'*, *show=True*, *markers='-d'*, ***kwargs*)

Displays a validation curve for the specified param and values, plotting both the train and cross-validated test scores. The validation curve is a visual, single-parameter grid search used to tune a model to find the best balance between error due to bias and error due to variance.

This helper function is a wrapper to use the ValidationCurve in a fast, visual analysis.

Parameters

estimator
[a scikit-learn estimator] An object that implements `fit` and `predict`, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric.

Note that the object is cloned for each validation.

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y
[array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

param_name
[string] Name of the parameter that will be varied.

param_range
[array-like, shape (n_values,)] The values of the parameter that will be evaluated.

ax
[matplotlib.Axes object, optional] The axes object to plot the figure on.

logx
[boolean, optional] If True, plots the x-axis with a logarithmic scale.

groups
[array-like, with shape (n_samples,)] Optional group labels for the samples used while splitting the dataset into train/test sets.

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

scoring
[string, callable or None, optional, default: None] A string or scorer callable object / function with signature `scorer(estimator, X, y)`. See scikit-learn model evaluation documentation for names of possible metrics.

n_jobs
[integer, optional] Number of jobs to run in parallel (default 1).

pre_dispatch
[integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like '2*n_jobs'.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

markers
[string, default: '-d'] Matplotlib style markers for points on the plot points Options: '-,', '-+', '-o', '-*', '-v', '-h', '-d'

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers. These arguments are also passed to the `show()` method, e.g. can pass a path to save the figure to.

Returns

visualizer
[ValidationCurve] The fitted visualizer

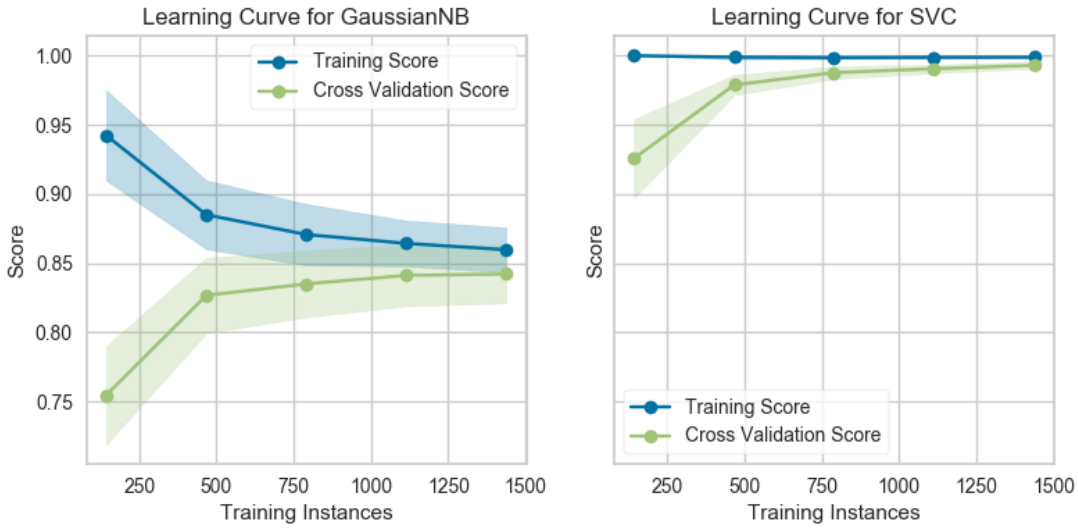
Learning Curve

Visualizer	*LearningCurve*
Quick Method	*learning_curve()*
Models	Classification, Regression, Clustering
Workflow	Model Selection

A learning curve shows the relationship of the training score versus the cross validated test score for an estimator with a varying number of training samples. This visualization is typically used to show two things:

1. How much the estimator benefits from more data (e.g. do we have "enough data" or will the estimator get better if used in an online fashion).
2. If the estimator is more sensitive to error due to variance vs. error due to bias.

Consider the following learning curves (generated with Yellowbrick, but from Plotting Learning Curves in the scikit-learn documentation):



If the training and cross-validation scores converge together as more data is added (shown in the left figure), then the model will probably not benefit from more data. If the training score is much greater than the validation score then the model probably requires more training examples in order to generalize more effectively.

The curves are plotted with the mean scores, however variability during cross-validation is shown with the shaded areas that represent a standard deviation above and below the mean for all cross-validations. If the model suffers from error due to bias, then there will likely be more variability around the training score curve. If the model suffers from error due to variance, then there will be more variability around the cross validated score.

Note: Learning curves can be generated for all estimators that have `fit()` and `predict()` methods as well as a single scoring metric. This includes classifiers, regressors, and clustering as we will see in the following examples.

Classification

In the following example, we show how to visualize the learning curve of a classification model. After loading a `DataFrame` and performing categorical encoding, we create a `StratifiedKFold` cross-validation strategy to ensure all of our classes in each split are represented with the same proportion. We then fit the visualizer using the `f1_weighted` scoring metric as opposed to the default metric, accuracy, to get a better sense of the relationship of precision and recall in our classifier.

```
import numpy as np

from sklearn.model_selection import StratifiedKFold
from sklearn.naive_bayes import MultinomialNB
from sklearn.preprocessing import OneHotEncoder, LabelEncoder

from yellowbrick.datasets import load_game
from yellowbrick.model_selection import LearningCurve

# Load a classification dataset
X, y = load_game()

# Encode the categorical data
X = OneHotEncoder().fit_transform(X)
y = LabelEncoder().fit_transform(y)

# Create the learning curve visualizer
cv = StratifiedKFold(n_splits=12)
sizes = np.linspace(0.3, 1.0, 10)

# Instantiate the classification model and visualizer
model = MultinomialNB()
visualizer = LearningCurve(
    model, cv=cv, scoring='f1_weighted', train_sizes=sizes, n_jobs=4
)

visualizer.fit(X, y)        # Fit the data to the visualizer
visualizer.show()           # Finalize and render the figure
```

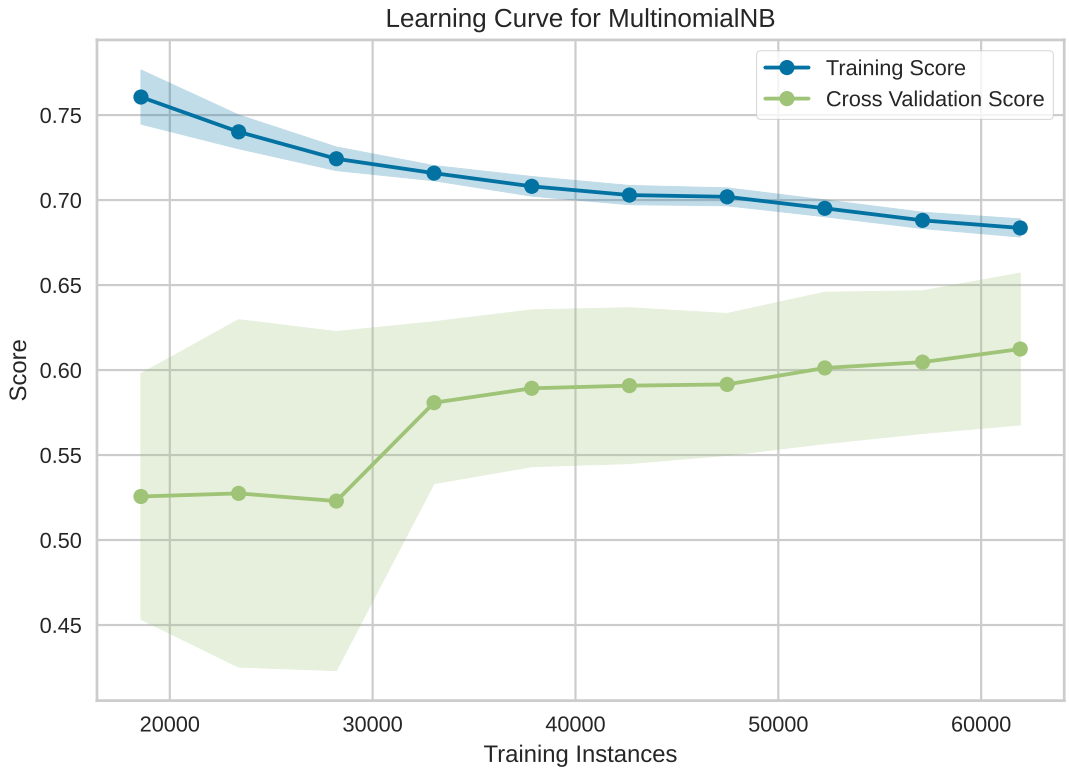
This learning curve shows high test variability and a low score up to around 30,000 instances, however after this level the model begins to converge on an F1 score of around 0.6. We can see that the training and test scores have not yet converged, so potentially this model would benefit from more training data. Finally, this model suffers primarily from error due to variance (the CV scores for the test data are more variable than for training data) so it is possible that the model is overfitting.

Regression

Building a learning curve for a regression is straight forward and very similar. In the below example, after loading our data and selecting our target, we explore the learning curve score according to the coefficient of determination or R2 score.

```
from sklearn.linear_model import RidgeCV

from yellowbrick.datasets import load_energy
```

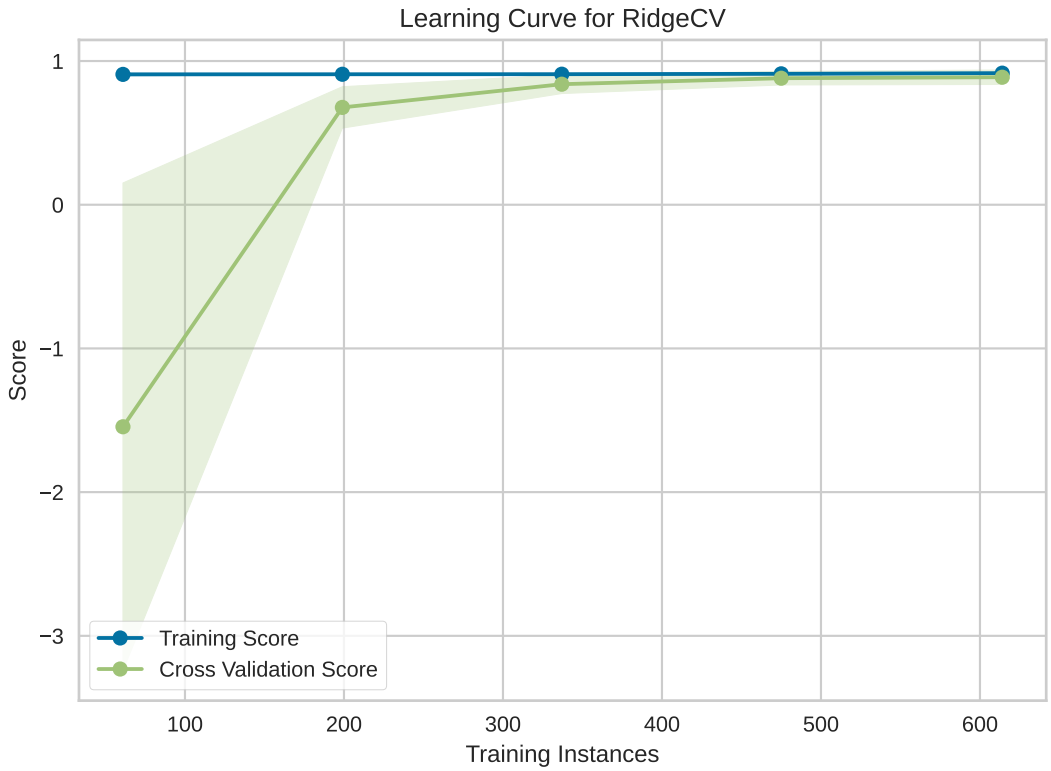
(continues on next page)



(continued from previous page)

```
from yellowbrick.model_selection import LearningCurve

# Load a regression dataset
X, y = load_energy()

# Instantiate the regression model and visualizer
model = RidgeCV()
visualizer = LearningCurve(model, scoring='r2')

visualizer.fit(X, y)        # Fit the data to the visualizer
visualizer.show()           # Finalize and render the figure
```

This learning curve shows a very high variability and much lower score until about 350 instances. It is clear that this model could benefit from more data because it is converging at a very high score. Potentially, with more data and a larger alpha for regularization, this model would become far less variable in the test data.

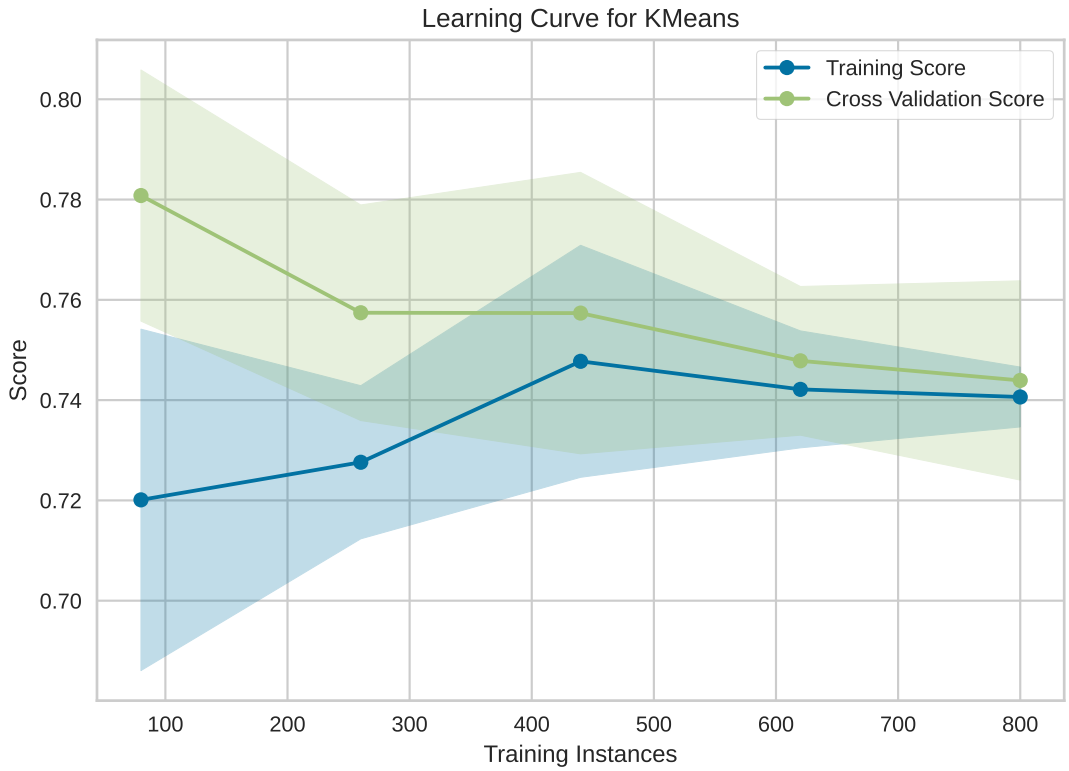


Clustering

Learning curves also work for clustering models and can use metrics that specify the shape or organization of clusters such as silhouette scores or density scores. If the membership is known in advance, then rand scores can be used to compare clustering performance as shown below:

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

from yellowbrick.model_selection import LearningCurve

# Generate synthetic dataset with 5 random clusters
X, y = make_blobs(n_samples=1000, centers=5, random_state=42)

# Instantiate the clustering model and visualizer
model = KMeans()
visualizer = LearningCurve(model, scoring="adjusted_rand_score", random_state=42)

visualizer.fit(X, y)        # Fit the data to the visualizer
visualizer.show()           # Finalize and render the figure
```

Unfortunately, with random data these curves are highly variable, but serve to point out some clustering-specific items. First, note the y-axis is very narrow, roughly speaking these curves are converged and actually the clustering algorithm is performing very well. Second, for clustering, convergence for data points is not necessarily a bad thing; in fact we want to ensure as more data is added, the training and cross-validation scores do not diverge.



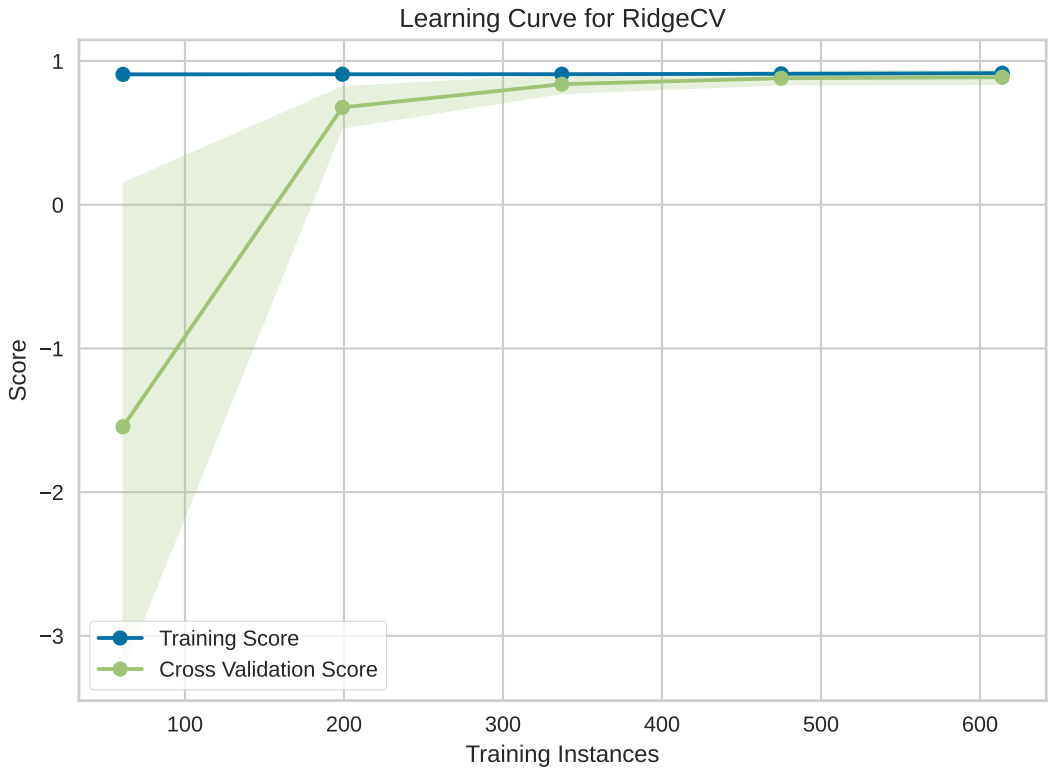
Quick Method

The same functionality can be achieved with the associated quick method `learning_curve`. This method will build the `LearningCurve` object with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from sklearn.linear_model import RidgeCV

from yellowbrick.datasets import load_energy
from yellowbrick.model_selection import learning_curve

# Load a regression dataset
X, y = load_energy()

learning_curve(RidgeCV(), X, y, scoring='r2')
```

See also:

This visualizer is based on the validation curve described in the scikit-learn documentation: Learning Curves. The visualizer wraps the learning_curve function and most of the arguments are passed directly to it.



API Reference

Implements a learning curve visualization for model selection.

class yellowbrick.model_selection.learning_curve.**LearningCurve**(*estimator*, *ax=None*, *groups=None*, *train_sizes=array([0.1, 0.325, 0.55, 0.775, 1.0])*, *cv=None*, *scoring=None*, *exploit_incremental_learning=False*, *n_jobs=1*, *pre_dispatch='all'*, *shuffle=False*, *random_state=None*, ***kwargs*)

Bases: `ModelVisualizer`

Visualizes the learning curve for both test and training data for different training set sizes. These curves can act as a proxy to demonstrate the implied learning rate with experience (e.g. how much data is required to make an adequate model). They also demonstrate if the model is more sensitive to error due to bias vs. error due to variance and can be used to quickly check if a model is overfitting.

The visualizer evaluates cross-validated training and test scores for different training set sizes. These curves are plotted so that the x-axis is the training set size and the y-axis is the score.

The cross-validation generator splits the whole dataset k times, scores are averaged over all k runs for the training subset. The curve plots the mean score for the k splits, and the filled in area suggests the variability of the cross-validation by plotting one standard deviation above and below the mean for each split.

Parameters

estimator
[a scikit-learn estimator] An object that implements `fit` and `predict`, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric.

Note that the object is cloned for each validation.

ax
[matplotlib.Axes object, optional] The axes object to plot the figure on.

groups
[array-like, with shape (n_samples,)] Optional group labels for the samples used while splitting the dataset into train/test sets.

train_sizes
[array-like, shape (n_ticks,)] default: `np.linspace(0.1,1.0,5)`

Relative or absolute numbers of training examples that will be used to generate the learning curve. If the dtype is float, it is regarded as a fraction of the maximum size of the training set, otherwise it is interpreted as absolute sizes of the training sets.

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

scoring
[string, callable or None, optional, default: None] A string or scorer callable object / function with signature `scorer(estimator, X, y)`. See scikit-learn model evaluation documentation for names of possible metrics.

exploit_incremental_learning
[boolean, default: False] If the estimator supports incremental learning, this will be used to speed up fitting for different training set sizes.

n_jobs
[integer, optional] Number of jobs to run in parallel (default 1).

pre_dispatch
[integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like '2*n_jobs'.

shuffle
[boolean, optional] Whether to shuffle training data before taking prefixes of it based on``train_sizes``.

random_state
[int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used when `shuffle` is True.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

This visualizer is essentially a wrapper for the `sklearn.model_selection.learning_curve utility`, discussed in the validation curves documentation.

See also:

The documentation for the learning_curve function, which this visualizer wraps.

Examples

```
>>> from yellowbrick.model_selection import LearningCurve
>>> from sklearn.naive_bayes import GaussianNB
>>> model = LearningCurve(GaussianNB())
>>> model.fit(X, y)
>>> model.show()
```

Attributes

train_sizes_
[array, shape = (n_unique_ticks,), dtype int] Numbers of training examples that has been used to generate the learning curve. Note that the number of ticks might be less than n_ticks because duplicate entries will be removed.

train_scores_
[array, shape (n_ticks, n_cv_folds)] Scores on training sets.

train_scores_mean_
[array, shape (n_ticks,)] Mean training data scores for each training split

train_scores_std_
[array, shape (n_ticks,)] Standard deviation of training data scores for each training split

test_scores_
[array, shape (n_ticks, n_cv_folds)] Scores on test set.

test_scores_mean_
[array, shape (n_ticks,)] Mean test data scores for each test split

test_scores_std_
[array, shape (n_ticks,)] Standard deviation of test data scores for each test split

draw(***kwargs*)

Renders the training and test learning curves.

finalize(***kwargs*)

Add the title, legend, and other visual final touches to the plot.

fit(*X*, *y=None*)

Fits the learning curve with the wrapped model to the specified data. Draws training and test score curves and saves the scores to the estimator.

Parameters

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y
[array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

self
[instance] Returns the instance of the learning curve visualizer for use in pipelines and other sequential transformers.

yellowbrick.model_selection.learning_curve.**learning_curve**(*estimator*, *X*, *y*, *ax=None*, *groups=None*, *train_sizes=array([0.1, 0.325, 0.55, 0.775, 1.0])*, *cv=None*, *scoring=None*, *exploit_incremental_learning=False*, *n_jobs=1*, *pre_dispatch='all'*, *shuffle=False*, *random_state=None*, *show=True*, ***kwargs*)

Displays a learning curve based on number of samples vs training and cross validation scores. The learning curve aims to show how a model learns and improves with experience.

This helper function is a quick wrapper to utilize the LearningCurve for one-off analysis.

Parameters

estimator
[a scikit-learn estimator] An object that implements `fit` and `predict`, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric.

Note that the object is cloned for each validation.

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y
[array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

ax
[matplotlib.Axes object, optional] The axes object to plot the figure on.

groups
[array-like, with shape (n_samples,)] Optional group labels for the samples used while splitting the dataset into train/test sets.

train_sizes
[array-like, shape (n_ticks,)] default: `np.linspace(0.1,1.0,5)`

Relative or absolute numbers of training examples that will be used to generate the learning curve. If the dtype is float, it is regarded as a fraction of the maximum size of the training set, otherwise it is interpreted as absolute sizes of the training sets.

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,

- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

scoring
[string, callable or None, optional, default: None] A string or scorer callable object / function with signature `scorer(estimator, X, y)`. See scikit-learn model evaluation documentation for names of possible metrics.

exploit_incremental_learning
[boolean, default: False] If the estimator supports incremental learning, this will be used to speed up fitting for different training set sizes.

n_jobs
[integer, optional] Number of jobs to run in parallel (default 1).

pre_dispatch
[integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like '2*n_jobs'.

shuffle
[boolean, optional] Whether to shuffle training data before taking prefixes of it based on``train_sizes``.

random_state
[int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used when `shuffle` is True.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers. These arguments are also passed to the *show()* method, e.g. can pass a path to save the figure to.

Returns

visualizer
[LearningCurve] Returns the fitted visualizer.

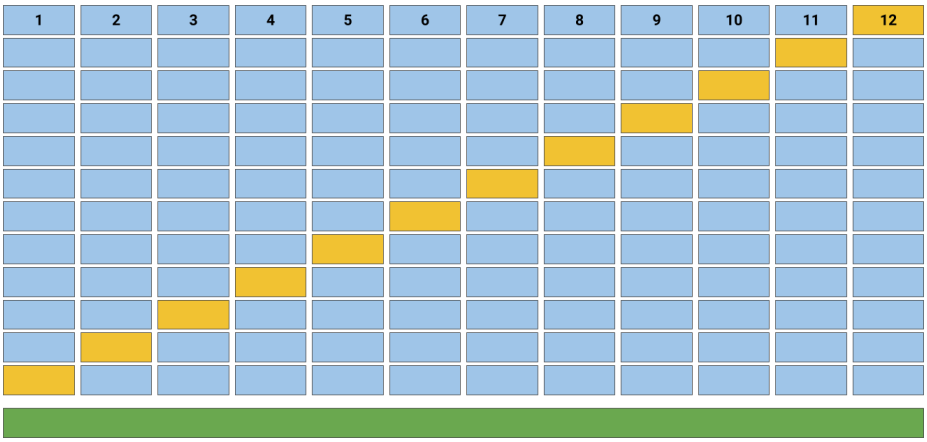
Cross Validation Scores

Visualizer	*CVScores*
Quick Method	*cv_scores()*
Models	Classification, Regression
Workflow	Model Selection

Generally we determine whether a given model is optimal by looking at it's F1, precision, recall, and accuracy (for classification), or it's coefficient of determination (R2) and error (for regression). However, real world data is often distributed somewhat unevenly, meaning that the fitted model is likely to perform better on some sections of the data than on others. Yellowbrick's `CVScores` visualizer enables us to visually explore these variations in performance using different cross validation strategies.

Cross Validation

Cross-validation starts by shuffling the data (to prevent any unintentional ordering errors) and splitting it into k folds. Then k models are fit on $\frac{k-1}{k}$ of the data (called the training split) and evaluated on $\frac{1}{k}$ of the data (called the test split). The results from each evaluation are averaged together for a final score, then the final model is fit on the entire dataset for operationalization.



In Yellowbrick, the `CVScores` visualizer displays cross-validated scores as a bar chart (one bar for each fold) with the average score across all folds plotted as a horizontal dotted line.

Classification

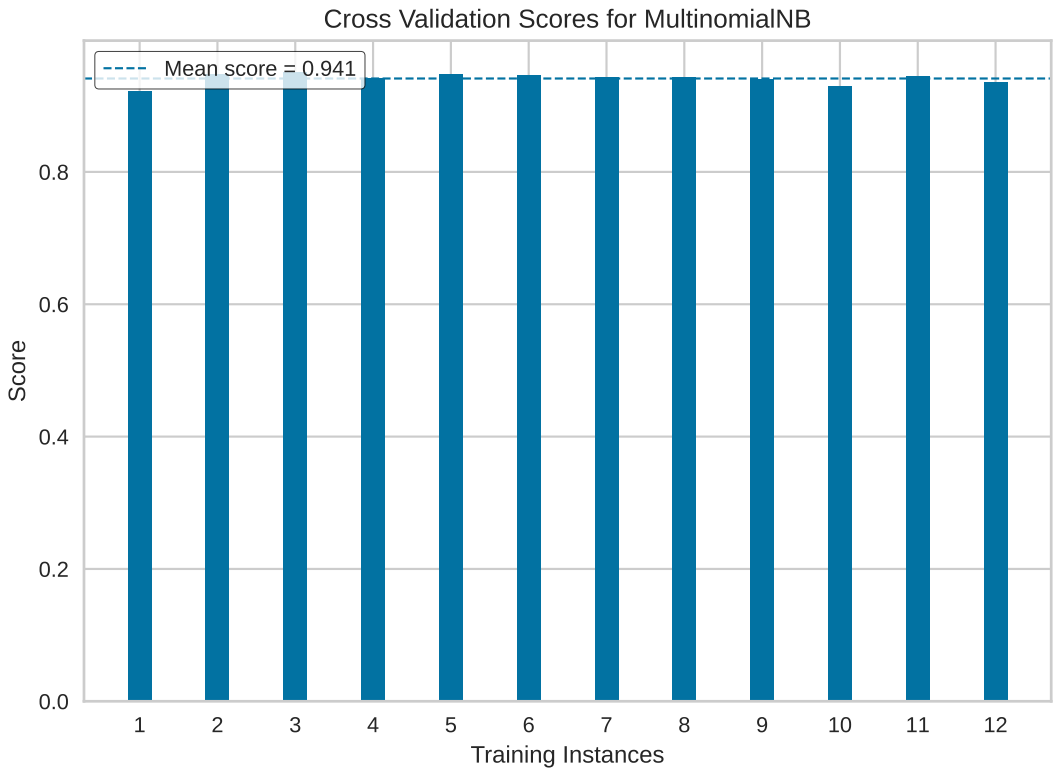
In the following example, we show how to visualize cross-validated scores for a classification model. After loading our occupancy data as a `DataFrame`, we created a `StratifiedKFold` cross-validation strategy to ensure all of our classes in each split are represented with the same proportion. We then fit the `CVScores` visualizer using the `f1_weighted` scoring metric as opposed to the default metric, accuracy, to get a better sense of the relationship of precision and recall in our classifier across all of our folds.

```
from sklearn.model_selection import StratifiedKFold
from sklearn.naive_bayes import MultinomialNB

from yellowbrick.datasets import load_occupancy
from yellowbrick.model_selection import CVScores
```

(continues on next page)

(continued from previous page)

```
# Load the classification dataset
X, y = load_occupancy()

# Create a cross-validation strategy
cv = StratifiedKFold(n_splits=12, shuffle=True, random_state=42)

# Instantiate the classification model and visualizer
model = MultinomialNB()
visualizer = CVScores(model, cv=cv, scoring='f1_weighted')

visualizer.fit(X, y)        # Fit the data to the visualizer
visualizer.show()           # Finalize and render the figure
```

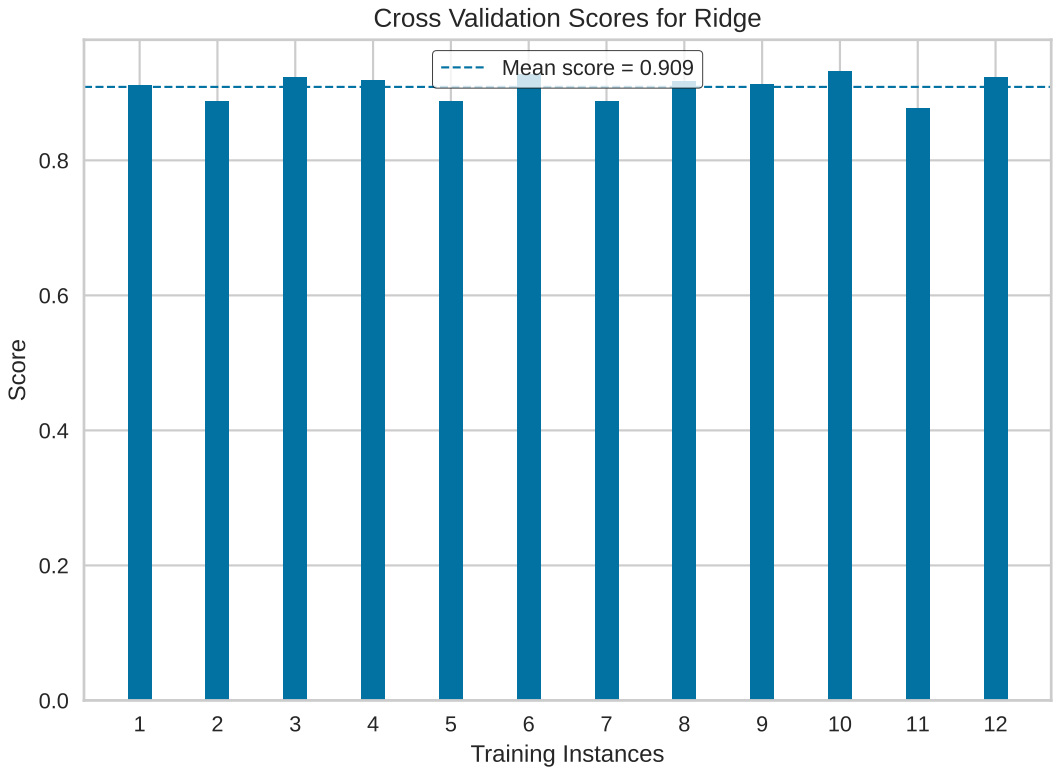


Our resulting visualization shows that while our average cross-validation score is quite high, there are some splits for which our fitted `MultinomialNB` classifier performs significantly less well.

Regression

In this next example we show how to visualize cross-validated scores for a regression model. After loading our energy data as a `DataFrame`, we instantiated a simple `KFold` cross-validation strategy. We then fit the `CVScores` visualizer using the `r2` scoring metric, to get a sense of the coefficient of determination for our regressor across all of our folds.

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import KFold

from yellowbrick.datasets import load_energy
from yellowbrick.model_selection import CVScores

# Load the regression dataset
X, y = load_energy()

# Instantiate the regression model and visualizer
cv = KFold(n_splits=12, shuffle=True, random_state=42)

model = Ridge()
visualizer = CVScores(model, cv=cv, scoring='r2')

visualizer.fit(X, y)        # Fit the data to the visualizer
visualizer.show()           # Finalize and render the figure
```



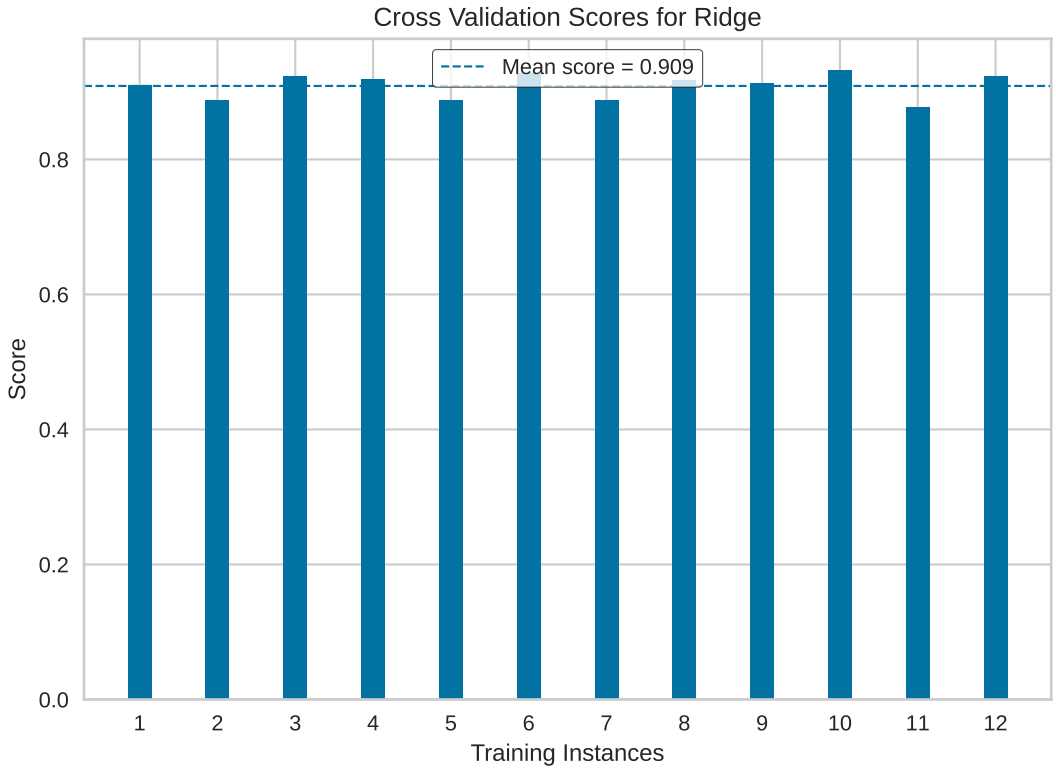
As with our classification `CVScores` visualization, our regression visualization suggests that our `Ridge` regressor performs very well (e.g. produces a high coefficient of determination) across nearly every fold, resulting in another

fairly high overall R2 score.

Quick Method

The same functionality above can be achieved with the associated quick method `cv_scores`. This method will build the `CVScores` object with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import KFold

from yellowbrick.datasets import load_energy
from yellowbrick.model_selection import cv_scores

# Load the regression dataset
X, y = load_energy()

# Instantiate the regression model and visualizer
cv = KFold(n_splits=12, shuffle=True, random_state=42)

model = Ridge()
visualizer = cv_scores(model, X, y, cv=cv, scoring='r2')
```



API Reference

Implements cross-validation score plotting for model selection.

class yellowbrick.model_selection.cross_validation.**CVScores**(*estimator*, *ax=None*, *cv=None*, *scoring=None*, *color=None*, ***kwargs*)

Bases: ModelVisualizer

CVScores displays cross-validated scores as a bar chart, with the average of the scores plotted as a horizontal line.

Parameters

estimator
[a scikit-learn estimator] An object that implements `fit` and `predict`, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric. Note that the object is cloned for each validation.

ax
[matplotlib.Axes object, optional] The axes object to plot the figure on.

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

See the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

scoring
[string, callable or None, optional, default: None] A string or scorer callable object / function with signature `scorer(estimator, X, y)`.

See scikit-learn cross-validation guide for more information on the possible metrics that can be used.

color: string
Specify color for barchart

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Attributes
——-

cv_scores_
[ndarray shape (n_splits,)] The cross-validated scores from each subsection of the data

cv_scores_mean_
[float] Average cross-validated score across all subsections of the data

Notes

This visualizer is a wrapper for sklearn.model_selection.cross_val_score.

Refer to the scikit-learn cross-validation guide for more details.

Examples

```
>>> from sklearn import datasets, svm
>>> iris = datasets.load_iris()
>>> clf = svm.SVC(kernel='linear', C=1)
>>> X = iris.data
>>> y = iris.target
>>> visualizer = CVScores(estimator=clf, cv=5, scoring='f1_macro')
>>> visualizer.fit(X,y)
>>> visualizer.show()
```

draw(***kwargs*)

Creates the bar chart of the cross-validated scores generated from the fit method and places a dashed horizontal line that represents the average value of the scores.

finalize(***kwargs*)

Add the title, legend, and other visual final touches to the plot.

fit(*X*, *y*, ***kwargs*)

Fits the learning curve with the wrapped model to the specified data. Draws training and test score curves and saves the scores to the estimator.

Parameters

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y
[array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

self
[instance]

yellowbrick.model_selection.cross_validation.**cv_scores**(*estimator*, *X*, *y*, *ax=None*, *cv=None*, *scoring=None*, *color=None*, *show=True*, ***kwargs*)

Displays cross validation scores as a bar chart and the average of the scores as a horizontal line

This helper function is a quick wrapper to utilize the CVScores visualizer for one-off analysis.

Parameters

estimator
[a scikit-learn estimator] An object that implements `fit` and `predict`, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric. Note that the object is cloned for each validation.

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y
[array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

ax
[matplotlib.Axes object, optional] The axes object to plot the figure on.

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

see the scikit-learn
`cross-validation guide <https://goo.gl/FS3VU6>`_
for more information on the possible strategies that can be used here.
scoring
[string, callable or None, optional, default: None] A string or scorer callable object / function with signature `scorer(estimator, X, y)`.

See scikit-learn cross-validation guide for more information on the possible metrics that can be used.

color: string
Specify color for barchart

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

visualizer
[CVScores] The fitted visualizer.

Feature Importances

The feature engineering process involves selecting the *minimum* required features to produce a valid model because the more features a model contains, the more complex it is (and the more sparse the data), therefore the more sensitive the model is to errors due to variance. A common approach to eliminating features is to describe their relative importance to a model, then eliminate weak features or combinations of features and re-evalute to see if the model fairs better during cross-validation.

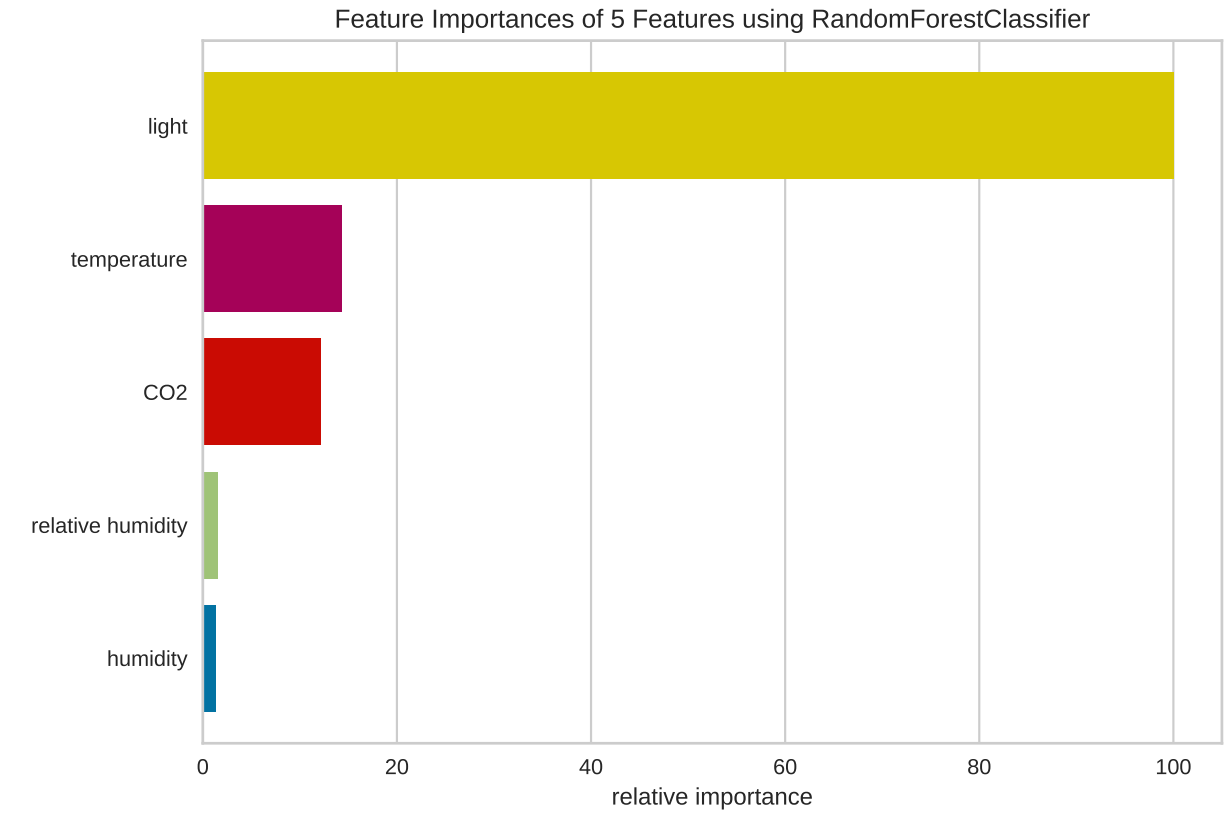
Many model forms describe the underlying impact of features relative to each other. In scikit-learn, Decision Tree models and ensembles of trees such as Random Forest, Gradient Boosting, and Ada Boost provide a `feature_importances_` attribute when fitted. The Yellowbrick `FeatureImportances` visualizer utilizes this attribute to rank and plot relative importances.

Visualizer	*FeatureImportances*
Quick Method	*feature_importances()*
Models	Classification, Regression
Workflow	Model selection, feature selection

Let's start with an example; first load a classification dataset.

Then we can create a new figure (this is optional, if an `Axes` isn't specified, Yellowbrick will use the current figure or create one). We can then fit a `FeatureImportances` visualizer with a `GradientBoostingClassifier` to visualize the ranked features.

```
from sklearn.ensemble import RandomForestClassifier

from yellowbrick.datasets import load_occupancy
from yellowbrick.model_selection import FeatureImportances

# Load the classification data set
X, y = load_occupancy()

model = RandomForestClassifier(n_estimators=10)
viz = FeatureImportances(model)
viz.fit(X, y)
viz.show()
```



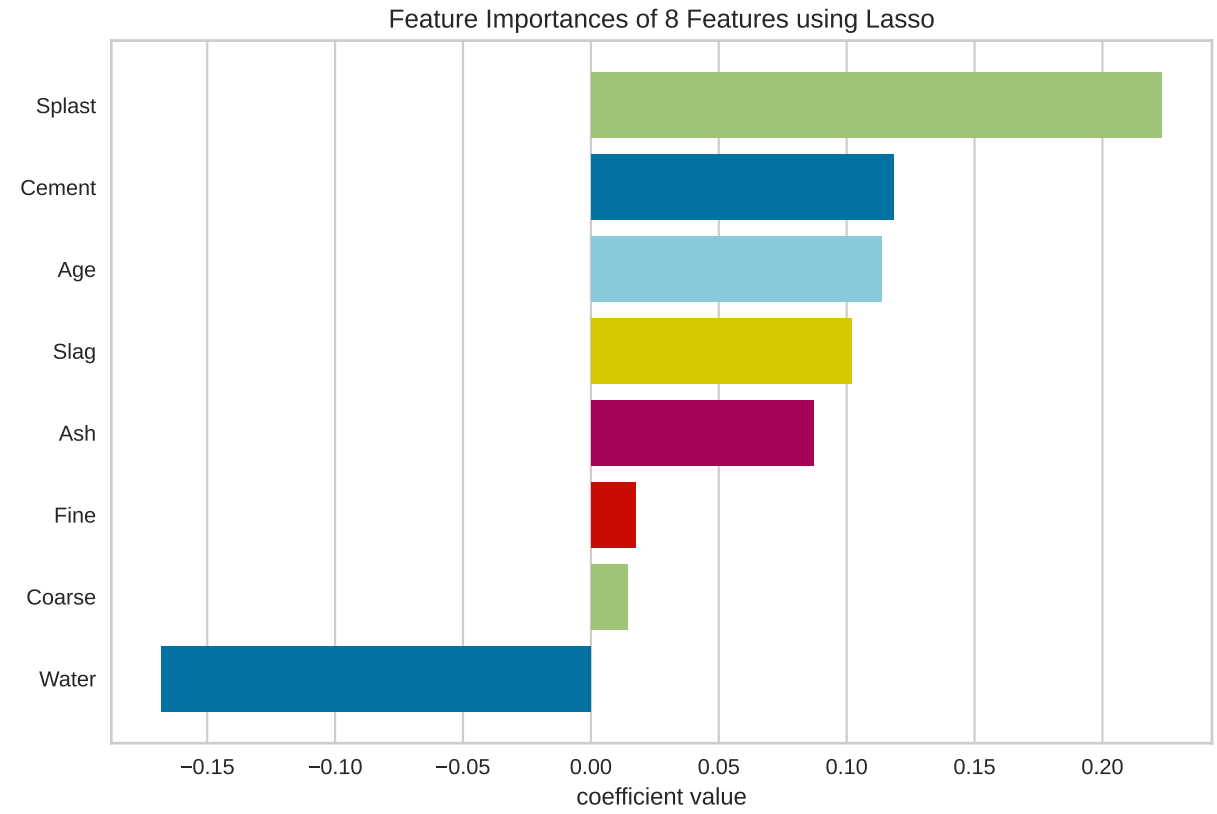
The above figure shows the features ranked according to the explained variance each feature contributes to the model. In this case the features are plotted against their *relative importance*, that is the percent importance of the most important

feature. The visualizer also contains `features_` and `feature_importances_` attributes to get the ranked numeric values.

For models that do not support a `feature_importances_` attribute, the `FeatureImportances` visualizer will also draw a bar plot for the `coef_` attribute that many linear models provide.

When using a model with a `coef_` attribute, it is better to set `relative=False` to draw the true magnitude of the coefficient (which may be negative). We can also specify our own set of labels if the dataset does not have column names or to print better titles. In the example below we title case our features for better readability:

```
from sklearn.linear_model import Lasso
from yellowbrick.datasets import load_concrete
from yellowbrick.model_selection import FeatureImportances

# Load the regression dataset
dataset = load_concrete(return_dataset=True)
X, y = dataset.to_data()

# Title case the feature for better display and create the visualizer
labels = list(map(lambda s: s.title(), dataset.meta['features']))
viz = FeatureImportances(Lasso(), labels=labels, relative=False)

# Fit and show the feature importances
viz.fit(X, y)
viz.show()
```



Note: The interpretation of the importance of coeficients depends on the model; see the discussion below for more details.

Stacked Feature Importances

Some estimators return a multi-dimensonal array for either `feature_importances_` or `coef_` attributes. For example the `LogisticRegression` classifier returns a `coef_` array in the shape of `(n_classes, n_features)` in the multiclass case. These coefficients map the importance of the feature to the prediction of the probability of a specific class. Although the interpretation of multi-dimensional feature importances depends on the specific estimator and model family, the data is treated the same in the `FeatureImportances` visualizer – namely the importances are averaged.

Taking the mean of the importances may be undesirable for several reasons. For example, a feature may be more informative for some classes than others. Multi-output estimators also do not benefit from having averages taken across what are essentially multiple internal models. In this case, use the `stack=True` parameter to draw a stacked bar chart of importances as follows:

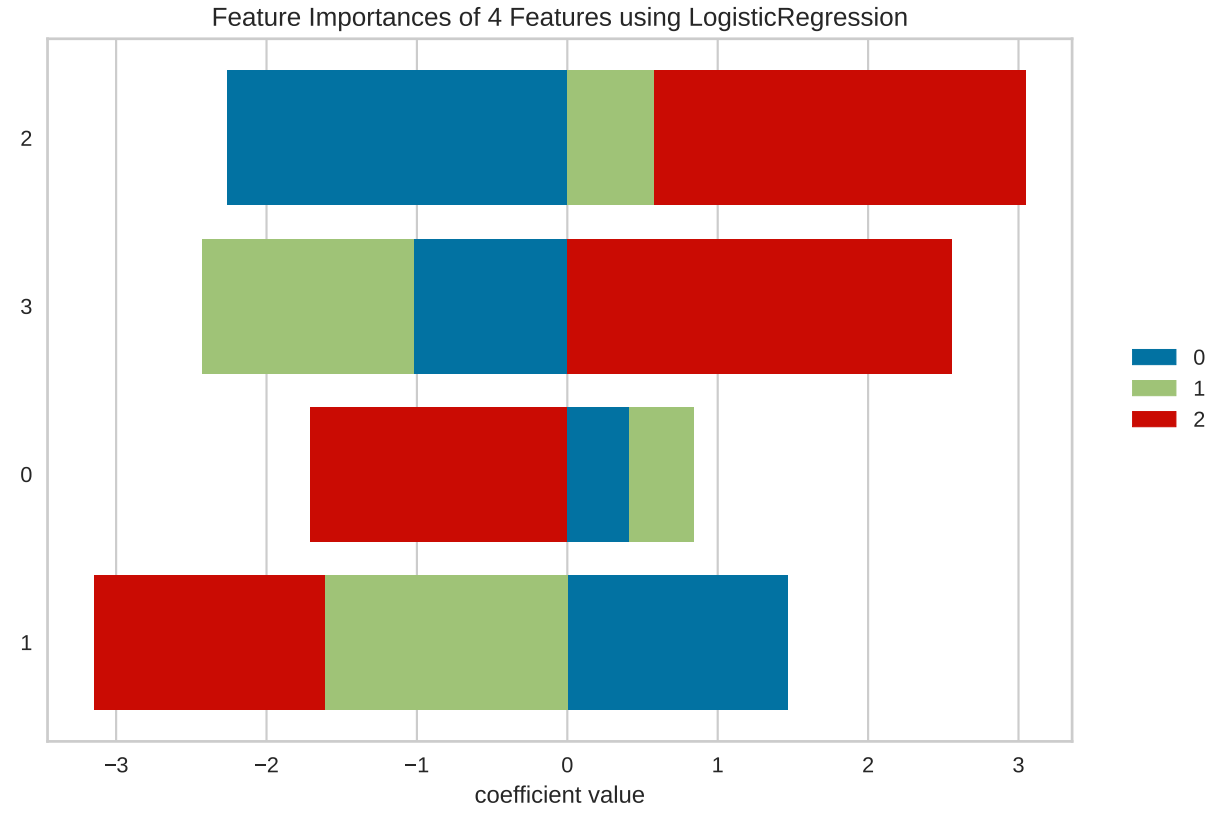
```
from yellowbrick.model_selection import FeatureImportances
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris

data = load_iris()
X, y = data.data, data.target

model = LogisticRegression(multi_class="auto", solver="liblinear")
viz = FeatureImportances(model, stack=True, relative=False)
viz.fit(X, y)
viz.show()
```

Top and Bottom Feature Importances

It may be more illuminating to the feature engineering process to identify the most or least informative features. To view only the N most informative features, specify the `topn` argument to the visualizer. Similar to slicing a ranked list by their importance, if `topn` is a postive integer, then the most highly ranked features are used. If `topn` is a negative integer, then the lowest ranked features are displayed instead.

```
from sklearn.linear_model import Lasso
from yellowbrick.datasets import load_concrete
from yellowbrick.model_selection import FeatureImportances

# Load the regression dataset
dataset = load_concrete(return_dataset=True)
X, y = dataset.to_data()

# Title case the feature for better display and create the visualizer
labels = list(map(lambda s: s.title(), dataset.meta['features']))
viz = FeatureImportances(Lasso(), labels=labels, relative=False, topn=3)

# Fit and show the feature importances
```

(continues on next page)



(continued from previous page)

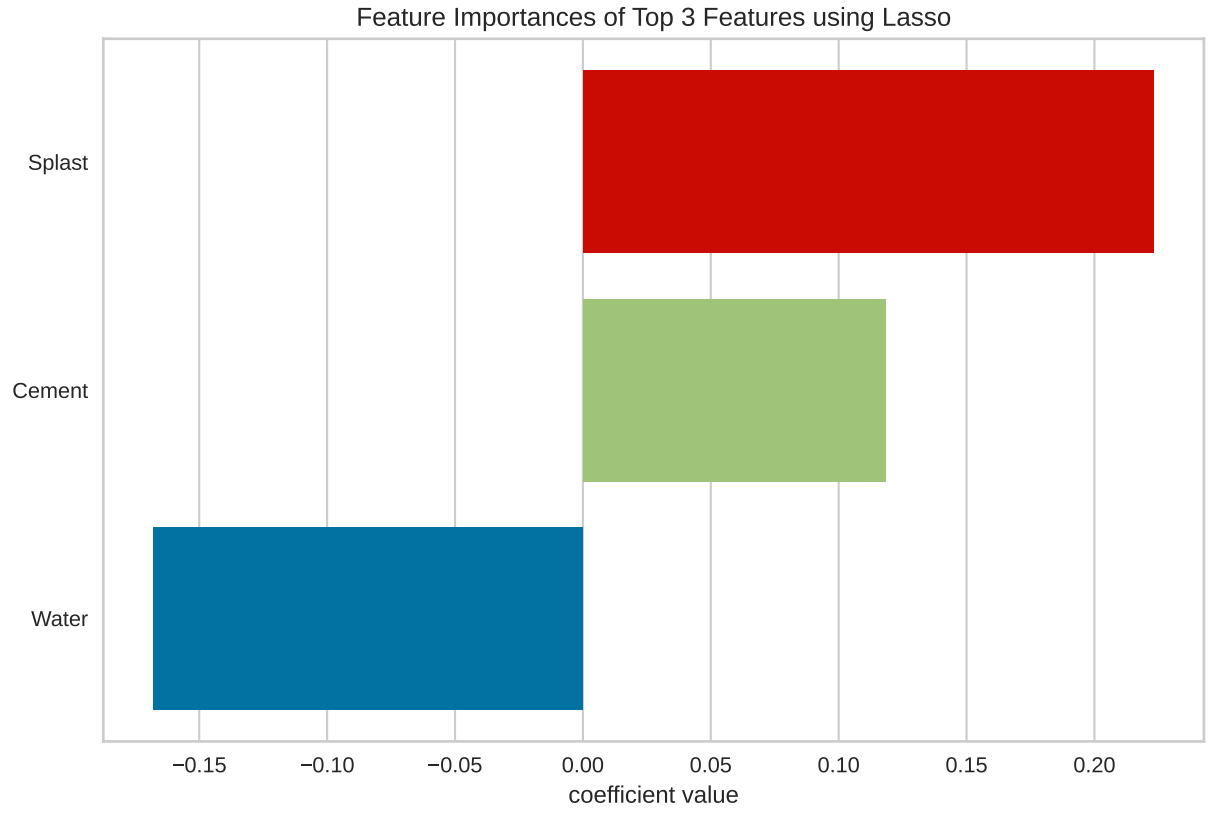
```
viz.fit(X, y)
viz.show()
```

Using `topn=3`, we can identify the three most informative features in the concrete dataset as `splast`, `cement`, and `water`. This approach to visualization may assist with *factor analysis* - the study of how variables contribute to an overall model. Note that although `water` has a negative coefficient, it is the magnitude (absolute value) of the feature that matters since we are closely inspecting the negative correlation of `water` with the strength of concrete. Alternatively, `topn=-3` would reveal the three least informative features in the model. This approach is useful to model tuning similar to *Recursive Feature Elimination*, but instead of automatically removing features, it would allow you to identify the lowest-ranked features as they change in different model instantiations. In either case, if you have many features, using `topn` can significantly increase the visual and analytical capacity of your analysis.

The `topn` parameter can also be used when `stacked=True`. In the context of stacked feature importance graphs, the information of a feature is the width of the entire bar, or the sum of the absolute value of all coefficients contained therein.

```
from yellowbrick.model_selection import FeatureImportances
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris

data = load_iris()
X, y = data.data, data.target

model = LogisticRegression(multi_class="auto", solver="liblinear")
```

(continues on next page)



(continued from previous page)

```
viz = FeatureImportances(model, stack=True, relative=False, topn=-3)
viz.fit(X, y)
viz.show()
```

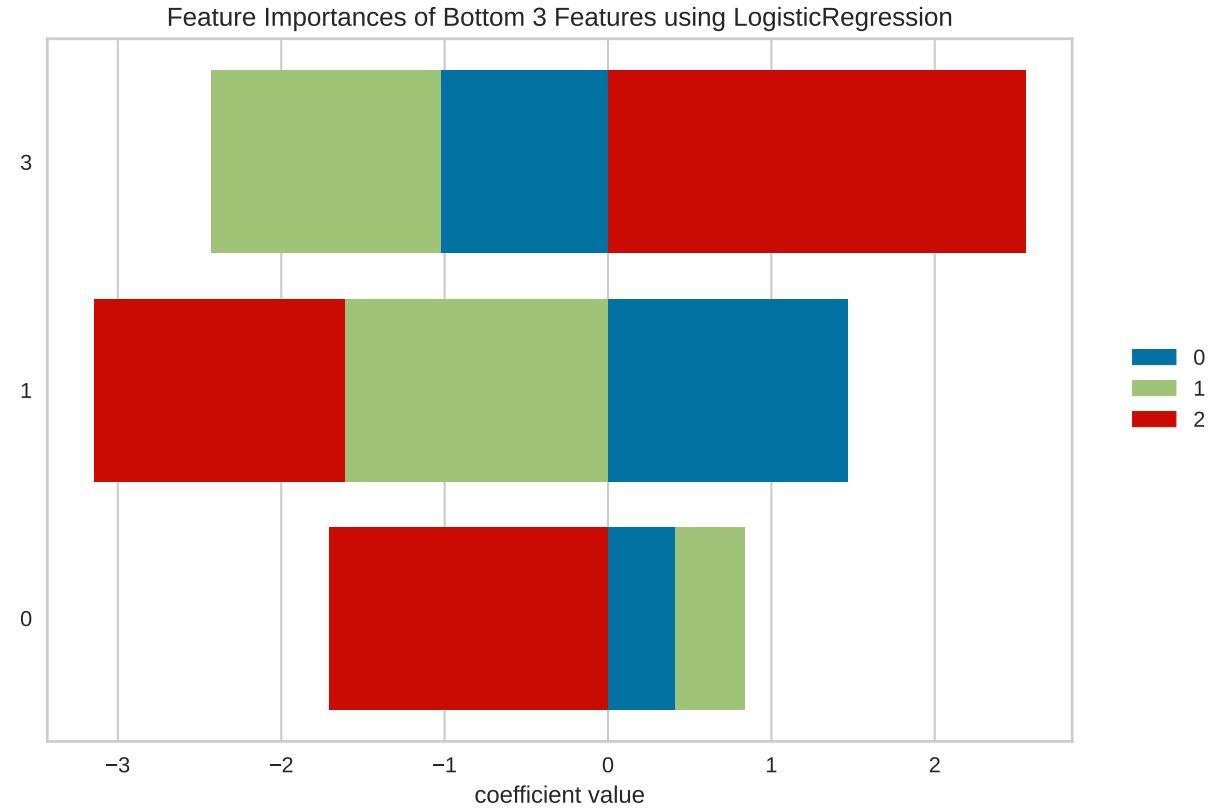
Discussion

Generalized linear models compute a predicted independent variable via the linear combination of an array of coefficients with an array of dependent variables. GLMs are fit by modifying the coefficients so as to minimize error and regularization techniques specify how the model modifies coefficients in relation to each other. As a result, an opportunity presents itself: larger coefficients are necessarily "more informative" because they contribute a greater weight to the final prediction in most cases.

Additionally we may say that instance features may also be more or less "informative" depending on the product of the instance feature value with the feature coefficient. This creates two possibilities:

1. We can compare models based on ranking of coefficients, such that a higher coefficient is "more informative".
2. We can compare instances based on ranking of feature/coefficient products such that a higher product is "more informative".

In both cases, because the coefficient may be negative (indicating a strong negative correlation) we must rank features by the absolute values of their coefficients. Visualizing a model or multiple models by most informative feature is usually done via bar chart where the y-axis is the feature names and the x-axis is numeric value of the coefficient such that the x-axis has both a positive and negative quadrant. The bigger the size of the bar, the more informative that feature is.

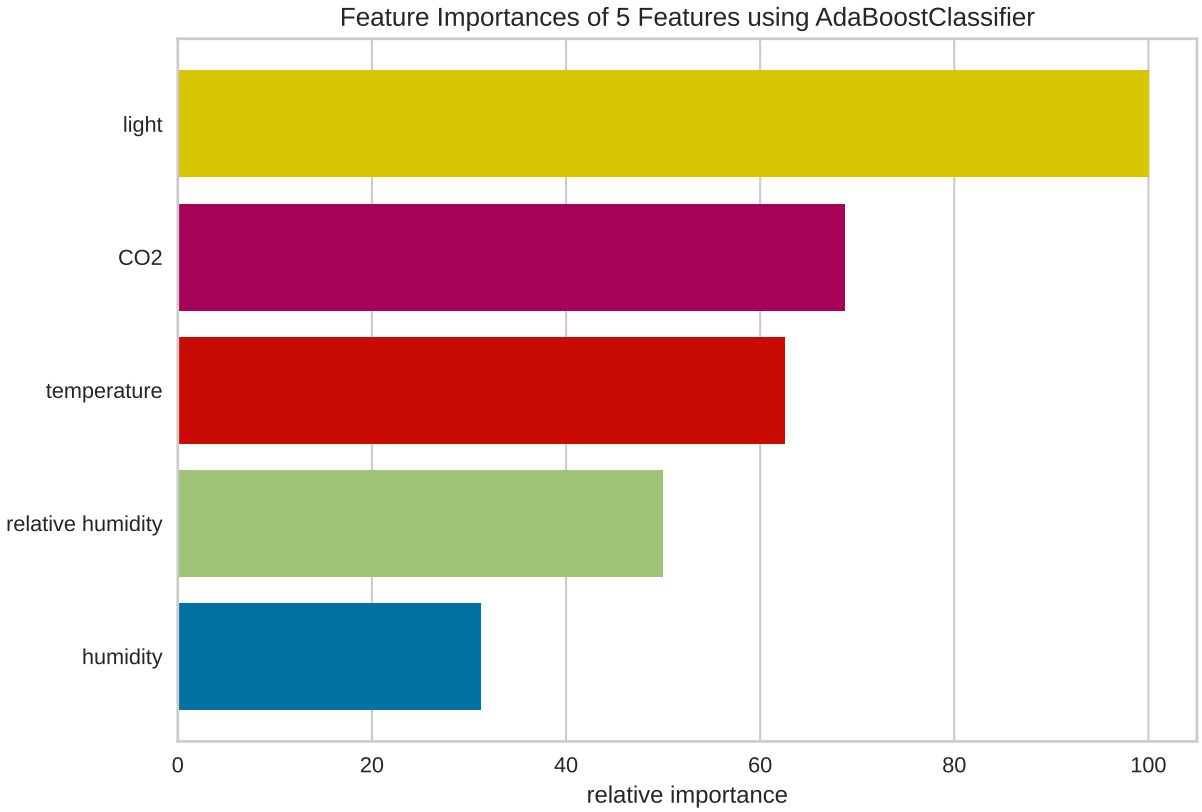


This method may also be used for instances; but generally there are very many instances relative to the number models being compared. Instead a heatmap grid is a better choice to inspect the influence of features on individual instances. Here the grid is constructed such that the x-axis represents individual features, and the y-axis represents individual instances. The color of each cell (an instance, feature pair) represents the magnitude of the product of the instance value with the feature's coefficient for a single model. Visual inspection of this diagnostic may reveal a set of instances for which one feature is more predictive than another; or other types of regions of information in the model itself.

Quick Method

The same functionality above can be achieved with the associated quick method *feature_importances*. This method will build the `FeatureImportances` object with the associated arguments, fit it, then (optionally) immediately show it.

```
from sklearn.ensemble import AdaBoostClassifier
from yellowbrick.datasets import load_occupancy
from yellowbrick.model_selection import feature_importances

# Load dataset
X, y = load_occupancy()

# Use the quick method and immediately show the figure
feature_importances(AdaBoostClassifier(), X, y)
```



API Reference

Implementation of a feature importances visualizer. This visualizer sits in kind of a weird place since it is technically a model scoring visualizer, but is generally used for feature engineering.

class yellowbrick.model_selection.importances.**FeatureImportances**(*estimator*, *ax=None*, *labels=None*, *relative=True*, *absolute=False*, *xlabel=None*, *stack=False*, *colors=None*, *colormap=None*, *is_fitted='auto'*, *topn=None*, ***kwargs*)

Bases: ModelVisualizer

Displays the most informative features in a model by showing a bar chart of features ranked by their importances. Although primarily a feature engineering mechanism, this visualizer requires a model that has either a coef_ or feature_importances_ parameter after fit.

Note: Some classification models such as LogisticRegression, return coef_ as a multidimensional array of shape (n_classes, n_features). In this case, the FeatureImportances visualizer computes the mean of the coefs_ by class for each feature.

Parameters

estimator

[Estimator] A Scikit-Learn estimator that learns feature importances. Must support either coef_ or feature_importances_ parameters. If the estimator is not fitted, it is fit when

the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

labels
[list, default: None] A list of feature names to use. If a DataFrame is passed to fit and features is None, feature names are selected as the column names.

relative
[bool, default: True] If true, the features are described by their relative importance as a percentage of the strongest feature component; otherwise the raw numeric description of the feature importance is shown.

absolute
[bool, default: False] Make all coeficients absolute to more easily compare negative coefficients with positive ones.

xlabel
[str, default: None] The label for the X-axis. If None is automatically determined by the underlying model and options provided.

stack
[bool, default: False] If true and the classifier returns multi-class feature importance, then a stacked bar plot is plotted; otherwise the mean of the feature importance across classes are plotted.

colors: list of strings
Specify colors for each bar in the chart if `stack==False`.

colormap
[string or matplotlib cmap] Specify a colormap to color the classes if `stack==True`.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

topn
[int, default=None] Display only the top N results with a positive integer, or the bottom N results with a negative integer. If None or 0, all results are shown.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> visualizer = FeatureImportances(GradientBoostingClassifier())
>>> visualizer.fit(X, y)
>>> visualizer.show()
```

Attributes

features_
[np.array] The feature labels ranked according to their importance

feature_importances_
[np.array] The numeric value of the feature importance computed by the model

classes_
[np.array] The classes labeled. Is not None only for classifier.

`draw`(***kwargs*)

Draws the feature importances as a bar chart; called from fit.

`finalize`(***kwargs*)

Finalize the drawing setting labels and title.

`fit`(*X*, *y=None*, ***kwargs*)

Fits the estimator to discover the feature importances described by the data, then draws those importances as a bar plot.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

kwargs
[dict] Keyword arguments passed to the fit method of the estimator.

Returns

self
[visualizer] The fit method must always return self to support pipelines.

yellowbrick.model_selection.importances.`feature_importances`(*estimator*, *X*, *y=None*, *ax=None*, *labels=None*, *relative=True*, *absolute=False*, *xlabel=None*, *stack=False*, *colors=None*, *colormap=None*, *is_fitted='auto'*, *topn=None*, *show=True*, ***kwargs*)

Quick Method: Displays the most informative features in a model by showing a bar chart of features ranked by their importances. Although primarily a feature engineering mechanism, this visualizer requires a model that has either a `coef_` or `feature_importances_` parameter after fit.

Parameters

estimator
[Estimator] A Scikit-Learn estimator that learns feature importances. Must support either `coef_` or `feature_importances_` parameters. If the estimator is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n, optional] An array or series of target or class values

ax
[matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).

labels
[list, default: None] A list of feature names to use. If a DataFrame is passed to fit and features is None, feature names are selected as the column names.

relative
[bool, default: True] If true, the features are described by their relative importance as a percentage of the strongest feature component; otherwise the raw numeric description of the feature importance is shown.

absolute
[bool, default: False] Make all coeficients absolute to more easily compare negative coeficients with positive ones.

xlabel
[str, default: None] The label for the X-axis. If None is automatically determined by the underlying model and options provided.

stack
[bool, default: False] If true and the classifier returns multi-class feature importance, then a stacked bar plot is plotted; otherwise the mean of the feature importance across classes are plotted.

colors: list of strings
Specify colors for each bar in the chart if `stack==False`.

colormap
[string or matplotlib cmap] Specify a colormap to color the classes if `stack==True`.

is_fitted
[bool or str, default='auto'] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If 'auto' (default), a helper method will check if the estimator is fitted before fitting it again.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

topn
[int, default=None] Display only the top N results with a positive integer, or the bottom N results with a negative integer. If None or 0, all results are shown.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

viz
[FeatureImportances] The feature importances visualizer, fitted and finalized.

Recursive Feature Elimination

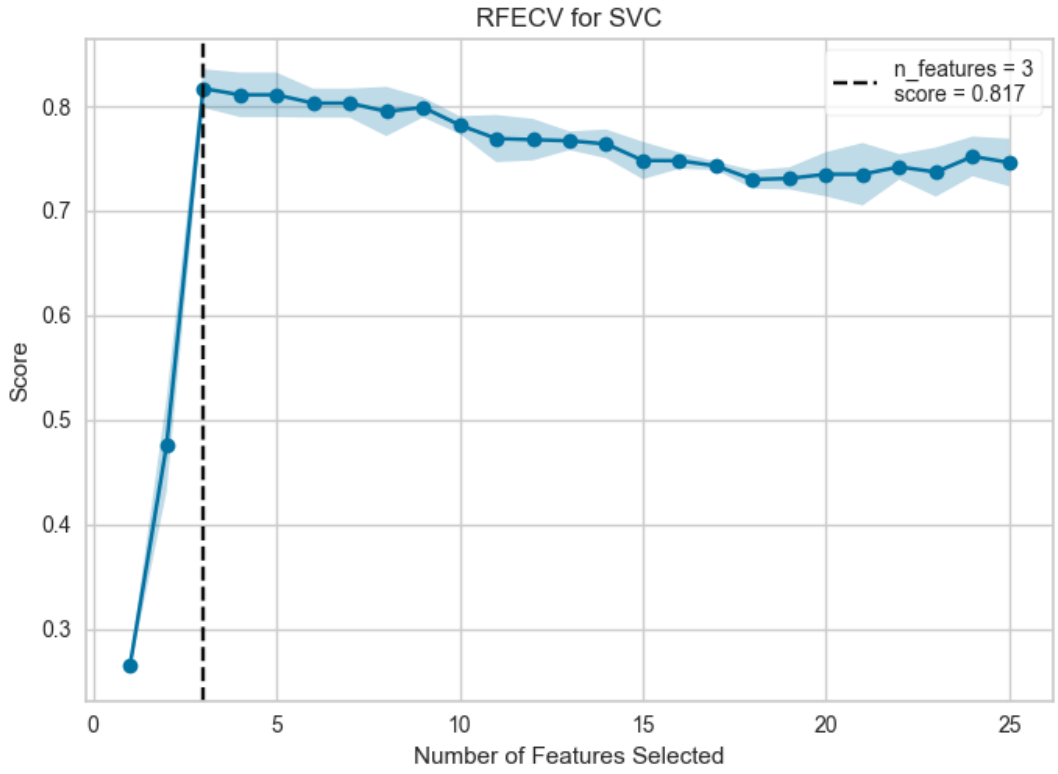
Visualizer	*RFECV*
Quick Method	*rfecv()*
Models	Classification, Regression
Workflow	Model Selection

Recursive feature elimination (RFE) is a feature selection method that fits a model and removes the weakest feature (or features) until the specified number of features is reached. Features are ranked by the model's `coef_` or `feature_importances_` attributes, and by recursively eliminating a small number of features per loop, RFE attempts to eliminate dependencies and collinearity that may exist in the model.

RFE requires a specified number of features to keep, however it is often not known in advance how many features are valid. To find the optimal number of features cross-validation is used with RFE to score different feature subsets and select the best scoring collection of features. The `RFECV` visualizer plots the number of features in the model along with their cross-validated test score and variability and visualizes the selected number of features.

To show how this works in practice, we'll start with a contrived example using a dataset that has only 3 informative features out of 25.

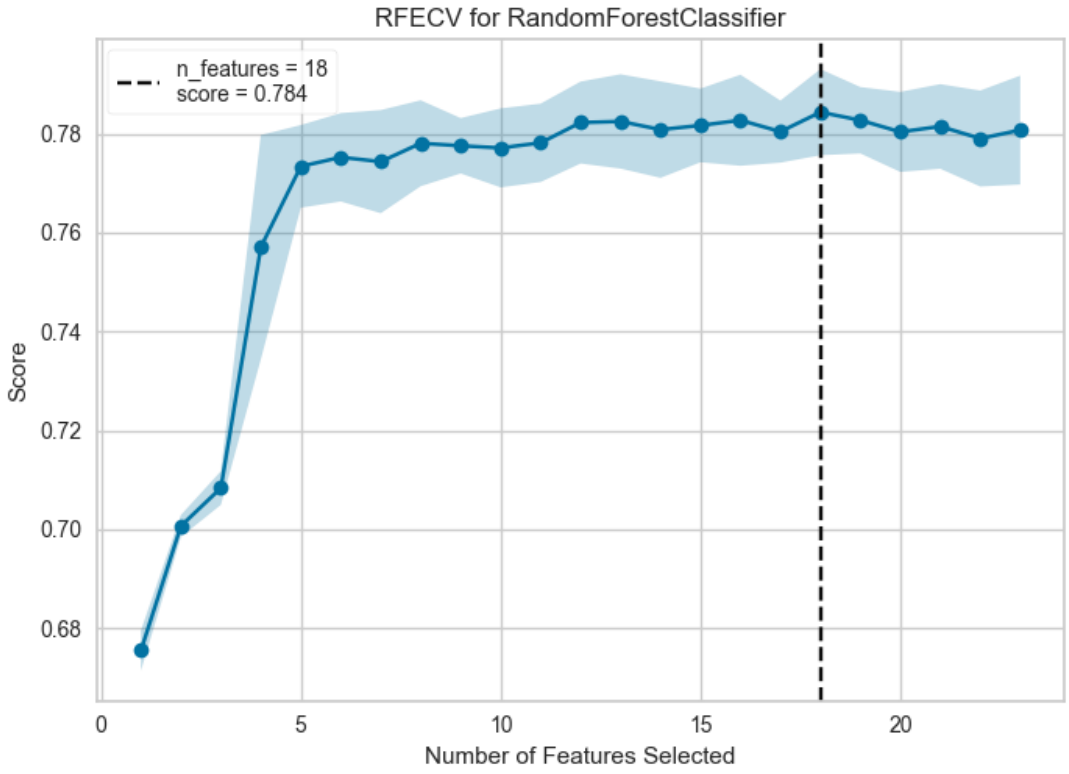
```
from sklearn.svm import SVC
from sklearn.datasets import make_classification

from yellowbrick.model_selection import RFECV

# Create a dataset with only 3 informative features
X, y = make_classification(
    n_samples=1000, n_features=25, n_informative=3, n_redundant=2,
    n_repeated=0, n_classes=8, n_clusters_per_class=1, random_state=0
)

# Instantiate RFECV visualizer with a linear SVM classifier
visualizer = RFECV(SVC(kernel='linear', C=1))

visualizer.fit(X, y)        # Fit the data to the visualizer
visualizer.show()           # Finalize and render the figure
```



This figure shows an ideal RFECV curve, the curve jumps to an excellent accuracy when the three informative features are captured, then gradually decreases in accuracy as the non informative features are added into the model. The shaded area represents the variability of cross-validation, one standard deviation above and below the mean accuracy score drawn by the curve.

Exploring a real dataset, we can see the impact of RFECV on a credit default binary classifier.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold

from yellowbrick.model_selection import RFECV
from yellowbrick.datasets import load_credit

# Load classification dataset
X, y = load_credit()

cv = StratifiedKFold(5)
visualizer = RFECV(RandomForestClassifier(), cv=cv, scoring='f1_weighted')

visualizer.fit(X, y)        # Fit the data to the visualizer
visualizer.show()           # Finalize and render the figure
```



In this example we can see that 19 features were selected, though there doesn't appear to be much improvement in the f1 score of the model after around 5 features. Selection of the features to eliminate plays a large role in determining the outcome of each recursion; modifying the `step` parameter to eliminate more than one feature at each step may help to eliminate the worst features early, strengthening the remaining features (and can also be used to speed up feature elimination for datasets with a large number of features).

See also:

This visualizer is is based off of the visualization in the scikit-learn documentation: recursive feature elimination with cross-validation. However, the Yellowbrick version does not use `sklearn.feature_selection.RFECV` but instead wraps `sklearn.feature_selection.RFE` models. The fitted model can be accessed on the visualizer using the `viz.rfe_estimator_` attribute, and in fact the visualizer acts as the fitted model when using `predict()` or `score()`.

Quick Method

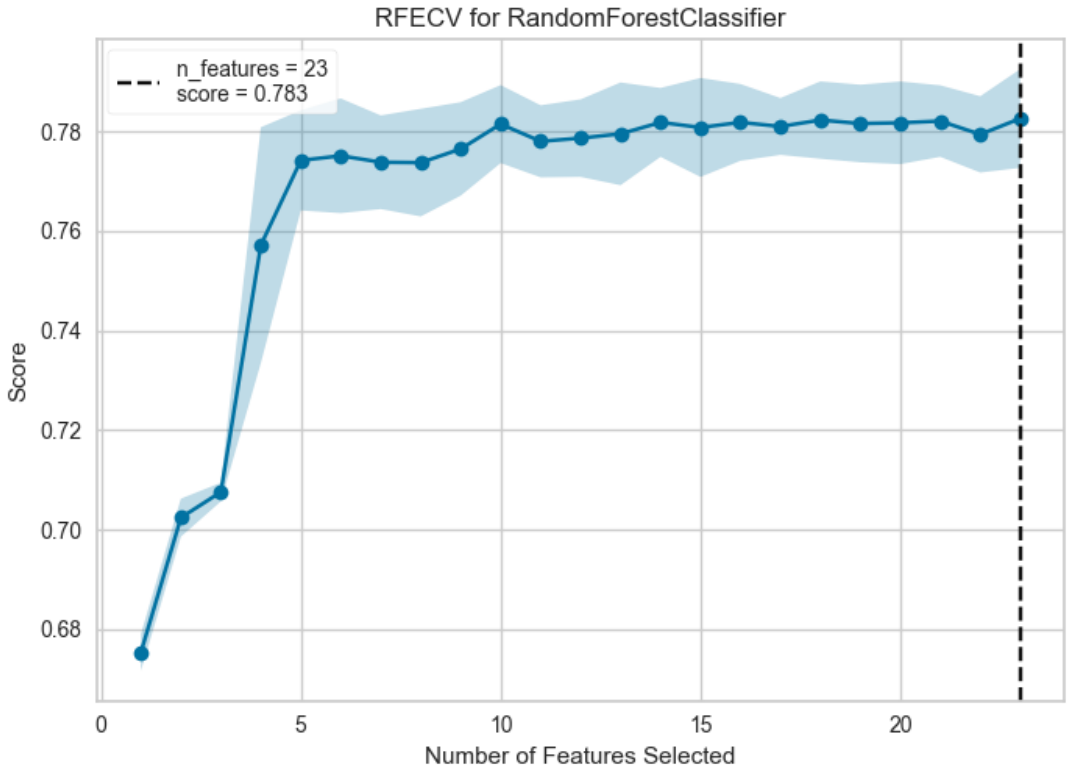
The same functionality above can be achieved with the associated quick method `rfecv`. This method will build the `RFECV` object with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold

from yellowbrick.model_selection import rfecv
from yellowbrick.datasets import load_credit

# Load classification dataset
X, y = load_credit()
```

(continues on next page)

(continued from previous page)

```
cv = StratifiedKFold(5)
visualizer = rfecv(RandomForestClassifier(), X=X, y=y, cv=cv, scoring='f1_weighted')
```



API Reference

Visualize the number of features selected using recursive feature elimination

class yellowbrick.model_selection.rfecv.**RFECV**(*estimator*, *ax=None*, *step=1*, *groups=None*, *cv=None*, *scoring=None*, ***kwargs*)

Bases: ModelVisualizer

Recursive Feature Elimination, Cross-Validated (RFECV) feature selection.

Selects the best subset of features for the supplied estimator by removing 0 to N features (where N is the number of features) using recursive feature elimination, then selecting the best subset based on the cross-validation score of the model. Recursive feature elimination eliminates n features from a model by fitting the model multiple times and at each step, removing the weakest features, determined by either the `coef_` or `feature_importances_` attribute of the fitted model.

The visualization plots the score relative to each subset and shows trends in feature elimination. If the feature elimination CV score is flat, then potentially there are not enough features in the model. An ideal curve is when the score jumps from low to high as the number of features removed increases, then slowly decreases again from the optimal number of features.

Parameters

estimator
[a scikit-learn estimator] An object that implements `fit` and provides information about the relative importance of features with either a `coef_` or `feature_importances_` attribute.

Note that the object is cloned for each validation.

ax
[matplotlib.Axes object, optional] The axes object to plot the figure on.

step
[int or float, optional (default=1)] If greater than or equal to 1, then step corresponds to the (integer) number of features to remove at each iteration. If within (0.0, 1.0), then step corresponds to the percentage (rounded down) of features to remove at each iteration.

groups
[array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

scoring
[string, callable or None, optional, default: None] A string or scorer callable object / function with signature `scorer(estimator, X, y)`. See scikit-learn model evaluation documentation for names of possible metrics.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

This model wraps `sklearn.feature_selection.RFE` and not `sklearn.feature_selection.RFECV` because access to the internals of the CV and RFE estimators is required for the visualization. The visualizer does take similar arguments, however it does not expose the same internal attributes.

Additionally, the RFE model can be accessed via the `rfe_estimator_` attribute. Once fitted, the visualizer acts as a wrapper for this estimator and not for the original model passed to the model. This way the visualizer model can be used to make predictions.

Caution: This visualizer requires a model that has either a `coef_` or `feature_importances_` attribute when fitted.

Attributes

n_features_
[int] The number of features in the selected subset

support_
[array of shape [n_features]] A mask of the selected features

ranking_
[array of shape [n_features]] The feature ranking, such that `ranking_[i]` corresponds to the ranked position of feature i. Selected features are assigned rank 1.

cv_scores_
[array of shape [n_subsets_of_features, n_splits]] The cross-validation scores for each subset of features and splits in the cross-validation strategy.

rfe_estimator_
[sklearn.feature_selection.RFE] A fitted RFE estimator wrapping the original estimator. All estimator functions such as `predict()` and `score()` are passed through to this estimator (it rewraps the original model).

n_feature_subsets_
[array of shape [n_subsets_of_features]] The number of features removed on each iteration of RFE, computed by the number of features in the dataset and the step parameter.

`draw`(***kwargs*)

Renders the rfecv curve.

`finalize`(***kwargs*)

Add the title, legend, and other visual final touches to the plot.

`fit`(*X*, *y=None*)

Fits the RFECV with the wrapped model to the specified data and draws the rfecv curve with the optimal number of features found.

Parameters

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y
[array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression.

Returns

self
[instance] Returns the instance of the RFECV visualizer.

`yellowbrick.model_selection.rfecv.`**`rfecv`**(*estimator*, *X*, *y*, *ax=None*, *step=1*, *groups=None*, *cv=None*, *scoring=None*, *show=True*, ***kwargs*)

Performs recursive feature elimination with cross-validation to determine an optimal number of features for a model. Visualizes the feature subsets with respect to the cross-validation score.

This helper function is a quick wrapper to utilize the RFECV visualizer for one-off analysis.

Parameters

estimator
[a scikit-learn estimator] An object that implements `fit` and provides information about the relative importance of features with either a `coef_` or `feature_importances_` attribute.

Note that the object is cloned for each validation.

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y
[array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression.

ax
[matplotlib.Axes object, optional] The axes object to plot the figure on.

step
[int or float, optional (default=1)] If greater than or equal to 1, then step corresponds to the (integer) number of features to remove at each iteration. If within (0.0, 1.0), then step corresponds to the percentage (rounded down) of features to remove at each iteration.

groups
[array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

scoring
[string, callable or None, optional, default: None] A string or scorer callable object / function with signature `scorer(estimator, X, y)`. See scikit-learn model evaluation documentation for names of possible metrics.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers. These arguments are also passed to the *show()* method, e.g. can pass a path to save the figure to.

Returns

viz
[RFECV] Returns the fitted, finalized visualizer.

Feature Dropping Curve

Visualizer	*DroppingCurve*
Quick Method	*dropping_curve()*
Models	Classification, Regression, Clustering
Workflow	Model Selection

A feature dropping curve (FDC) shows the relationship between the score and the number of features used. This visualizer randomly drops input features, showing how the estimator benefits from additional features of the same type. For example, how many air quality sensors are needed across a city to accurately predict city-wide pollution levels?

Feature dropping curves helpfully complement *Recursive Feature Elimination* (RFECV). In the air quality sensor example, RFECV finds which sensors to keep in the specific city. Feature dropping curves estimate how many sensors a similar-sized city might need to track pollution levels.

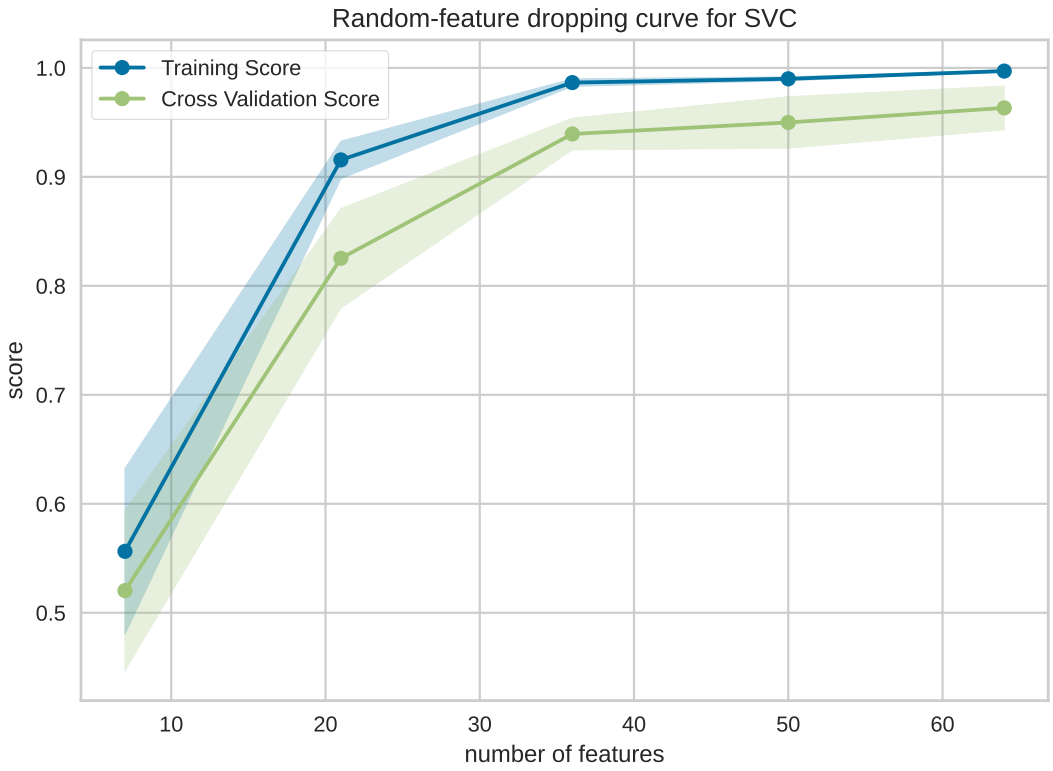
Feature dropping curves are common in the field of neural decoding, where they are called neuron dropping curves (example, panels C and H). Neural decoding research often quantifies how performance scales with neuron (or electrode) count. Because neurons do not correspond directly between participants, we use random neuron subsets to simulate what performance to expect when recording from other participants.

To show how this works in practice, consider an image classification example using handwritten digits.

```
from sklearn.svm import SVC
from sklearn.datasets import load_digits

from yellowbrick.model_selection import DroppingCurve

# Load dataset
X, y = load_digits(return_X_y=True)

# Initialize visualizer with estimator
visualizer = DroppingCurve(SVC())

# Fit the data to the visualizer
visualizer.fit(X, y)
# Finalize and render the figure
visualizer.show()
```

This figure shows an input feature dropping curve. Since the features are informative, the accuracy increases with more larger feature subsets. The shaded area represents the variability of cross-validation, one standard deviation above and below the mean accuracy score drawn by the curve.

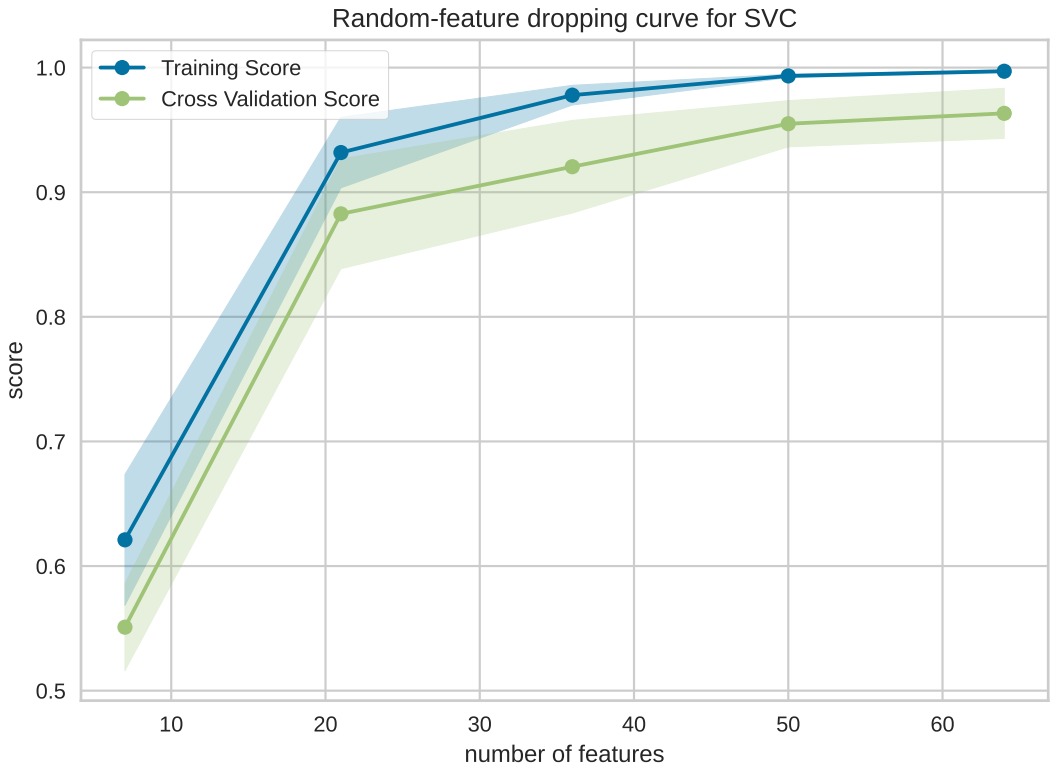
The visualization can be interpreted as the performance if we knew some image pixels were corrupted. As an alternative interpretation, the dropping curve roughly estimates the accuracy if the image resolution was downsampled.



Quick Method

The same functionality can be achieved with the associated quick method `dropping_curve`. This method will build the `DroppingCurve` with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from sklearn.svm import SVC
from sklearn.datasets import load_digits

from yellowbrick.model_selection import dropping_curve

# Load dataset
X, y = load_digits(return_X_y=True)

dropping_curve(SVC(), X, y)
```



API Reference

Implements a random-input-dropout curve visualization for model selection. Another common name: neuron dropping curve (NDC), in neural decoding research

class yellowbrick.model_selection.dropping_curve.**DroppingCurve**(*estimator*, *ax=None*, *feature_sizes=array([0.1, 0.325, 0.55, 0.775, 1.0])*, *groups=None*, *logx=False*, *cv=None*, *scoring=None*, *n_jobs=None*, *pre_dispatch='all'*, *random_state=None*, ***kwargs*)

Bases: `ModelVisualizer`

Selects random subsets of features and estimates the training and crossvalidation performance. Subset sizes are swept to visualize a feature-dropping curve.

The visualization plots the score relative to each subset and shows the number of (randomly selected) features needed to achieve a score. The curve is often shaped like log(1+x). For example, see: https://www.frontiersin.org/articles/10.3389/fnsys.2014.00102/full

Parameters

estimator
[a scikit-learn estimator] An object that implements `fit` and `predict`, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric.

Note that the object is cloned for each validation.

feature_sizes: array-like, shape (n_values,)
default: `np.linspace(0.1,1.0,5)`

Relative or absolute numbers of input features that will be used to generate the learning curve. If the dtype is float, it is regarded as a fraction of the maximum number of features, otherwise it is interpreted as absolute numbers of features.

groups
[array-like, with shape (n_samples,)] Optional group labels for the samples used while splitting the dataset into train/test sets.

ax
[matplotlib.Axes object, optional] The axes object to plot the figure on.

logx
[boolean, optional] If True, plots the x-axis with a logarithmic scale.

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

scoring
[string, callable or None, optional, default: None] A string or scorer callable object / function with signature `scorer(estimator, X, y)`. See scikit-learn model evaluation documentation for names of possible metrics.

n_jobs
[integer, optional] Number of jobs to run in parallel (default 1).

pre_dispatch
[integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like '2*n_jobs'.

random_state
[int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used to generate feature subsets.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

This visualizer is based on sklearn.model_selection.validation_curve

Examples

```
>>> from yellowbrick.model_selection import DroppingCurve
>>> from sklearn.naive_bayes import GaussianNB
>>> model = DroppingCurve(GaussianNB())
>>> model.fit(X, y)
>>> model.show()
```

Attributes

feature_sizes_
: [array, shape = (n_unique_ticks,), dtype int] Numbers of features that have been used to generate the dropping curve. Note that the number of ticks might be less than n_ticks because duplicate entries will be removed.

train_scores_
: [array, shape (n_ticks, n_cv_folds)] Scores on training sets.

train_scores_mean_
: [array, shape (n_ticks,)] Mean training data scores for each training split

train_scores_std_
: [array, shape (n_ticks,)] Standard deviation of training data scores for each training split

valid_scores_
: [array, shape (n_ticks, n_cv_folds)] Scores on validation set.

valid_scores_mean_
: [array, shape (n_ticks,)] Mean scores for each validation split

valid_scores_std_
: [array, shape (n_ticks,)] Standard deviation of scores for each validation split

draw(***kwargs*)

Renders the training and validation learning curves.

finalize(***kwargs*)

Add the title, legend, and other visual final touches to the plot.

fit(*X*, *y=None*)

Fits the feature dropping curve with the wrapped model to the specified data. Draws training and cross-validation score curves and saves the scores to the estimator.

Parameters

X
: [array-like, shape (n_samples, n_features)] Input vector, where n_samples is the number of samples and n_features is the number of features.

y
: [array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

yellowbrick.model_selection.dropping_curve.**dropping_curve**(*estimator*, *X*, *y*, *feature_sizes=array([0.1, 0.325, 0.55, 0.775, 1.0])*, *groups=None*, *ax=None*, *logx=False*, *cv=None*, *scoring=None*, *n_jobs=None*, *pre_dispatch='all'*, *random_state=None*, *show=True*, ***kwargs*) → *DroppingCurve*

Displays a random-feature dropping curve, comparing feature size to training and cross validation scores. The dropping curve aims to show how a model improves with more information.

This helper function wraps the DroppingCurve class for one-off analysis.

Parameters

estimator
[a scikit-learn estimator] An object that implements `fit` and `predict`, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric.

Note that the object is cloned for each validation.

X
[array-like, shape (n_samples, n_features)] Input vector, where n_samples is the number of samples and n_features is the number of features.

y
[array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

feature_sizes: array-like, shape (n_values,)
default: `np.linspace(0.1,1.0,5)`

Relative or absolute numbers of input features that will be used to generate the learning curve. If the dtype is float, it is regarded as a fraction of the maximum number of features, otherwise it is interpreted as absolute numbers of features.

groups
[array-like, with shape (n_samples,)] Optional group labels for the samples used while splitting the dataset into train/test sets.

ax
[matplotlib.Axes object, optional] The axes object to plot the figure on.

logx
[boolean, optional] If True, plots the x-axis with a logarithmic scale.

cv
[int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- An object to be used as a cross-validation generator.
- An iterable yielding train/test splits.

see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

scoring
[string, callable or None, optional, default: None] A string or scorer callable object / function

with signature `scorer(estimator, X, y)`. See scikit-learn model evaluation documentation for names of possible metrics.

n_jobs
[integer, optional] Number of jobs to run in parallel (default 1).

pre_dispatch
[integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like '2*n_jobs'.

random_state
[int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used to generate feature subsets.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Returns

dc
[DroppingCurve] Returns the fitted visualizer.

8.3.9 Text Modeling Visualizers

Yellowbrick provides the `yellowbrick.text` module for text-specific visualizers. The `TextVisualizer` class specifically deals with datasets that are corpora and not simple numeric arrays or DataFrames, providing utilities for analyzing word dispersion and distribution, showing document similarity, or simply wrapping some of the other standard visualizers with text-specific display properties.

We currently have five text-specific visualizations implemented:

- *Token Frequency Distribution*: plot the frequency of tokens in a corpus
- *t-SNE Corpus Visualization*: plot similar documents closer together to discover clusters
- *UMAP Corpus Visualization*: plot similar documents closer together to discover clusters
- *Dispersion Plot*: plot the dispersion of target words throughout a corpus
- *Word Correlation Plot*: plot the correlation between target words across the documents in a corpus
- *PosTag Visualization*: plot the counts of different parts-of-speech throughout a tagged corpus

Note that the examples in this section require a corpus of text data, see *the hobbies corpus* for a sample dataset.

```
from yellowbrick.text import FreqDistVisualizer
from yellowbrick.text import TSNEVisualizer
from yellowbrick.text import UMAPVisualizer
from yellowbrick.text import DispersionPlot
from yellowbrick.text import WordCorrelationPlot
from yellowbrick.text import PosTagVisualizer

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
```

Token Frequency Distribution

A method for visualizing the frequency of tokens within and across corpora is frequency distribution. A frequency distribution tells us the frequency of each vocabulary item in the text. In general, it could count any kind of observable event. It is a distribution because it tells us how the total number of word tokens in the text are distributed across the vocabulary items.

Visualizer	*FrequencyVisualizer*
Quick Method	*freqdist()*
Models	Text
Workflow	Text Analysis

Note: The `FreqDistVisualizer` does not perform any normalization or vectorization, and it expects text that has already been count vectorized.

We first instantiate a `FreqDistVisualizer` object, and then call `fit()` on that object with the count vectorized documents and the features (i.e. the words from the corpus), which computes the frequency distribution. The visualizer then plots a bar chart of the top 50 most frequent terms in the corpus, with the terms listed along the x-axis and frequency counts depicted at y-axis values. As with other Yellowbrick visualizers, when the user invokes `show()`, the finalized visualization is shown. Note that in this plot and in the subsequent one, we can orient our plot vertically by passing in `orient='v'` on instantiation (the plot will orient horizontally by default):

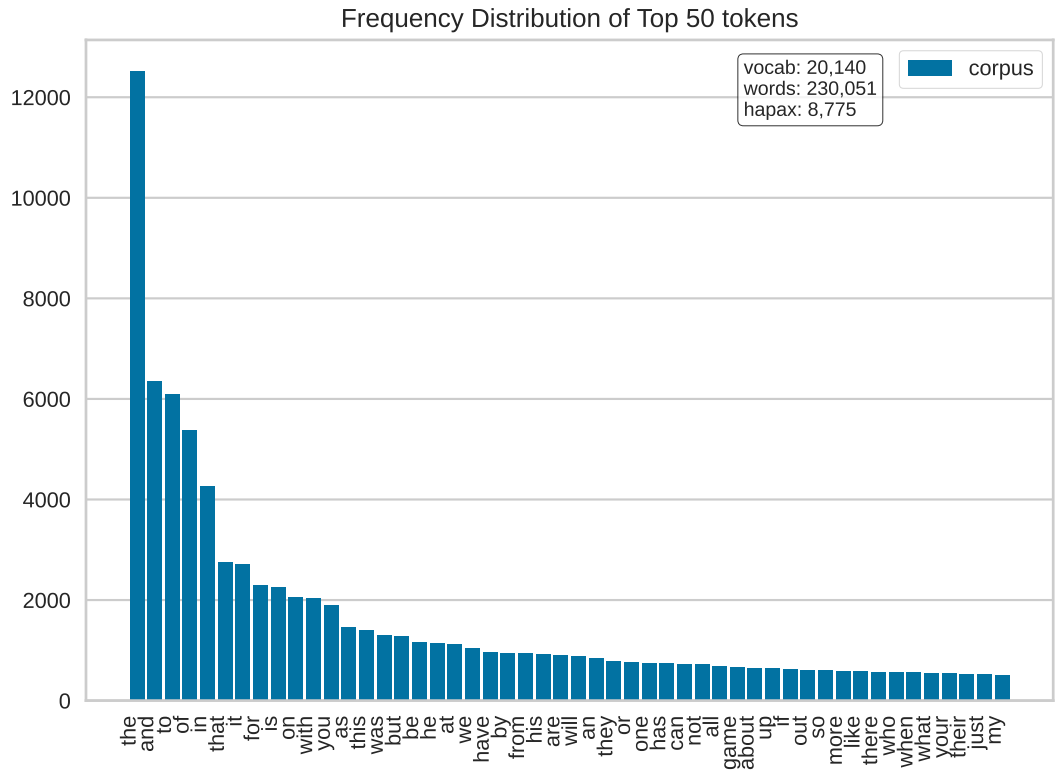
```
from sklearn.feature_extraction.text import CountVectorizer

from yellowbrick.text import FreqDistVisualizer
from yellowbrick.datasets import load_hobbies

# Load the text data
corpus = load_hobbies()

vectorizer = CountVectorizer()
docs       = vectorizer.fit_transform(corpus.data)
features   = vectorizer.get_feature_names()

visualizer = FreqDistVisualizer(features=features, orient='v')
visualizer.fit(docs)
visualizer.show()
```

It is interesting to compare the results of the `FreqDistVisualizer` before and after stopwords have been removed from the corpus:

It is also interesting to explore the differences in tokens across a corpus. The hobbies corpus that comes with Yellowbrick has already been categorized (try `corpus.target`), so let's visually compare the differences in the frequency distributions for two of the categories: *"cooking"* and *"gaming"*.

Here is the plot for the cooking corpus (oriented horizontally this time):

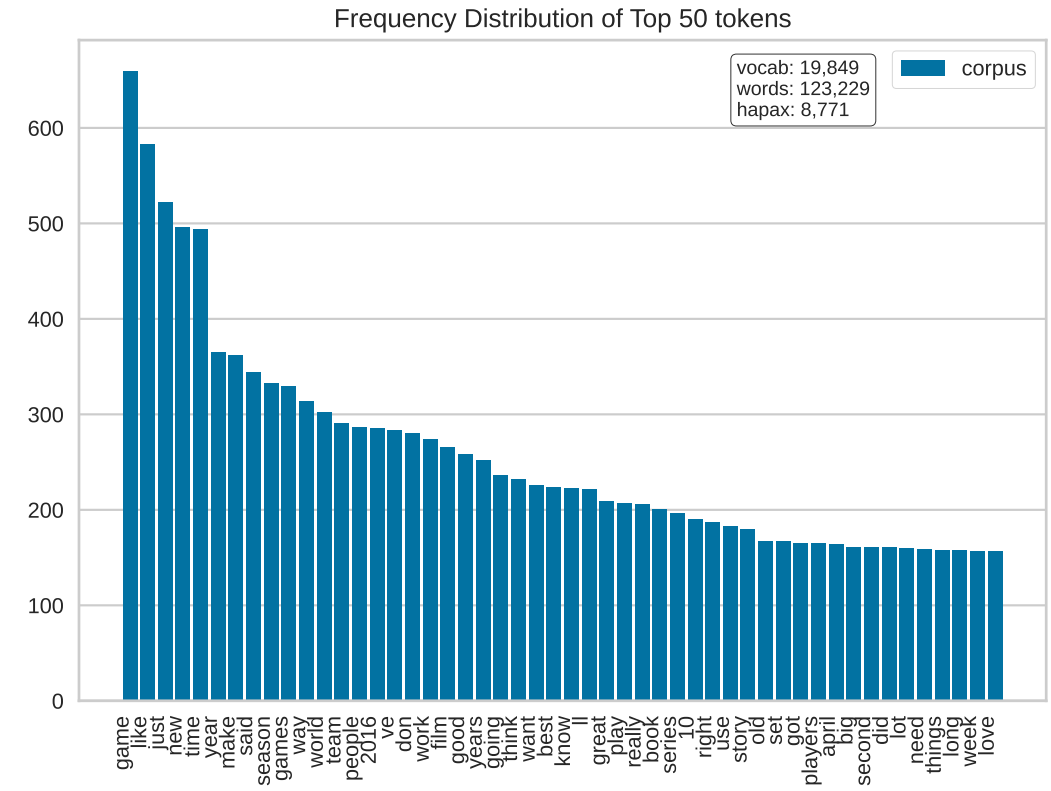
And for the gaming corpus (again oriented horizontally):

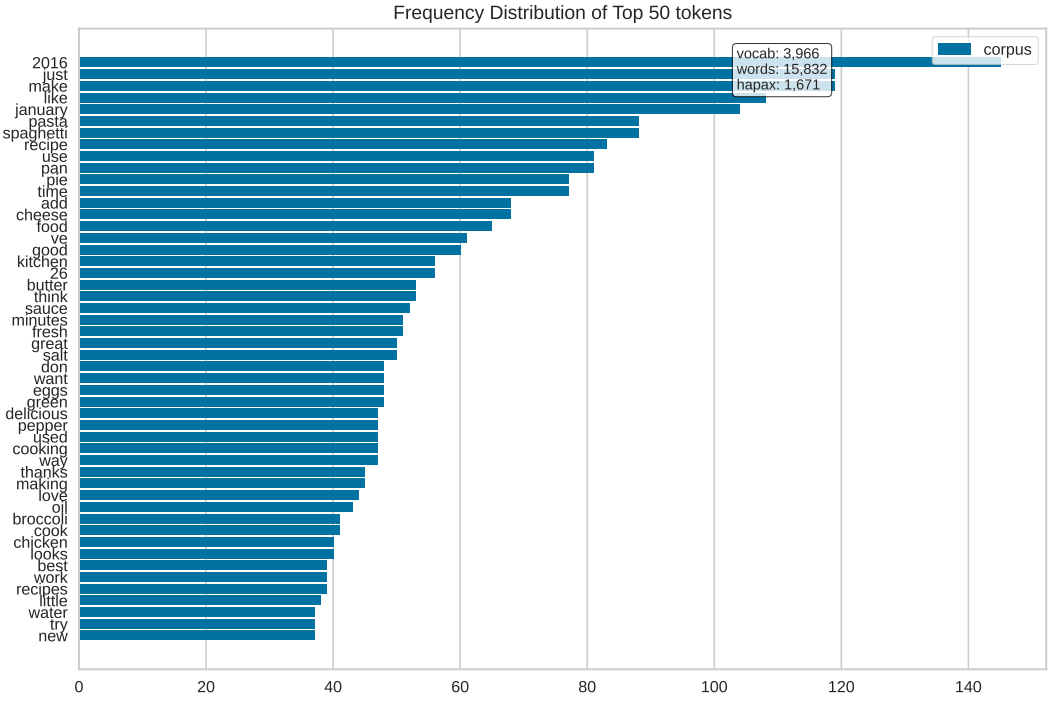


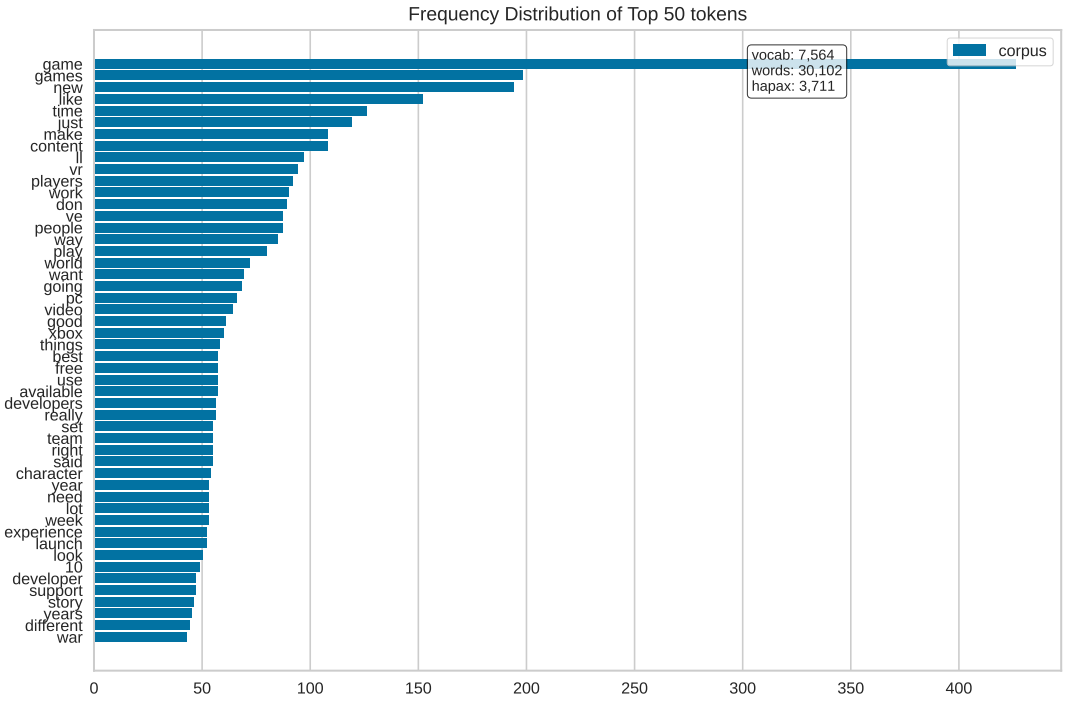
Quick Method

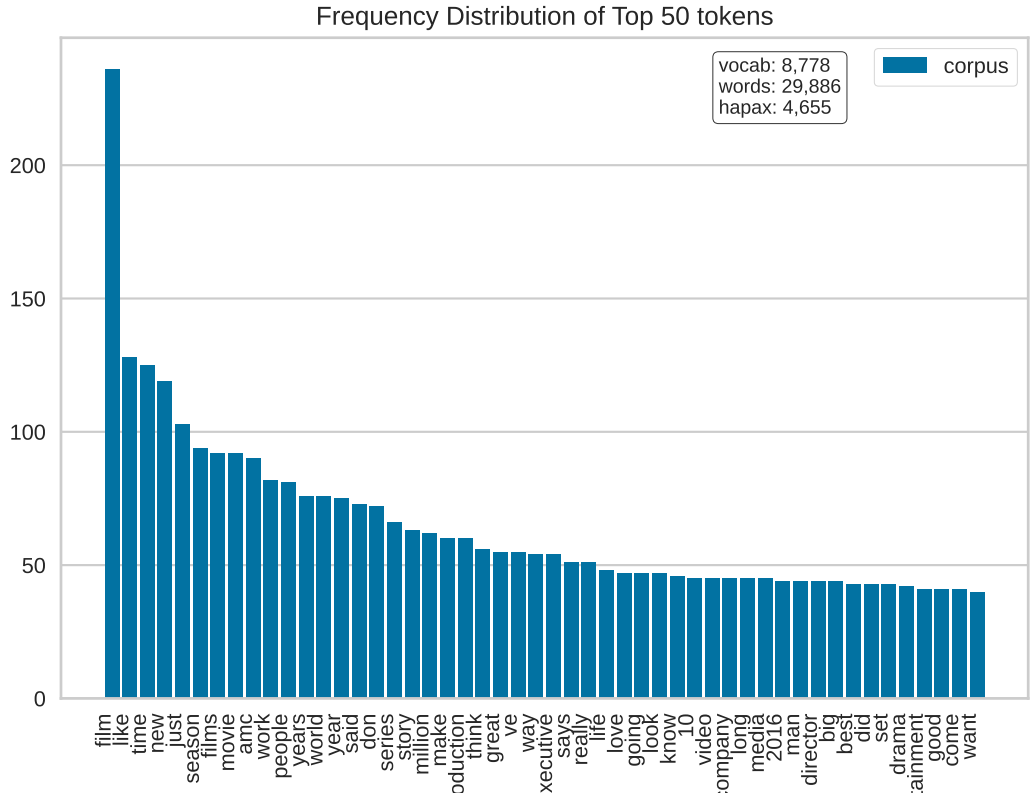
Similar functionality as above can be achieved in one line using the associated quick method, `freqdist`. This method will instantiate with features(words) and fit a `FreqDistVisualizer` visualizer on the documents).

```
from collections import defaultdict

from sklearn.feature_extraction.text import CountVectorizer

from yellowbrick.text import freqdist
from yellowbrick.datasets import load_hobbies

# Load the text data
corpus = load_hobbies()

# Create a dict to map target labels to documents of that category
hobbies = defaultdict(list)
for text, label in zip(corpus.data, corpus.target):
    hobbies[label].append(text)

vectorizer = CountVectorizer(stop_words='english')
docs       = vectorizer.fit_transform(text for text in hobbies['cinema'])
features   = vectorizer.get_feature_names()

freqdist(features, docs, orient='v')
```


Frequency Distribution of Top 50 tokens
vocab: 19,849
words: 123,229
hapax: 8,771
corpus
600
500
400
300
200
100
0
game
like
just
new
time
year
make
said
season
games
way
world
team
people
2016
ve
don
work
film
good
years
going
think
want
best
know
ll
great
play
really
book
series
10
right
use
story
old
set
got
players
april
big
second
did
lot
need
things
long
week
love


Frequency Distribution of Top 50 tokens
vocab: 3,966
words: 15,832
hapax: 1,671
corpus
2016
just
make
like
january
pasta
spaghetti
recipe
use
pan
pie
time
add
cheese
food
ve
good
kitchen
26
butter
think
sauce
minutes
fresh
great
salt
don
want
eggs
green
delicious
pepper
used
cooking
way
thanks
making
love
oil
broccoli
cook
chicken
looks
best
work
recipes
little
water
try
new
0
20
40
60
80
100
120
140


Frequency Distribution of Top 50 tokens
corpus
vocab: 7,564
words: 30,102
hapax: 3,711
game
games
new
like
time
just
make
content
ll
vr
players
work
don
ve
people
way
play
world
want
going
pc
video
good
xbox
things
best
free
use
available
developers
really
set
team
right
said
character
year
need
lot
week
experience
launch
look
10
developer
support
story
years
different
war
0
50
100
150
200
250
300
350
400


Frequency Distribution of Top 50 tokens
corpus
vocab: 8,778
words: 29,886
hapax: 4,655
0
50
100
150
200
film
like
time
new
just
season
films
movie
amc
work
people
years
world
year
said
don
series
story
million
make
oduction
think
great
ve
way
xecutive
says
really
life
love
going
look
know
10
video
ompany
long
media
2016
man
director
big
best
did
set
drama
ainment
good
come
want

API Reference

Implementations of frequency distributions for text visualization

class yellowbrick.text.freqdist.**FrequencyVisualizer**(*features*, *ax=None*, *n=50*, *orient='h'*, *color=None*, ***kwargs*)

Bases: TextVisualizer

A frequency distribution tells us the frequency of each vocabulary item in the text. In general, it could count any kind of observable event. It is a distribution because it tells us how the total number of word tokens in the text are distributed across the vocabulary items.

Parameters

features
[list, default: None] The list of feature names from the vectorizer, ordered by index. E.g. a lexicon that specifies the unique vocabulary of the corpus. This can be typically fetched using the `get_feature_names()` method of the transformer in Scikit-Learn.

ax
[matplotlib axes, default: None] The axes to plot the figure on.

n: integer, default: 50
Top N tokens to be plotted.

orient
['h' or 'v', default: 'h'] Specifies a horizontal or vertical bar chart.

color
[string] Specify color for bars

kwargs
[dict] Pass any additional keyword arguments to the super class.

These parameters can be influenced later on in the visualization process, but can and should be set as early as possible.

count(*X*)

Called from the fit method, this method gets all the words from the corpus and their corresponding frequency counts.

Parameters

X
[ndarray or masked ndarray] Pass in the matrix of vectorized documents, can be masked in order to sum the word frequencies for only a subset of documents.

Returns

counts
[array] A vector containing the counts of all words in X (columns)

draw(***kwargs*)

Called from the fit method, this method creates the canvas and draws the distribution plot on it.

Parameters

kwargs: generic keyword arguments.

finalize(***kwargs*)

The finalize method executes any subclass-specific axes finalization steps. The user calls show & show calls finalize.

Parameters

kwargs: generic keyword arguments.

`fit`(*X*, *y=None*)

The fit method is the primary drawing input for the frequency distribution visualization. It requires vectorized lists of documents and a list of features, which are the actual words from the original corpus (needed to label the x-axis ticks).

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features representing the corpus of frequency vectorized documents.

y
[ndarray or DataFrame of shape n] Labels for the documents for conditional frequency distribution.

Notes

Note: Text documents must be vectorized before `fit()`.

`yellowbrick.text.freqdist.`**`freqdist`**(*features*, *X*, *y=None*, *ax=None*, *n=50*, *orient='h'*, *color=None*, *show=True*, ***kwargs*)

Displays frequency distribution plot for text.

This helper function is a quick wrapper to utilize the FreqDist Visualizer (Transformer) for one-off analysis.

Parameters

features
[list, default: None] The list of feature names from the vectorizer, ordered by index. E.g. a lexicon that specifies the unique vocabulary of the corpus. This can be typically fetched using the `get_feature_names()` method of the transformer in Scikit-Learn.

X: ndarray or DataFrame of shape n x m
A matrix of n instances with m features. In the case of text, X is a list of list of already preprocessed words

y: ndarray or Series of length n
An array or series of target or class values

ax
[matplotlib axes, default: None] The axes to plot the figure on.

n: integer, default: 50
Top N tokens to be plotted.

orient
['h' or 'v', default: 'h'] Specifies a horizontal or vertical bar chart.

color
[string] Specify color for bars

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs: dict
Keyword arguments passed to the super class.

Returns

visualizer: FreqDistVisualizer
Returns the fitted, finalized visualizer

t-SNE Corpus Visualization

Visualizer	*TSNEVisualizer*
Quick Method	*tsne()*
Models	Decomposition
Workflow	Feature Engineering/Selection

One very popular method for visualizing document similarity is to use t-distributed stochastic neighbor embedding, t-SNE. Scikit-learn implements this decomposition method as the `sklearn.manifold.TSNE` transformer. By decomposing high-dimensional document vectors into 2 dimensions using probability distributions from both the original dimensionality and the decomposed dimensionality, t-SNE is able to effectively cluster similar documents. By decomposing to 2 or 3 dimensions, the documents can be visualized with a scatter plot.

Unfortunately, `TSNE` is very expensive, so typically a simpler decomposition method such as SVD or PCA is applied ahead of time. The `TSNEVisualizer` creates an inner transformer pipeline that applies such a decomposition first (SVD with 50 components by default), then performs the t-SNE embedding. The visualizer then plots the scatter plot, coloring by cluster or by class, or neither if a structural analysis is required.

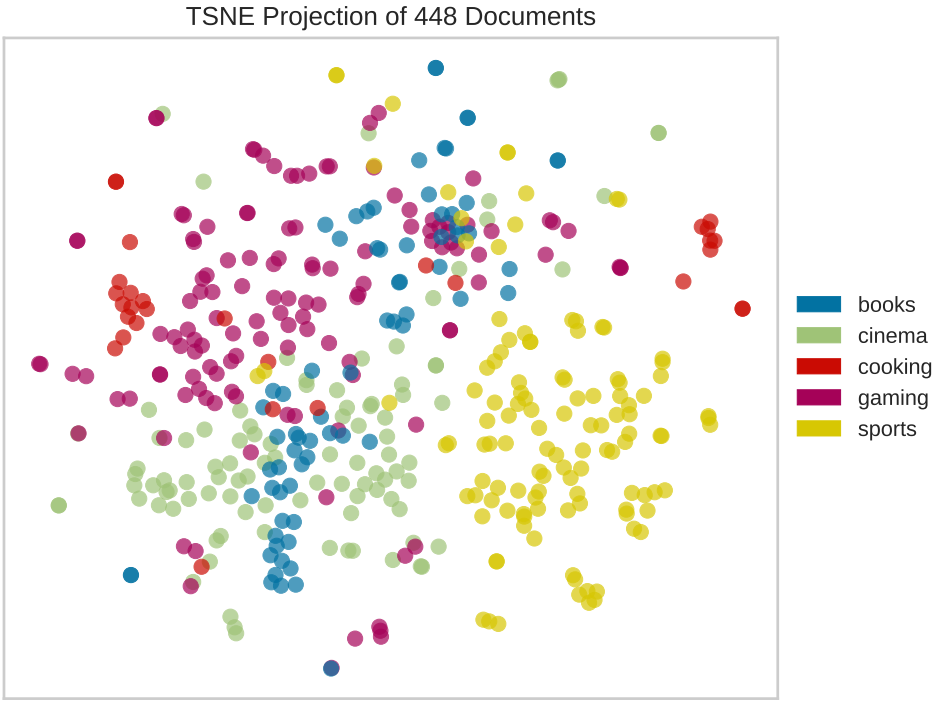
After importing the required tools, we can use the *hobbies corpus* and vectorize the text using TF-IDF. Once the corpus is vectorized we can visualize it, showing the distribution of classes.

```
from sklearn.feature_extraction.text import TfidfVectorizer

from yellowbrick.text import TSNEVisualizer
from yellowbrick.datasets import load_hobbies

# Load the data and create document vectors
corpus = load_hobbies()
tfidf = TfidfVectorizer()

X = tfidf.fit_transform(corpus.data)
y = corpus.target

# Create the visualizer and draw the vectors
tsne = TSNEVisualizer()
tsne.fit(X, y)
tsne.show()
```

Note that you can pass the class labels or document categories directly to the `TSNEVisualizer` as follows:

```
labels = corpus.labels
tsne = TSNEVisualizer(labels=labels)
tsne.fit(X, y)
tsne.show()
```

If we omit the target during fit, we can visualize the whole dataset to see if any meaningful patterns are observed.

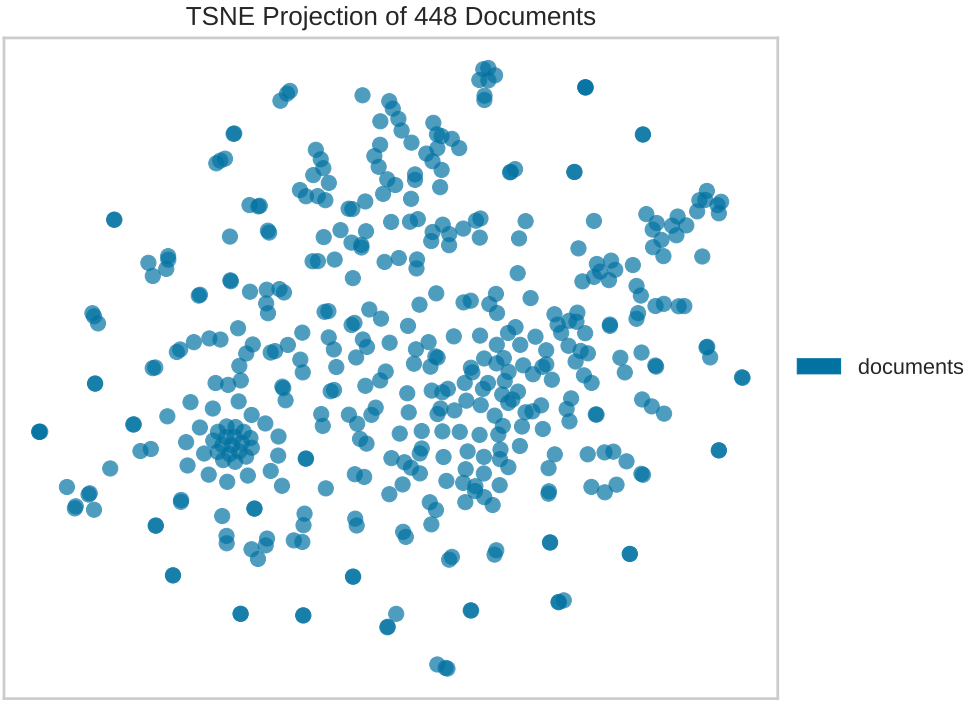


This means we don't have to use class labels at all. Instead we can use cluster membership from K-Means to label each document. This will allow us to look for clusters of related text by their contents:

Quick Method

The same functionality above can be achieved with the associated quick method `tsne`. This method will build the `TSNEVisualizer` object with the associated arguments, fit it, then (optionally) immediately show it

```
from yellowbrick.text.tsne import tsne
from sklearn.feature_extraction.text import TfidfVectorizer
from yellowbrick.datasets import load_hobbies

# Load the data and create document vectors
corpus = load_hobbies()
tfidf = TfidfVectorizer()

X = tfidf.fit_transform(corpus.data)
y = corpus.target

tsne(X, y)
```



API Reference

Implements TSNE visualizations of documents in 2D space.

class yellowbrick.text.tsne.**TSNEVisualizer**(*ax=None*, *decompose='svd'*, *decompose_by=50*, *labels=None*, *classes=None*, *colors=None*, *colormap=None*, *random_state=None*, *alpha=0.7*, ***kwargs*)

Bases: TextVisualizer

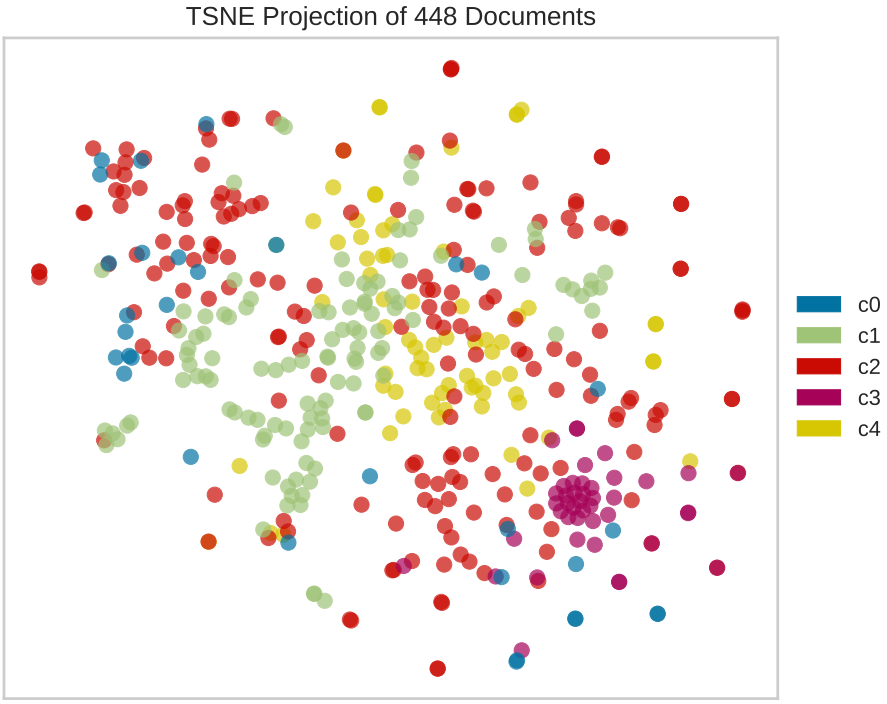
Display a projection of a vectorized corpus in two dimensions using TSNE, a nonlinear dimensionality reduction method that is particularly well suited to embedding in two or three dimensions for visualization as a scatter plot. TSNE is widely used in text analysis to show clusters or groups of documents or utterances and their relative proximities.

TSNE will return a scatter plot of the vectorized corpus, such that each point represents a document or utterance. The distance between two points in the visual space is embedded using the probability distribution of pairwise similarities in the higher dimensionality; thus TSNE shows clusters of similar documents and the relationships between groups of documents as a scatter plot.

TSNE can be used with either clustering or classification; by specifying the `classes` argument, points will be colored based on their similar traits. For example, by passing `cluster.labels_` as `y` in `fit()`, all points in the same cluster will be grouped together. This extends the neighbor embedding with more information about similarity, and can allow better interpretation of both clusters and classes.

For more, see https://lvdmaaten.github.io/tsne/

Parameters



ax
: [matplotlib axes] The axes to plot the figure on.

decompose
: [string or None, default: `'svd'`] A preliminary decomposition is often used prior to TSNE to make the projection faster. Specify `"svd"` for sparse data or `"pca"` for dense data. If None, the original data set will be used.

decompose_by
: [int, default: 50] Specify the number of components for preliminary decomposition, by default this is 50; the more components, the slower TSNE will be.

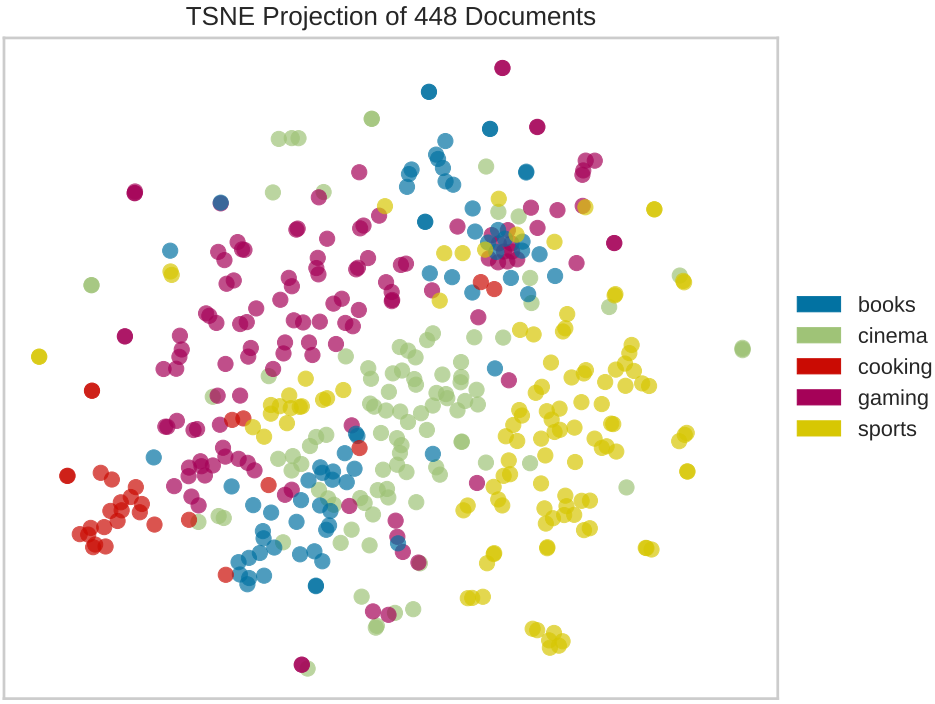
labels
: [list of strings] The names of the classes in the target, used to create a legend. Labels must match names of classes in sorted order.

colors
: [list or tuple of colors] Specify the colors for each individual class

colormap
: [string or matplotlib cmap] Sequential colormap for continuous target

random_state
: [int, RandomState instance or None, optional, default: None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. The random state is applied to the preliminary decomposition as well as tSNE.

alpha



[float, default: 0.7] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

kwargs
[dict] Pass any additional keyword arguments to the TSNE transformer.

NULL_CLASS = None

draw(*points*, *target=None*, ***kwargs*)

Called from the fit method, this method draws the TSNE scatter plot, from a set of decomposed points in 2 dimensions. This method also accepts a third dimension, target, which is used to specify the colors of each of the points. If the target is not specified, then the points are plotted as a single cloud to show similar documents.

finalize(***kwargs*)

Finalize the drawing by adding a title and legend, and removing the axes objects that do not convey information about TNSE.

fit(*X*, *y=None*, ***kwargs*)

The fit method is the primary drawing input for the TSNE projection since the visualization requires both X and an optional y value. The fit method expects an array of numeric vectors, so text documents must be vectorized before passing them to this method.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features representing the corpus of vectorized documents to visualize with tsne.

y
[ndarray or Series of length n] An optional array or series of target or class values for instances. If this is specified, then the points will be colored according to their class. Often cluster labels are passed in to color the documents in cluster space, so this method is used both for classification and clustering methods.

kwargs
[dict] Pass generic arguments to the drawing method

Returns

self
[instance] Returns the instance of the transformer/visualizer

`make_transformer`(*decompose='svd'*, *decompose_by=50*, *tsne_kwargs={}*)

Creates an internal transformer pipeline to project the data set into 2D space using TSNE, applying an pre-decomposition technique ahead of embedding if necessary. This method will reset the transformer on the class, and can be used to explore different decompositions.

Parameters

decompose
[string or None, default: `'svd'`] A preliminary decomposition is often used prior to TSNE to make the projection faster. Specify `"svd"` for sparse data or `"pca"` for dense data. If decompose is None, the original data set will be used.

decompose_by
[int, default: 50] Specify the number of components for preliminary decomposition, by default this is 50; the more components, the slower TSNE will be.

Returns

transformer
[Pipeline] Pipelined transformer for TSNE projections

`yellowbrick.text.tsne.`**`tsne`**(*X*, *y=None*, *ax=None*, *decompose='svd'*, *decompose_by=50*, *labels=None*, *colors=None*, *colormap=None*, *alpha=0.7*, *show=True*, ***kwargs*)

Display a projection of a vectorized corpus in two dimensions using TSNE, a nonlinear dimensionality reduction method that is particularly well suited to embedding in two or three dimensions for visualization as a scatter plot. TSNE is widely used in text analysis to show clusters or groups of documents or utterances and their relative proximities.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features representing the corpus of vectorized documents to visualize with tsne.

y
[ndarray or Series of length n] An optional array or series of target or class values for instances. If this is specified, then the points will be colored according to their class. Often cluster labels are passed in to color the documents in cluster space, so this method is used both for classification and clustering methods.

ax
[matplotlib axes] The axes to plot the figure on.

decompose
[string or None] A preliminary decomposition is often used prior to TSNE to make the projection faster. Specify *"svd"* for sparse data or *"pca"* for dense data. If decompose is None, the original data set will be used.

decompose_by
[int] Specify the number of components for preliminary decomposition, by default this is 50; the more components, the slower TSNE will be.

labels
[list of strings] The names of the classes in the target, used to create a legend.

colors
[list or tuple of colors] Specify the colors for each individual class

colormap
[string or matplotlib cmap] Sequential colormap for continuous target

alpha
[float, default: 0.7] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Pass any additional keyword arguments to the TSNE transformer.

Returns

visualizer: TSNEVisualizer
Returns the fitted, finalized visualizer

UMAP Corpus Visualization

Uniform Manifold Approximation and Projection (UMAP) is a nonlinear dimensionality reduction method that is well suited to embedding in two or three dimensions for visualization as a scatter plot. UMAP is a relatively new technique but is very effective for visualizing clusters or groups of data points and their relative proximities. It does a good job of learning the local structure within your data but also attempts to preserve the relationships between your groups as can be seen in its exploration of MNIST. It is fast, scalable, and can be applied directly to sparse matrices, eliminating the need to run `TruncatedSVD` as a pre-processing step. Additionally, it supports a wide variety of distance measures allowing for easy exploration of your data. For a more detailed explanation of the algorithm the paper can be found here.

Visualizer	*UMAPVisualizer*
Quick Method	*umap()*
Models	Decomposition
Workflow	Feature Engineering/Selection

In this example, we represent documents via a term frequency inverse document frequency (TF-IDF) vector and then use UMAP to find a low dimensional representation of these documents. Next, the Yellowbrick visualizer plots the scatter plot, coloring by cluster or by class, or neither if a structural analysis is required.

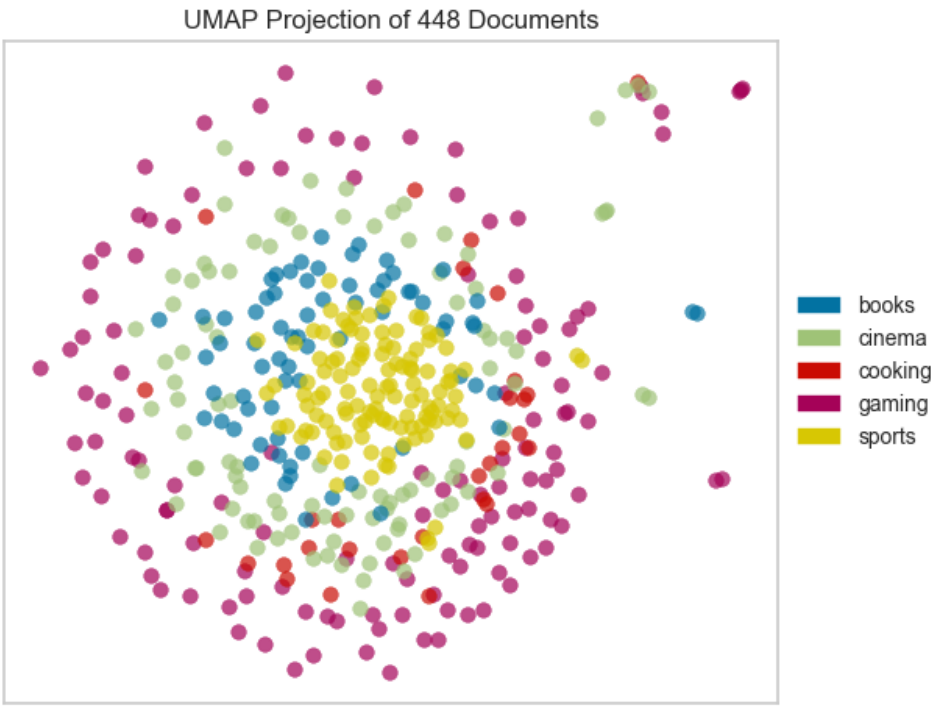
After importing the required tools, we can use the *the hobbies corpus* and vectorize the text using TF-IDF. Once the corpus is vectorized we can visualize it, showing the distribution of classes.

```
from sklearn.feature_extraction.text import TfidfVectorizer

from yellowbrick.datasets import load_hobbies
from yellowbrick.text import UMAPVisualizer
```

(continues on next page)

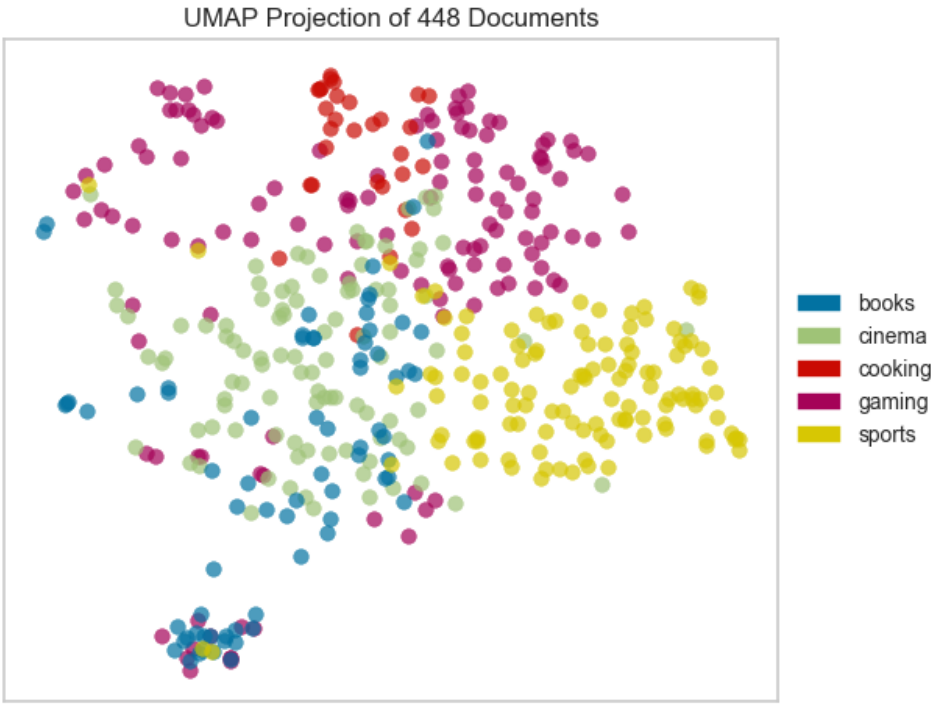
(continued from previous page)

```
# Load the text data
corpus = load_hobbies()

tfidf = TfidfVectorizer()
docs = tfidf.fit_transform(corpus.data)
labels = corpus.target

# Instantiate the text visualizer
umap = UMAPVisualizer()
umap.fit(docs, labels)
umap.show()
```



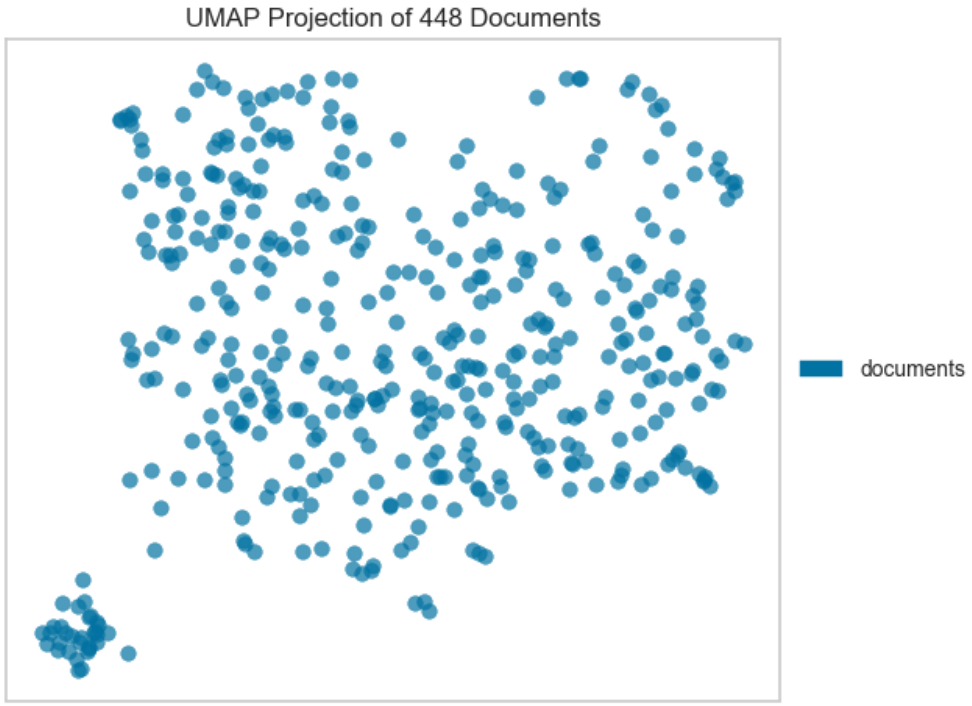
Alternatively, if we believed that cosine distance was a more appropriate metric on our feature space we could specify that via a `metric` parameter passed through to the underlying UMAP function by the `UMAPVisualizer`.

```
umap = UMAPVisualizer(metric='cosine')
umap.fit(docs, labels)
umap.show()
```



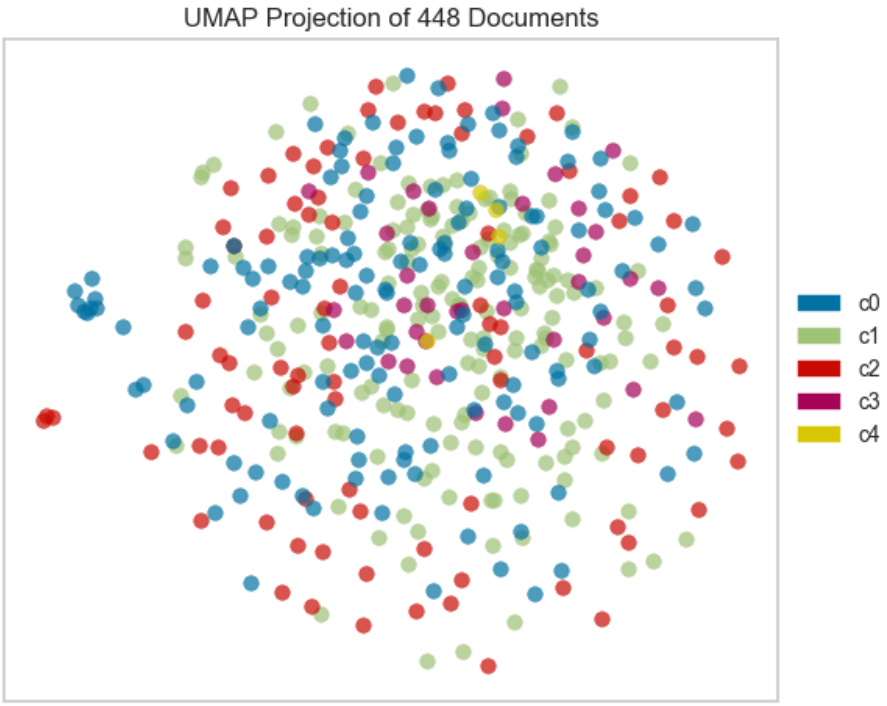
If we omit the target during fit, we can visualize the whole dataset to see if any meaningful patterns are observed.

```
# Don't color points with their classes
umap = UMAPVisualizer(labels=["documents"], metric='cosine')
umap.fit(docs)
umap.show()
```



This means we don't have to use class labels at all. Instead, we can use cluster membership from K-Means to label each document. This will allow us to look for clusters of related text by their contents:

```
from sklearn.cluster import KMeans
from sklearn.feature_extraction.text import TfidfVectorizer

from yellowbrick.datasets import load_hobbies
from yellowbrick.text import UMAPVisualizer

# Load the text data
corpus = load_hobbies()

tfidf = TfidfVectorizer()
docs = tfidf.fit_transform(corpus.data)

# Instantiate the clustering model
clusters = KMeans(n_clusters=5)
clusters.fit(docs)

umap = UMAPVisualizer()
umap.fit(docs, ["c{}".format(c) for c in clusters.labels_])
umap.show()
```



On one hand, these clusters aren't particularly well concentrated by the two-dimensional embedding of UMAP; while on the other hand, the true labels for this data are. That is a good indication that your data does indeed live on a manifold in your TF-IDF space and that structure is being ignored by the K-Means algorithm. Clustering can be quite tricky in high dimensional spaces and it is often a good idea to reduce your dimension before running clustering algorithms on your data.

UMAP, it should be noted, is a manifold learning technique and as such does not seek to preserve the distances between your data points in high space but instead to learn the distances along an underlying manifold on which your data points lie. As such, one shouldn't be too surprised when it disagrees with a non-manifold based clustering technique. A detailed explanation of this phenomenon can be found in this UMAP documentation.

Quick Method

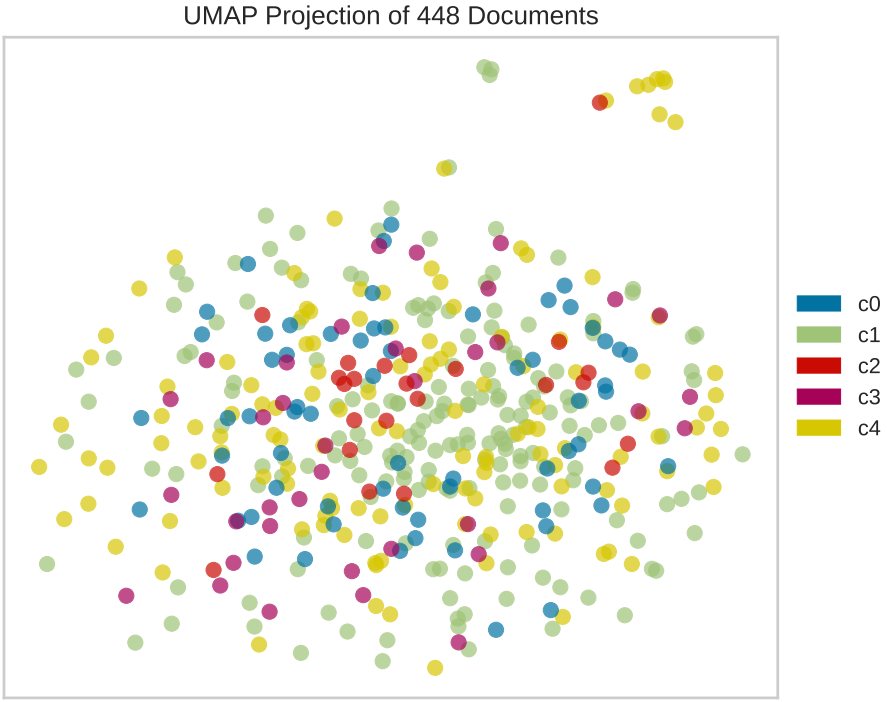
The same functionality above can be achieved with the associated quick method `umap`. This method will build the `UMAPVisualizer` object with the associated arguments, fit it, then (optionally) immediately show it

```
from sklearn.cluster import KMeans
from sklearn.feature_extraction.text import TfidfVectorizer

from yellowbrick.text import umap
from yellowbrick.datasets import load_hobbies

# Load the text data
corpus = load_hobbies()
```

(continues on next page)

(continued from previous page)

```
tfidf = TfidfVectorizer()
docs = tfidf.fit_transform(corpus.data)

# Instantiate the clustering model
clusters = KMeans(n_clusters=5)
clusters.fit(docs)

viz = umap(docs, ["c{}".format(c) for c in clusters.labels_])
```



API Reference

Implements UMAP visualizations of documents in 2D space.

class yellowbrick.text.umap_vis.**UMAPVisualizer**(*ax=None*, *labels=None*, *classes=None*, *colors=None*, *colormap=None*, *random_state=None*, *alpha=0.7*, ***kwargs*)

Bases: TextVisualizer

Display a projection of a vectorized corpus in two dimensions using UMAP (Uniform Manifold Approximation and Projection), a nonlinear dimensionality reduction method that is particularly well suited to embedding in two or three dimensions for visualization as a scatter plot. UMAP is a relatively new technique but is often used to visualize clusters or groups of data points and their relative proximities. It typically is fast, scalable, and can be applied directly to sparse matrices eliminating the need to run a TruncatedSVD as a pre-processing step.

The current default for UMAP is Euclidean distance. Hellinger distance would be a more appropriate distance function to use with CountVectorize data. That will be released in a forthcoming version of UMAP. In the meantime cosine distance is likely a better text default that Euclidean and can be set using the keyword argument `metric='cosine'`.

For more, see https://github.com/lmcinnes/umap

Parameters

ax
[matplotlib axes] The axes to plot the figure on.

labels
[list of strings] The names of the classes in the target, used to create a legend. Labels must match names of classes in sorted order.

colors
[list or tuple of colors] Specify the colors for each individual class

colormap
[string or matplotlib cmap] Sequential colormap for continuous target

random_state
[int, RandomState instance or None, optional, default: None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. The random state is applied to the preliminary decomposition as well as UMAP.

alpha
[float, default: 0.7] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

kwargs
[dict] Pass any additional keyword arguments to the UMAP transformer.

Examples

```
>>> model = MyVisualizer(metric='cosine')
>>> model.fit(X)
>>> model.show()
```

NULL_CLASS = None

draw(*points*, *target=None*, ***kwargs*)

Called from the fit method, this method draws the UMAP scatter plot, from a set of decomposed points in 2 dimensions. This method also accepts a third dimension, target, which is used to specify the colors of each of the points. If the target is not specified, then the points are plotted as a single cloud to show similar documents.

finalize(***kwargs*)

Finalize the drawing by adding a title and legend, and removing the axes objects that do not convey information about UMAP.

fit(*X*, *y=None*, ***kwargs*)

The fit method is the primary drawing input for the UMAP projection since the visualization requires both X and an optional y value. The fit method expects an array of numeric vectors, so text documents must be vectorized before passing them to this method.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features representing the corpus of vectorized documents to visualize with UMAP.

y
[ndarray or Series of length n] An optional array or series of target or class values for instances. If this is specified, then the points will be colored according to their class. Often cluster labels are passed in to color the documents in cluster space, so this method is used both for classification and clustering methods.

kwargs
[dict] Pass generic arguments to the drawing method

Returns

self
[instance] Returns the instance of the transformer/visualizer

make_transformer(*umap_kwargs={}*)

Creates an internal transformer pipeline to project the data set into 2D space using UMAP. This method will reset the transformer on the class.

Parameters

umap_kwargs
[dict] Keyword arguments for the internal UMAP transformer

Returns

transformer
[Pipeline] Pipelined transformer for UMAP projections

yellowbrick.text.umap_vis.**umap**(*X*, *y=None*, *ax=None*, *classes=None*, *colors=None*, *colormap=None*, *alpha=0.7*, *show=True*, ***kwargs*)

Display a projection of a vectorized corpus in two dimensions using UMAP (Uniform Manifold Approximation and Projection), a nonlinear dimensionality reduction method that is particularly well suited to embedding in two or three dimensions for visualization as a scatter plot. UMAP is a relatively new technique but is often used to visualize clusters or groups of data points and their relative proximities. It typically is fast, scalable, and can be applied directly to sparse matrices eliminating the need to run a `TruncatedSVD` as a pre-processing step.

The current default for UMAP is Euclidean distance. Hellinger distance would be a more appropriate distance function to use with CountVectorize data. That will be released in a forthcoming version of UMAP. In the meantime cosine distance is likely a better text default that Euclidean and can be set using the keyword argument `metric='cosine'`.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features representing the corpus of vectorized documents to visualize with umap.

y
[ndarray or Series of length n] An optional array or series of target or class values for instances. If this is specified, then the points will be colored according to their class. Often cluster labels are passed in to color the documents in cluster space, so this method is used both for classification and clustering methods.

ax
[matplotlib axes] The axes to plot the figure on.

classes
[list of strings] The names of the classes in the target, used to create a legend.

colors
[list or tuple of colors] Specify the colors for each individual class

colormap
[string or matplotlib cmap] Sequential colormap for continuous target

alpha
[float, default: 0.7] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Pass any additional keyword arguments to the UMAP transformer.

——-
visualizer: UMAPVisualizer
Returns the fitted, finalized visualizer

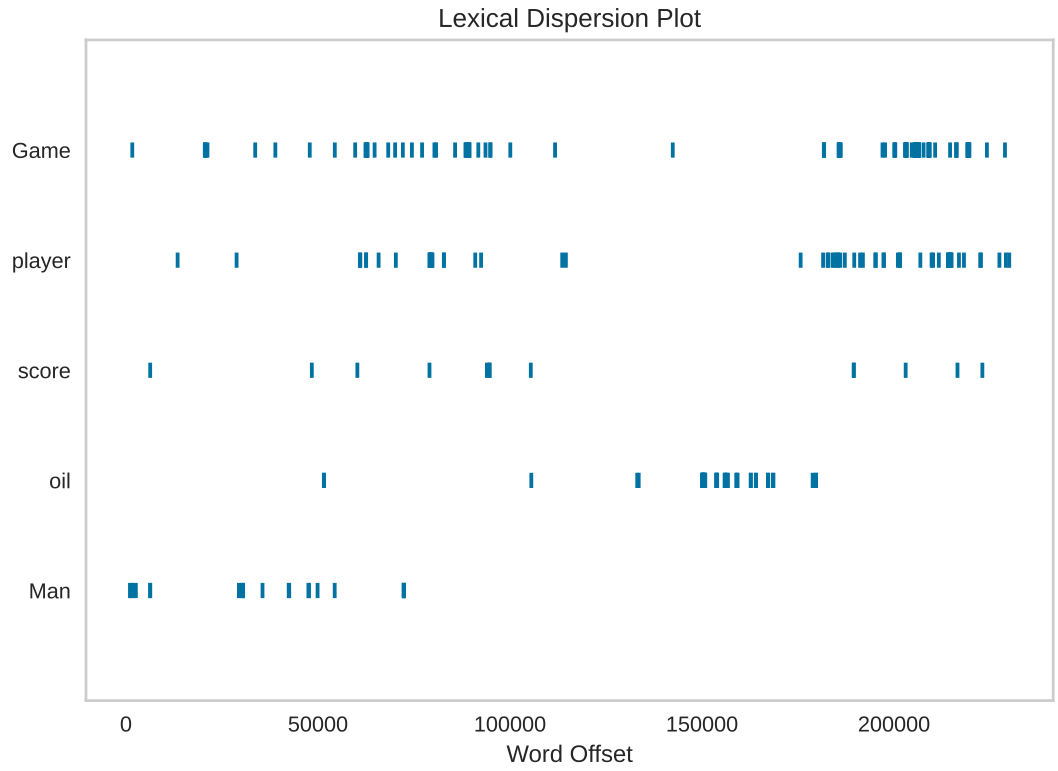
Dispersion Plot

A word's importance can be weighed by its dispersion in a corpus. Lexical dispersion is a measure of a word's homogeneity across the parts of a corpus.

Lexical dispersion illustrates the homogeneity of a word (or set of words) across the documents of a corpus. `DispersionPlot` allows for visualization of the lexical dispersion of words in a corpus. This plot illustrates with vertical lines the occurrences of one or more search terms throughout the corpus, noting how many words relative to the beginning of the corpus it appears.

Visualizer	*DispersionPlot*
Quick Method	*dispersion()*
Models	Text Modeling
Workflow	Feature Engineering

```
from yellowbrick.text import DispersionPlot
from yellowbrick.datasets import load_hobbies

# Load the text data
corpus = load_hobbies()

# Create a list of words from the corpus text
text = [doc.split() for doc in corpus.data]

# Choose words whose occurence in the text will be plotted
target_words = ['Game', 'player', 'score', 'oil', 'Man']

# Create the visualizer and draw the plot
visualizer = DispersionPlot(target_words)
visualizer.fit(text)
visualizer.show()
```



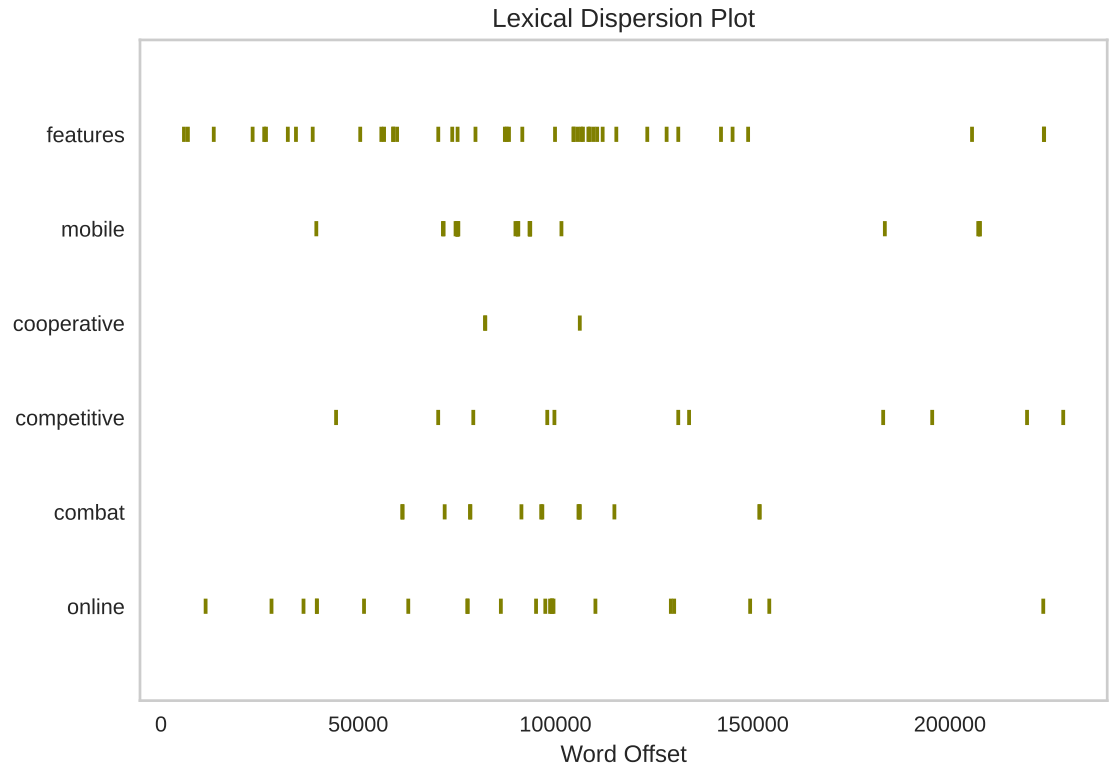
If the target vector of the corpus documents is provided, the points will be colored with respect to their document category, which allows for additional analysis of relationships in search term homogeneity within and across document categories.

```
from yellowbrick.text import DispersionPlot
from yellowbrick.datasets import load_hobbies

corpus = load_hobbies()
text = [doc.split() for doc in corpus.data]
y = corpus.target

target_words = ['points', 'money', 'score', 'win', 'reduce']

visualizer = DispersionPlot(
    target_words,
    colormap="Accent",
    title="Lexical Dispersion Plot, Broken Down by Class"
)
visualizer.fit(text, y)
visualizer.show()
```

Quick Method

The same functionality above can be achieved with the associated quick method *dispersion*. This method will build the Dispersion Plot object with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from yellowbrick.text import DispersionPlot, dispersion
from yellowbrick.datasets import load_hobbies

# Load the text data
corpus = load_hobbies()

# Create a list of words from the corpus text
text = [doc.split() for doc in corpus.data]

# Choose words whose occurence in the text will be plotted
target_words = ['features', 'mobile', 'cooperative', 'competitive', 'combat', 'online']

# Create the visualizer and draw the plot
dispersion(target_words, text, colors=['olive'])
```



API Reference

Implementation of lexical dispersion for text visualization

class yellowbrick.text.dispersion.**DispersionPlot**(*search_terms*, *ax=None*, *colors=None*, *colormap=None*, *ignore_case=False*, *annotate_docs=False*, *labels=None*, ***kwargs*)

Bases: TextVisualizer

Lexical dispersion illustrates the homogeneity of a word (or set of words) across the documents of a corpus.

DispersionPlot allows for visualization of the lexical dispersion of words in a corpus. This plot illustrates with vertical lines the occurrences of one or more search terms throughout the corpus, noting how many words relative to the beginning of the corpus it appears. If the target vector of the corpus documents is provided, the points will be colored with respect to their document category, which allows for additional analysis of relationships in search term homogeneity within and across document categories. If annotation is requested, document boundaries will be displayed as vertical lines in the plot.

Parameters

search_terms
[list] A list of search terms whose dispersion across a corpus passed at fit should be visualized.

ax
[matplotlib axes, default: None] The axes to plot the figure on.

colors

[list or tuple of colors] Specify the colors for each individual class. Will override colormap if both are provided.

colormap
[string or matplotlib cmap] Qualitative colormap for discrete target

ignore_case
[boolean, default: False] Specify whether input will be case-sensitive.

annotate_docs
[boolean, default: False] Specify whether document boundaries will be displayed. Vertical lines are positioned at the end of each document.

labels
[list of strings] The names of the classes in the target, used to create a legend. Labels must match names of classes in sorted order.

kwargs
[dict] Pass any additional keyword arguments to the super class.

Attributes

self.classes_
[list] A list of strings representing the unique classes in the target in sorted order. If `y` is provided, these are extracted from `y`, unless a list of class labels is provided by the user on instantiation.

self.boundaries_
[list] A list of integers indicating the document boundaries with respect to word offsets.

self.indexed_words_
[list] A list of integers indicating the y position for each occurrence of each of the search terms.

self.word_categories_
[list] A list of strings indicating the corresponding document category of each search term occurrence.

`NULL_CLASS = None`

`draw`(*points*, ***kwargs*)

Called from the fit method, this method creates the canvas and draws the plot on it.

Parameters

kwargs: generic keyword arguments.

`finalize`(****kwargs*)

Prepares the figure for rendering by adding a title, axis labels, and managing the limits of the text labels. Adds a legend outside of the plot.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

fit(*X*, *y=None*, ***kwargs*)

The fit method is the primary drawing input for the dispersion visualization.

Parameters

X
[list or generator] Should be provided as a list of documents or a generator that yields a list of documents that contain a list of words in the order they appear in the document.

y
[ndarray or Series of length n] An optional array or series of target or class values for instances. If this is specified, then the points will be colored according to their class.

Returns

self
[instance] Returns the instance of the transformer/visualizer

Attributes

self.classes_
[list] A list of strings representing the unique classes in the target in sorted order. If `y` is provided, these are extracted from `y`, unless a list of class labels is provided by the user on instantiation.

self.indexed_words_
[list] A list of integers indicating the y position for each occurrence of each of the search terms.

self.word_categories_
[list] A list of strings indicating the corresponding document category of each search term occurrence.

yellowbrick.text.dispersion.**dispersion**(*search_terms*, *corpus*, *y=None*, *ax=None*, *colors=None*, *colormap=None*, *annotate_docs=False*, *ignore_case=False*, *labels=None*, *show=True*, ***kwargs*)

Displays lexical dispersion plot for words in a corpus

This helper function is a quick wrapper to utilize the DispersionPlot Visualizer for one-off analysis

Parameters

search_terms
[list] A list of words whose dispersion will be examined within a corpus

corpus
[list] Should be provided as a list of documents that contain a list of words in the order they appear in the document.

y
[ndarray or Series of length n] An optional array or series of target or class values for instances. If this is specified, then the points will be colored according to their class.

ax
[matplotlib axes, default: None] The axes to plot the figure on.

colors
[list or tuple of colors] Specify the colors for each individual class. Will override colormap if both are provided.

colormap
[string or matplotlib cmap] Qualitative colormap for discrete target

annotate_docs
[boolean, default: False] Specify whether document boundaries will be displayed. Vertical lines are positioned at the end of each document.

ignore_case
[boolean, default: False] Specify whether input will be case-sensitive.

labels
[list of strings] The names of the classes in the target, used to create a legend. Labels must match names of classes in sorted order.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Pass any additional keyword arguments to the super class.

Returns

viz: DispersionPlot
Returns the fitted, finalized visualizer

Word Correlation Plot

Word correlation illustrates the extent to which words or phrases co-appear across the documents in a corpus. This can be useful for understanding the relationships between known text features in a corpus with many documents. `WordCorrelationPlot` allows for the visualization of the document occurrence correlations between select words in a corpus. For a number of features n, the plot renders an n x n heatmap containing correlation values.

The correlation values are computed using the phi coefficient metric, which is a measure of the association between two binary variables. A value close to 1 or -1 indicates that the occurrences of the two features are highly positively or negatively correlated, while a value close to 0 indicates no relationship between the two features.

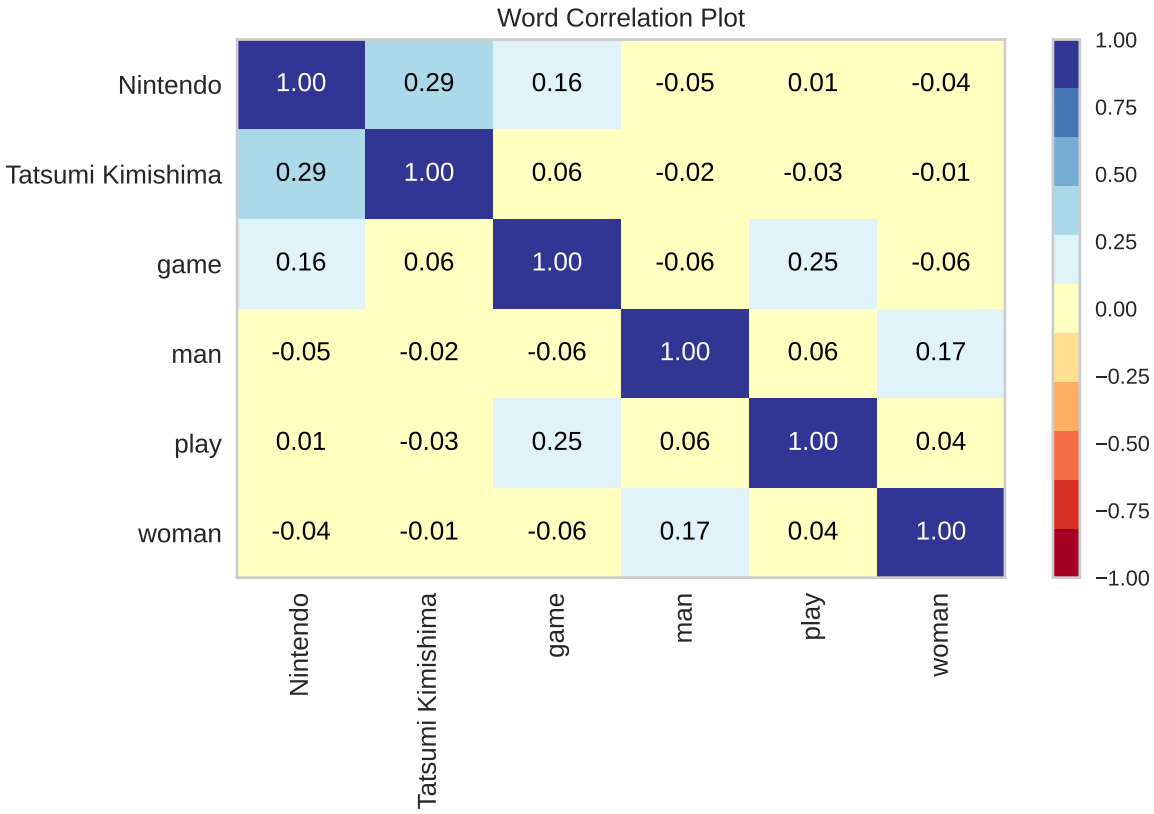
Visualizer	*WordCorrelationPlot*
Quick Method	*word_correlation()*
Models	Text Modeling
Workflow	Feature Engineering

```
from yellowbrick.datasets import load_hobbies
from yellowbrick.text.correlation import WordCorrelationPlot

# Load the text corpus
corpus = load_hobbies()

# Create the list of words to plot
words = ["Tatsumi Kimishima", "Nintendo", "game", "play", "man", "woman"]
```

(continues on next page)

(continued from previous page)

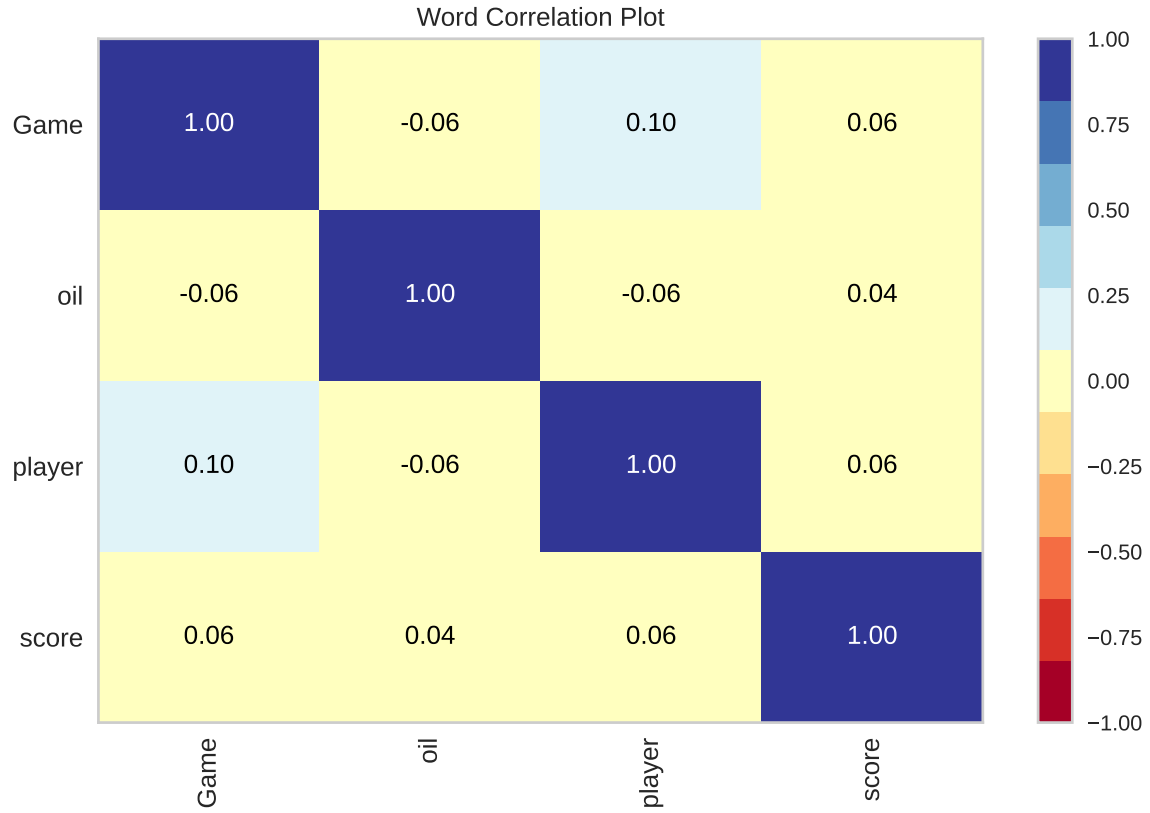
```
# Instantiate the visualizer and draw the plot
viz = WordCorrelationPlot(words)
viz.fit(corpus.data)
viz.show()
```



Quick Method

The same functionality above can be achieved with the associated quick method *word_correlation*. This method will build the Word Correlation Plot object with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from yellowbrick.datasets import load_hobbies
from yellowbrick.text.correlation import word_correlation

# Load the text corpus
corpus = load_hobbies()

# Create the list of words to plot
words = ["Game", "player", "score", "oil"]

# Draw the plot
word_correlation(words, corpus.data)
```



API Reference

Implementation of word correlation for text visualization.

class yellowbrick.text.correlation.**WordCorrelationPlot**(*words*, *ignore_case=False*, *ax=None*, *cmap='RdYlBu'*, *colorbar=True*, *fontsize=None*, ***kwargs*)

Bases: TextVisualizer

Word correlation illustrates the extent to which words in a corpus appear in the same documents.

WordCorrelationPlot visualizes the binary correlation between words across documents as a heatmap. The correlation is defined using the mean square contingency coefficient (phi-coefficient) between any two words m and n. The coefficient is a value between -1 and 1, inclusive. A value close to 1 or -1 indicates strong positive or negative correlation between m and n, while a value close to 0 indicates little or no correlation. The constructor takes one required argument, which is the list of words or n-grams to be plotted.

Parameters

words
[list of str] The list of words or n-grams to be plotted. The words must be present in the provided corpus on fit().

ignore_case
[bool, default: False] If True, all words will be converted to lowercase before processing.

ax
[matplotlib Axes, default: None] The axes to plot the figure on.

cmap
[str or cmap, default: "RdYlBu"] Colormap to use for the heatmap.

colorbar
[bool, default: True] If True, a colorbar will be added to the heatmap.

fontsize
[int, default: None] Font size to use for the labels on the axes.

kwargs
[dict] Pass any additional keyword arguments to the super class.

Attributes

self.doc_term_matrix_
[array of shape (n_docs, n_features)] The computed sparse document-term matrix containing binary values indicating if a word is present in a document.

self.num_docs_
[int] The number of observed documents in the corpus.

self.vocab_
[dict] A dictionary mapping words to their indices in the document-term matrix.

self.num_features_
[int] The number of features (word labels) in the resulting plot.

self.correlation_matrix_
[ndarray of shape (n_features, n_features)] The computed matrix containing the phi-coefficients between all features.

`draw`(*X*)

Called from the fit() method, this metod draws the heatmap on the figure using the computed correlation matrix.

`finalize`()

Prepares the figure for rendering by adding the title. This method is usually called from show() and not directly by the user.

`fit`(*X*, *y=None*)

The fit method is the primary drawing input for the word correlation visualization.

Parameters

X
[list of str or generator] Should be provided as a list of strings or a generator yielding strings that represent the documents in the corpus.

y
[None] Labels are not used for the word correlation visualization.

Returns

self: instance
Returns the instance of the transformer/visualizer.

Attributes

self.doc_term_matrix_
[array of shape (n_docs, n_features)] The computed sparse document-term matrix containing binary values indicating if a word is present in a document.

self.num_docs_
[int] The number of observed documents in the corpus.

self.vocab_
[dict] A dictionary mapping words to their indices in the document-term matrix.

self.num_features_
[int] The number of features (word labels) in the resulting plot.

self.correlation_matrix_
[ndarray of shape (n_features, n_features)] The computed matrix containing the phi-coefficients between all features.

yellowbrick.text.correlation.**word_correlation**(*words*, *corpus*, *ignore_case=True*, *ax=None*, *cmap='RdYlBu'*, *show=True*, *colorbar=True*, *fontsize=None*, ***kwargs*)

Word Correlation

Displays the binary correlation between the given words across the documents in a corpus. For a list of words with length n, this produces an n x n heatmap of correlation values in the range [-1, 1].

Parameters

words
[list of str] The corpus words to display in the heatmap.

corpus
[list of str or generator] The corpus as a list of documents or a generator yielding documents.

ignore_case
[bool, default: True] If True, all words will be converted to lowercase before proessing.

ax
[matplotlib axes, default: None] The axes to plot the figure on.

cmap
[str, default: "RdYlBu"] Colormap to use for the heatmap.

show
[bool, default: True] If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

colorbar
[bool, default: True] If True, adds a colorbar to the figure.

fontsize
[int, default: None] If not None, sets the font size of the labels.

PosTag Visualization

Parts of speech (e.g. verbs, nouns, prepositions, adjectives) indicate how a word is functioning within the context of a sentence. In English as in many other languages, a single word can function in multiple ways. Part-of-speech tagging lets us encode information not only about a word's definition, but also its use in context (for example the words "ship" and "shop" can be either a verb or a noun, depending on the context).

The `PosTagVisualizer` is intended to support grammar-based feature extraction techniques for machine learning workflows that require natural language processing. The visualizer can either read in a corpus that has already been sentence- and word-segmented, and tagged, or perform this tagging automatically by specifying the parser to use (nltk or spacy). The visualizer creates a bar chart to visualize the relative proportions of different parts-of-speech in a corpus.

Visualizer	*PosTagVisualizer*
Quick Method	*postag()*
Models	Classification, Regression
Workflow	Feature Engineering

Note: The `PosTagVisualizer` currently works with both Penn-Treebank (e.g. via NLTK) and Universal Dependencies (e.g. via SpaCy)-tagged corpora. This expects either raw text, or corpora that have already been tagged which take the form of a list of (document) lists of (sentence) lists of `(token, tag)` tuples, as in the example below.

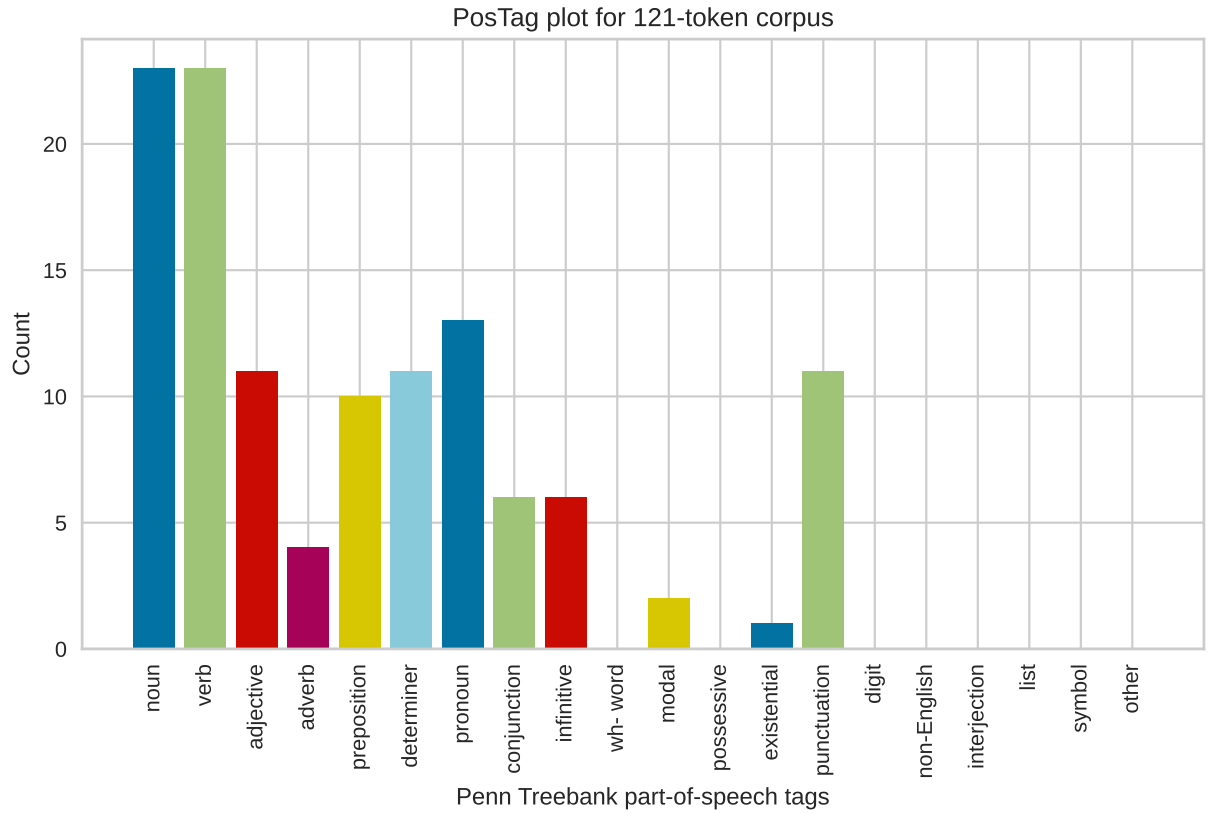
Penn Treebank Tags

```
from yellowbrick.text import PosTagVisualizer


tagged_stanzas = [
    [
        [
            ('Whose', 'JJ'),('woods', 'NNS'),('these', 'DT'),
            ('are', 'VBP'),('I', 'PRP'),('think', 'VBP'),('I', 'PRP'),
            ('know', 'VBP'),('.', '.')
            ],
        [
            ('His', 'PRP$'),('house', 'NN'),('is', 'VBZ'),('in', 'IN'),
            ('the', 'DT'),('village', 'NN'),('though', 'IN'),(';', ':'),
            ('He', 'PRP'),('will', 'MD'),('not', 'RB'),('see', 'VB'),
            ('me', 'PRP'),('stopping', 'VBG'), ('here', 'RB'),('To', 'TO'),
            ('watch', 'VB'),('his', 'PRP$'),('woods', 'NNS'),('fill', 'VB'),
            ('up', 'RP'),('with', 'IN'),('snow', 'NNS'),('.', '.')
            ]
        ],
    [
        [
            ('My', 'PRP$'),('little', 'JJ'),('horse', 'NN'),('must', 'MD'),
            ('think', 'VB'),('it', 'PRP'),('queer', 'JJR'),('To', 'TO'),
            ('stop', 'VB'),('without', 'IN'),('a', 'DT'),('farmhouse', 'NN'),
            ('near', 'IN'),('Between', 'NNP'),('the', 'DT'),('woods', 'NNS'),
            ('and', 'CC'),('frozen', 'JJ'),('lake', 'VB'),('The', 'DT'),
            ('darkest', 'JJS'),('evening', 'NN'),('of', 'IN'),('the', 'DT'),
            ('year', 'NN'),('.', '.')
            ]
        ],
    [
        [
            ('He', 'PRP'),('gives', 'VBZ'),('his', 'PRP$'),('harness', 'NN'),
            ('bells', 'VBZ'),('a', 'DT'),('shake', 'NN'),('To', 'TO'),
            ('ask', 'VB'),('if', 'IN'),('there', 'EX'),('is', 'VBZ'),
            ('some', 'DT'),('mistake', 'NN'),('.', '.')
            ],
        [
```

(continues on next page)

(continued from previous page)

```
            ('The', 'DT'),('only', 'JJ'),('other', 'JJ'),('sound', 'NN'),
            (''', 'NNP'),('s', 'VBZ'),('the', 'DT'),('sweep', 'NN'),
            ('Of', 'IN'),('easy', 'JJ'),('wind', 'NN'),('and', 'CC'),
            ('downy', 'JJ'),('flake', 'NN'),('.', '.')
            ]
        ],
    [
        [
            ('The', 'DT'),('woods', 'NNS'),('are', 'VBP'),('lovely', 'RB'),
            (',', ','),('dark', 'JJ'),('and', 'CC'),('deep', 'JJ'),(',', ','),
            ('But', 'CC'),('I', 'PRP'),('have', 'VBP'),('promises', 'NNS'),
            ('to', 'TO'),('keep', 'VB'),(',', ','),('And', 'CC'),('miles', 'NNS'),
            ('to', 'TO'),('go', 'VB'),('before', 'IN'),('I', 'PRP'),
            ('sleep', 'VBP'),(',', ','),('And', 'CC'),('miles', 'NNS'),
            ('to', 'TO'),('go', 'VB'),('before', 'IN'),('I', 'PRP'),
            ('sleep', 'VBP'),('.', '.')
            ]
    ]
]

# Create the visualizer, fit, score, and show it
viz = PosTagVisualizer()
viz.fit(tagged_stanzas)
viz.show()
```



Universal Dependencies Tags

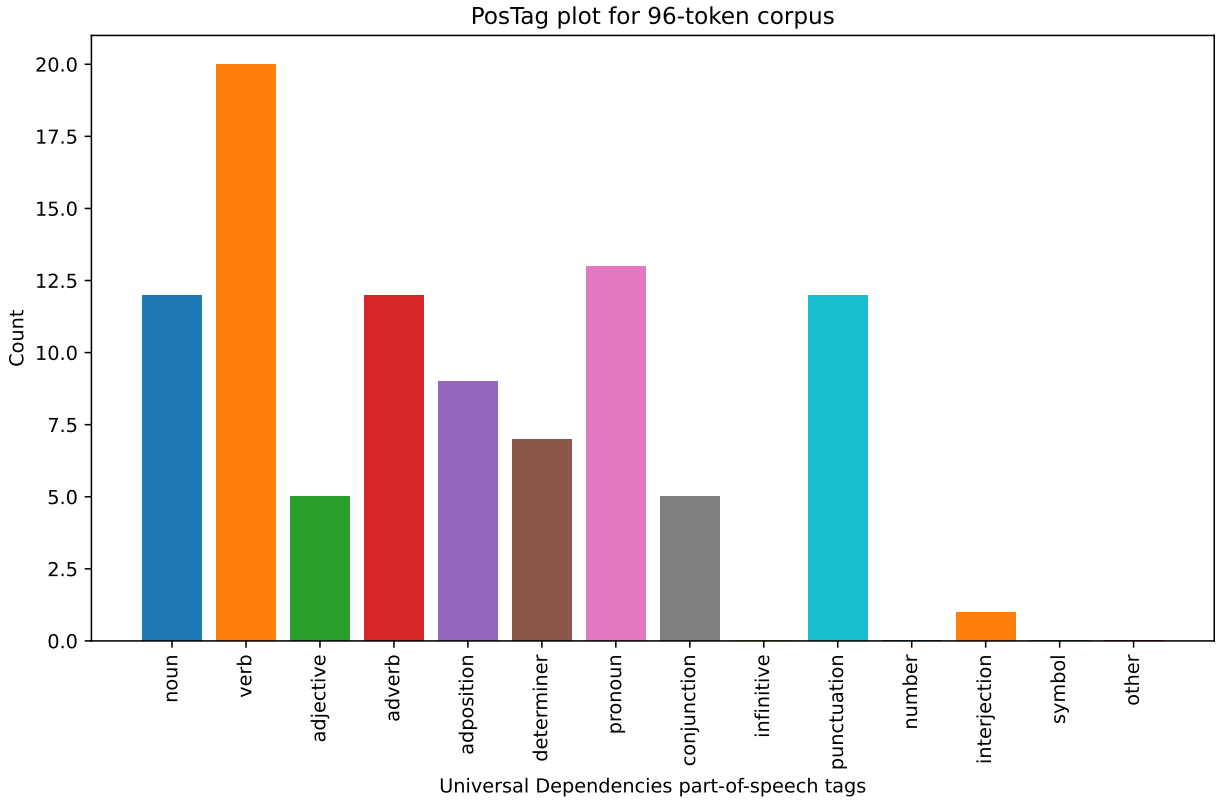
Libraries like SpaCy use tags from the Universal Dependencies (UD) framework. The `PosTagVisualizer` can also be used with text tagged using this framework by specifying the `tagset` keyword as "universal" on instantiation.

```
from yellowbrick.text import PosTagVisualizer

tagged_speech = [
    [
        [
            ('In', 'ADP'),('all', 'DET'),('honesty', 'NOUN'),(',', 'PUNCT'),
            ('I', 'PRON'),('said', 'VERB'),('yes', 'INTJ'),('to', 'ADP'),
            ('the', 'DET'),('fear', 'NOUN'),('of', 'ADP'),('being', 'VERB'),
            ('on', 'ADP'),('this', 'DET'),('stage', 'NOUN'),('tonight', 'NOUN'),
            ('because', 'ADP'),('I', 'PRON'),('wanted', 'VERB'),('to', 'PART'),
            ('be', 'VERB'),('here', 'ADV'),(',', 'PUNCT'),('to', 'PART'),
            ('look', 'VERB'),('out', 'PART'),('into', 'ADP'),('this', 'DET'),
            ('audience', 'NOUN'),(',', 'PUNCT'),('and', 'CCONJ'),
            ('witness', 'VERB'),('this', 'DET'),('moment', 'NOUN'),('of', 'ADP'),
            ('change', 'NOUN')
            ],
        [
            ('and', 'CCONJ'),('I', 'PRON'),("'m", 'VERB'),('not', 'ADV'),
            ('fooling', 'VERB'),('myself', 'PRON'),('.', 'PUNCT')
            ],
        [
            ('I', 'PRON'),("'m", 'VERB'),('not', 'ADV'),('fooling', 'VERB'),
            ('myself', 'PRON'),('.', 'PUNCT')
            ],
        [
            ('Next', 'ADJ'),('year', 'NOUN'),('could', 'VERB'),('be', 'VERB'),
            ('different', 'ADJ'),('.', 'PUNCT')
            ],
        [
            ('It', 'PRON'),('probably', 'ADV'),('will', 'VERB'),('be', 'VERB'),
            (',', 'PUNCT'),('but', 'CCONJ'),('right', 'ADV'),('now', 'ADV'),
            ('this', 'DET'),('moment', 'NOUN'),('is', 'VERB'),('real', 'ADJ'),
            ('.', 'PUNCT')
            ],
        [
            ('Trust', 'VERB'),('me', 'PRON'),(',', 'PUNCT'),('it', 'PRON'),
            ('is', 'VERB'),('real', 'ADJ'),('because', 'ADP'),('I', 'PRON'),
            ('see', 'VERB'),('you', 'PRON')
            ],
        [
            ('and', 'CCONJ'), ('I', 'PRON'), ('see', 'VERB'), ('you', 'PRON')
            ],
        [
            ('-', 'PUNCT')
            ],
        [
            ('all', 'ADJ'),('these', 'DET'),('faces', 'NOUN'),('of', 'ADP'),
            ('change', 'NOUN')
```

(continues on next page)

(continued from previous page)

```
            ],
        [
            ('-', 'PUNCT'),('and', 'CCONJ'),('now', 'ADV'),('so', 'ADV'),
            ('will', 'VERB'),('everyone', 'NOUN'),('else', 'ADV'), ('.', 'PUNCT')
            ]
    ]
]

# Create the visualizer, fit, score, and show it
viz = PosTagVisualizer(tagset="universal")
viz.fit(tagged_speech)
viz.show()
```



Quick Method

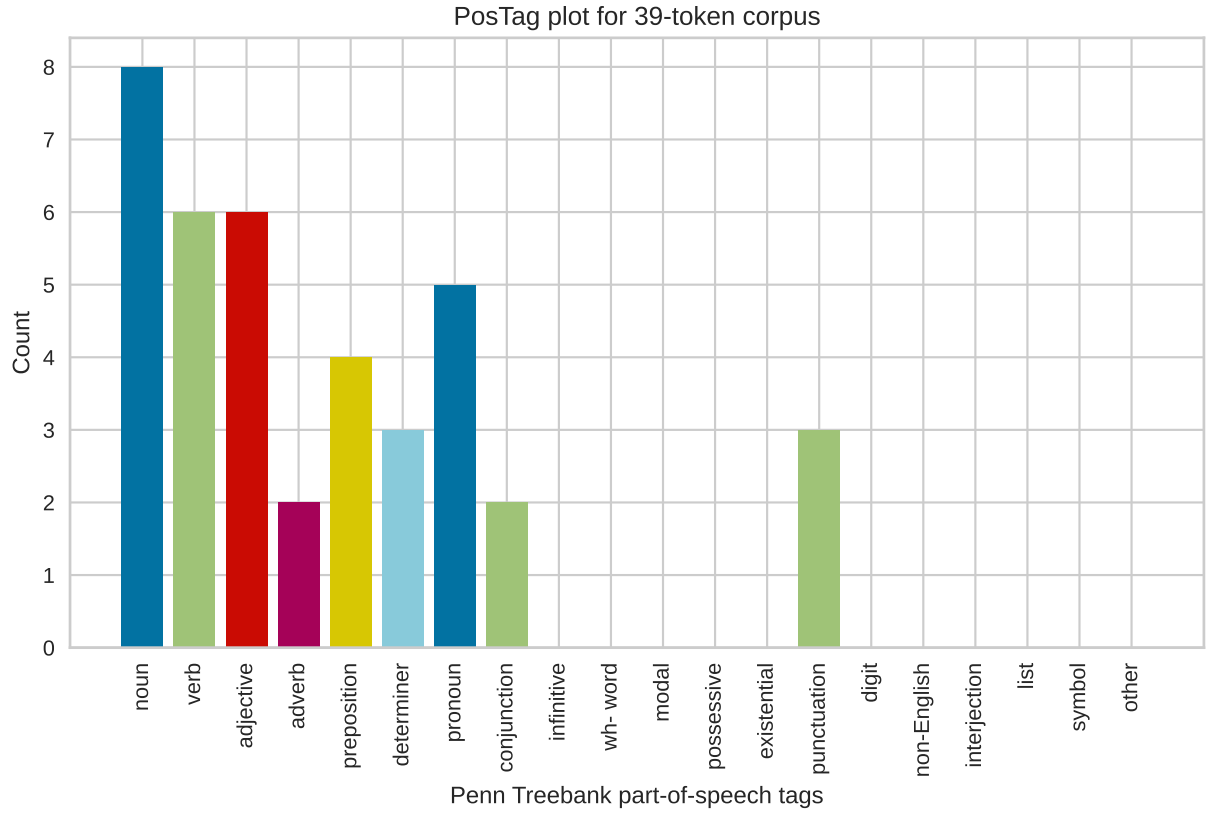
The same functionality above can be achieved with the associated quick method `postag`. This method will build the `PosTagVisualizer` object with the associated arguments, fit it, then (optionally) immediately show the visualization.

```
from yellowbrick.text.postag import postag

machado = [
    [
        [
```

(continues on next page)

(continued from previous page)

```
            ('Last', 'JJ'), ('night', 'NN'), ('as', 'IN'), ('I', 'PRP'),
            ('was', 'VBD'), ('sleeping', 'VBG'), (',', ','), ('I', 'PRP'),
            ('dreamt', 'VBP'), ('-', 'RB'), ('marvelous', 'JJ'), ('error', 'NN'),
            ('!-', 'IN'), ('that', 'DT'), ('I', 'PRP'), ('had', 'VBD'), ('a', 'DT'),
            ('beehive', 'NN'), ('here', 'RB'), ('inside', 'IN'), ('my', 'PRP$'),
            ('heart', 'NN'), ('.', '.')
            ],
        [
            ('And', 'CC'), ('the', 'DT'), ('golden', 'JJ'), ('bees', 'NNS'),
            ('were', 'VBD'), ('making', 'VBG'), ('white', 'JJ'), ('combs', 'NNS'),
            ('and', 'CC'), ('sweet', 'JJ'), ('honey', 'NN'), ('from', 'IN'),
            ('my', 'PRP$'), ('old', 'JJ'), ('failures', 'NNS'), ('.', '.')
            ]
    ]
]

# Create the visualizer, fit, score, and show it
postag(machado)
```



Part of Speech Tags

Penn-Treebank Tag	Description	Universal Tag	Description
CC	Coordinating conjunction	ADJ	adjective
CD	Cardinal number	ADP	adposition
DT	Determiner	ADV	adverb
EX	Existential *there*	AUX	auxiliary
FW	Foreign word	CONJ	conjunction
IN	Preposition or subordinating conjunction	CCONJ	coordinating conjunction
JJ	Adjective	DET	determiner
JJR	Adjective, comparative	INTJ	interjection
JJS	Adjective, superlative	NOUN	noun
LS	List item marker	NUM	numeral
MD	Modal	PART	particle
NN	Noun, singular or mass	PRON	pronoun
NNS	Noun, plural	PROPN	proper noun
NNP	Proper noun, singular	PUNCT	punctuation
NNPS	Proper noun, plural	SCONJ	subordinating conjunction
PDT	Predeterminer	SYM	symbol
POS	Possessive ending	VERB	verb
PRP	Personal pronoun	X	other
PRP$	Possessive pronoun	SPACE	space
RB	Adverb		
RBR	Adverb, comparative		
RBS	Adverb, superlative		
RP	Particle		
SYM	Symbol		
TO	*to*		
UH	Interjection		
VB	Verb, base form		
VBD	Verb, past tense		
VBG	Verb, gerund or present participle		
VBN	Verb, past participle		
VBP	Verb, non-3rd person singular present		
VBZ	Verb, 3rd person singular present		
WDT	Wh-determiner		
WP	Wh-pronoun		
WP$	Possessive wn-pronoun		
WRB	Wh-adverb		

Parsing raw text automatically

The `PosTagVisualizer` can also be used with untagged text by using the `parse` keyword on instantiation. The keyword to parse indicates which natural language processing library to use. To use `spacy`:

```
untagged_speech = u'Whose woods these are I think I know'

# Create the visualizer, fit, score, and show it
viz = PosTagVisualizer(parser='spacy')
viz.fit(untagged_speech)
```

(continues on next page)

(continued from previous page)

```
viz.show()
```

Or, using the `nltk` parser.

```
untagged_speech = u'Whose woods these are I think I know'

# Create the visualizer, fit, score, and show it
viz = PosTagVisualizer(parser='nltk')
viz.fit(untagged_speech)
viz.show()
```

Note: To use either of these parsers, either *nltk* or *spacy* must already be installed in your environment.

You can also change the tagger used. For example, using *nltk* you can select either *word* (default):

```
untagged_speech = u'Whose woods these are I think I know'

# Create the visualizer, fit, score, and show it
viz = PosTagVisualizer(parser='nltk_word')
viz.fit(untagged_speech)
viz.show()
```

Or using *wordpunct*.

```
untagged_speech = u'Whose woods these are I think I know'

# Create the visualizer, fit, score, and show it
viz = PosTagVisualizer(parser='nltk_wordpunct')
viz.fit(untagged_speech)
viz.show()
```

API Reference

Implementation of part-of-speech visualization for text, enabling the user to visualize a single document or small subset of documents.

class yellowbrick.text.postag.**PosTagVisualizer**(*ax=None*, *tagset='penn_treebank'*, *colormap=None*, *colors=None*, *frequency=False*, *stack=False*, *parser=None*, ***kwargs*)

Bases: `TextVisualizer`

Parts of speech (e.g. verbs, nouns, prepositions, adjectives) indicate how a word is functioning within the context of a sentence. In English as in many other languages, a single word can function in multiple ways. Part-of-speech tagging lets us encode information not only about a word's definition, but also its use in context (for example the words "ship" and "shop" can be either a verb or a noun, depending on the context).

The PosTagVisualizer creates a bar chart to visualize the relative proportions of different parts-of-speech in a corpus.

Note that the PosTagVisualizer requires documents to already be part-of-speech tagged; the visualizer expects the corpus to come in the form of a list of (document) lists of (sentence) lists of (tag, token) tuples.

Parameters

ax
[matplotlib axes] The axes to plot the figure on.

tagset: string
The tagset that was used to perform part-of-speech tagging. Either "penn_treebank" or "universal", defaults to "penn_treebank". Use "universal" if corpus has been tagged using SpaCy.

colors
[list or tuple of strings] Specify the colors for each individual part-of-speech. Will override colormap if both are provided.

colormap
[string or matplotlib cmap] Specify a colormap to color the parts-of-speech.

frequency: bool {True, False}, default: False
If set to True, part-of-speech tags will be plotted according to frequency, from most to least frequent.

stack
[bool {True, False}, default][False] Plot the PosTag frequency chart as a per-class stacked bar chart. Note that fit() requires y for this visualization.

parser
[string or None, default: None] If set to a string, string must be in the form of 'parser_tagger' or 'parser' to use defaults (for spacy this is 'en_core_web_sm', for nltk this is 'word'). The 'parser' argument is one of the accepted parsing libraries. Currently 'nltk' and 'spacy' are the only accepted libraries. NLTK or SpaCy must be installed into your environment. 'tagger' is the tagset to use. For example 'nltk_wordpunct' would use the NLTK library with 'wordpunct' tagset. Or 'spacy_en_core_web_sm' would use SpaCy with the 'en_core_web_sm' tagset.

kwargs
[dict] Pass any additional keyword arguments to the PosTagVisualizer.

Examples

```
>>> viz = PosTagVisualizer()
>>> viz.fit(X)
>>> viz.show()
```

Attributes

pos_tag_counts_: dict
Mapping of part-of-speech tags to counts.

`draw`(***kwargs*)

Called from the fit method, this method creates the canvas and draws the part-of-speech tag mapping as a bar chart.

Parameters

kwargs: dict
generic keyword arguments.

Returns

ax
[matplotlib axes] Axes on which the PosTagVisualizer was drawn.

finalize(***kwargs*)

Finalize the plot with ticks, labels, and title

Parameters

kwargs: dict
generic keyword arguments.

fit(*X*, *y=None*, ***kwargs*)

Fits the corpus to the appropriate tag map. Text documents must be tokenized & tagged before passing to fit if the 'parse' argument has not been specified at initialization. Otherwise X can be a raw text ready to be parsed.

Parameters

X
[list or generator or str (raw text)] Should be provided as a list of documents or a generator that yields a list of documents that contain a list of sentences that contain (token, tag) tuples. If X is a string, the 'parse' argument should be specified as 'nltk' or 'spacy' in order to parse the raw documents.

y
[ndarray or Series of length n] An optional array of target values that are ignored by the visualizer.

kwargs
[dict] Pass generic arguments to the drawing method

Returns

self
[instance] Returns the instance of the transformer/visualizer

parse_nltk(*X*)

Tag a corpora using NLTK tagging (Penn-Treebank) to produce a generator of tagged documents in the form of a list of (document) lists of (sentence) lists of (token, tag) tuples.

Parameters

X
[str (raw text) or list of paragraphs (containing str)]

parse_spacy(*X*)

Tag a corpora using SpaCy tagging (Universal Dependencies) to produce a generator of tagged documents in the form of a list of (document) lists of (sentence) lists of (token, tag) tuples.

Parameters

X
[str (raw text) or list of paragraphs (containing str)]

property parser

show(*outpath=None*, ***kwargs*)

Makes the magic happen and a visualizer appear! You can pass in a path to save the figure to disk with various backends, or you can call it with no arguments to show the figure either in a notebook or in a GUI window that pops up on screen.

Parameters

outpath: string, default: None
path or None. Save figure to disk or if None show in window

clear_figure: boolean, default: False
When True, this flag clears the figure after saving to file or showing on screen. This is useful when making consecutive plots.

kwargs: dict
generic keyword arguments.

Notes

Developers of visualizers don't usually override show, as it is primarily called by the user to render the visualization.

yellowbrick.text.postag.**postag**(*X*, *y=None*, *ax=None*, *tagset='penn_treebank'*, *colormap=None*, *colors=None*, *frequency=False*, *stack=False*, *parser=None*, *show=True*, ***kwargs*)

Display a barchart with the counts of different parts of speech in X, which consists of a part-of-speech-tagged corpus, which the visualizer expects to be a list of lists of lists of (token, tag) tuples.

Parameters

X
[list or generator] Should be provided as a list of documents or a generator that yields a list of documents that contain a list of sentences that contain (token, tag) tuples.

y
[ndarray or Series of length n] An optional array of target values that are ignored by the visualizer.

ax
[matplotlib axes] The axes to plot the figure on.

tagset: string
The tagset that was used to perform part-of-speech tagging. Either "penn_treebank" or "universal", defaults to "penn_treebank". Use "universal" if corpus has been tagged using SpaCy.

colors
[list or tuple of colors] Specify the colors for each individual part-of-speech.

colormap
[string or matplotlib cmap] Specify a colormap to color the parts-of-speech.

frequency: bool {True, False}, default: False
If set to True, part-of-speech tags will be plotted according to frequency, from most to least frequent.

stack
[bool {True, False}, default][False] Plot the PosTag frequency chart as a per-class stacked bar chart. Note that fit() requires y for this visualization.

parser
[string or None, default: None] If set to a string, string must be in the form of 'parser_tagger' or 'parser' to use defaults (for spacy this is 'en_core_web_sm', for nltk this is 'word'). The 'parser' argument is one of the accepted parsing libraries. Currently 'nltk' and 'spacy' are the only accepted libraries. NLTK or SpaCy must be installed into your environment. 'tagger' is the tagset to use. For example 'nltk_wordpunct' would use the NLTK library with 'wordpunct' tagset. Or 'spacy_en_core_web_sm' would use SpaCy with the 'en_core_web_sm' tagset.

show: bool, default: True
If True, calls `show()`, which in turn calls `plt.show()` however you cannot call `plt.savefig` from this signature, nor `clear_figure`. If False, simply calls `finalize()`

kwargs
[dict] Pass any additional keyword arguments to the PosTagVisualizer.

Returns

visualizer: PosTagVisualizer
Returns the fitted, finalized visualizer

8.3.10 Contrib and Third-Party Libraries

Yellowbrick's primary dependencies are scikit-learn and matplotlib, however the data science landscape in Python is large and there are many opportunties to use Yellowbrick with other machine learning and data analysis frameworks. The `yellowbrick.contrib` package contains several methods and mechanisms to support non-scikit-learn machine learning as well as extra tools and experimental visualizers that are outside of core support or are still in development.

Note: If you're interested in using a non-scikit-learn estimator with Yellowbrick, please see the *Using Third-Party Estimators* documentation. If the wrapper doesn't work out of the box, we welcome contributions to this module to include other libraries!

The following contrib packages are currently available:

Using Third-Party Estimators

Many machine learning libraries implement the scikit-learn estimator API to easily integrate alternative optimization or decision methods into a data science workflow. Because of this, it seems like it should be simple to drop in a non-scikit-learn estimator into a Yellowbrick visualizer, and in principle, it is. However, the reality is a bit more complicated.

Yellowbrick visualizers often utilize more than just the method interface of estimators (e.g. `fit()` and `predict()`), relying on the learned attributes (object properties with a single underscore suffix, e.g. `coef_`). The issue is that when a third-party estimator does not expose these attributes, truly gnarly exceptions and tracebacks occur. Yellowbrick is meant to aid machine learning diagnostics reasoning, therefore instead of just allowing drop-in functionality that may cause confusion, we've created a wrapper functionality that is a bit kinder with it's messaging.

But first, an example.

```
# Import the wrap function and a Yellowbrick visualizer
from yellowbrick.contrib.wrapper import wrap
from yellowbrick.model_selection import feature_importances

# Instantiate the third party estimator and wrap it, optionally fitting it
model = wrap(ThirdPartyEstimator())
model.fit(X_train, y_train)

# Use the visualizer
oz = feature_importances(model, X_test, y_test, is_fitted=True)
```

The `wrap` function initializes the third party model as a `ContribEstimator`, which passes through all functionality to the underlying estimator, however if an error occurs, the exception that will be raised looks like:

```
yellowbrick.exceptions.YellowbrickAttributeError: estimator is missing the 'fit'
attribute, which is required for this visualizer - please see the third party
estimators documentation.
```

Some estimators are required to pass type checking, for example the estimator must be a classifier, regressor, clusterer, density estimator, or outlier detector. A second argument can be passed to the `wrap` function declaring the type of estimator:

```
from yellowbrick.classifier import precision_recall_curve
from yellowbrick.contrib.wrapper import wrap, CLASSIFIER

model = wrap(ThirdPartyClassifier(), CLASSIFIER)
precision_recall_curve(model, X, y)
```

Or you can simply use the wrap helper functions of the specific type:

```
from yellowbrick.contrib.wrapper import regressor, classifier, clusterer
from yellowbrick.regressor import prediction_error
from yellowbrick.classifier import classification_report
from yellowbrick.cluster import intercluster_distance

reg = regressor(ThirdPartyRegressor())
prediction_error(reg, X, y)

clf = classifier(ThirdPartyClassifier())
classification_report(clf, X, y)

ctr = clusterer(ThirdPartyClusterer())
intercluster_distance(ctr, X)
```

So what should you do if a required attribute is missing from your estimator? The simplest and quickest thing to do is to subclass `ContribEstimator` and add the required functionality.

```
from yellowbrick.contrib.wrapper import ContribEstimator, CLASSIFIER

class MyWrapper(ContribEstimator):

    _estimator_type = CLASSIFIER

    @property
    def feature_importances_(self):
        return self.estimator.tree_feature_importances()


model = MyWrapper(ThirdPartyEstimator()
feature_importances(model, X, y)
```

This is certainly less than ideal - but we'd welcome a contrib PR to add more native functionality to Yellowbrick!

Tested Libraries

The following libraries have been tested with the Yellowbrick wrapper.

- xgboost: both the `XGBRFRegressor` and `XGBRFClassifier` have been tested with Yellowbrick both with and without the wrapper functionality.
- CatBoost: the `CatBoostClassifier` has been tested with the `ClassificationReport` visualizer.

The following libraries have been partially tested and will likely work without too much additional effort:

- cuML: it is likely that clustering, classification, and regression cuML estimators will work with Yellowbrick visualizers. However, the cuDF datasets have not been tested with Yellowbrick.
- Spark MLlib: The Spark DataFrame API and estimators should work with Yellowbrick visualizers in a local notebook context after collection.

Note: If you have used a Python machine learning library not listed here with Yellowbrick, please let us know - we'd love to add it to the list! Also if you're using a library that is not wholly compatible, please open an issue so that we can explore how to integrate it with the `yellowbrick.contrib` module!

API Reference

Wrapper for third-party estimators that implement the sklearn API but do not directly subclass the `sklearn.base.BaseEstimator` class. This method is a quick way to get other estimators into Yellowbrick, while avoiding weird errors and issues.

`class yellowbrick.contrib.wrapper.`**`ContribEstimator`**(*estimator*, *estimator_type=None*)

Bases: `object`

Wraps a third party estimator that implements the sckit-learn API and therefore could be used with Yellowbrick but doesn't subclass `BaseEstimator`. Since there are a number of pitfalls, this object provides sensible errors and warnings rather than completely blowing up, allowing contrib users to identify issues and fix them, smoothing the path to getting third party estimators into the Yellowbrick ecosystem.

Parameters

estimator
[object] The non-sklearn estimator to wrap and use for Visualizers

estimator_type
[str, optional] One of "classifier", "regressor", "clusterer", "DensityEstimator", or "outlier_detector" that allows the contrib estimator to pass the scikit-learn `is_classifier`, etc. functions. If not specified, the _estimator_type attr is passed through to the underlying estimator.

`yellowbrick.contrib.wrapper.`**`classifier`**(*estimator*)

Wrap a third-party classifier to make it available to Yellowbrick visualizers.

Parameters

estimator
[object] The non-sklearn classifier to wrap and use for Visualizers

`yellowbrick.contrib.wrapper.`**`clusterer`**(*estimator*)

Wrap a third-party clusterer to make it available to Yellowbrick visualizers.

Parameters

estimator
[object] The non-sklearn clusterer to wrap and use for Visualizers

`yellowbrick.contrib.wrapper.regressor(estimator)`

Wrap a third-party regressor to make it available to Yellowbrick visualizers.

Parameters

estimator
[object] The non-sklearn regressor to wrap and use for Visualizers

`yellowbrick.contrib.wrapper.wrap(estimator, estimator_type=None)`

Wrap a third-party estimator that implements portions of the scikit-learn API to make it available to Yellowbrick visualizers. If the Yellowbrick visualizer cannot succeed, then a sensible error is raised instead.

Parameters

estimator
[object] The non-sklearn estimator to wrap and use for Visualizers

estimator_type
[str, optional] One of "classifier", "regressor", "clusterer", "DensityEstimator", or "outlier_detector" that allows the contrib estimator to pass the scikit-learn `is_classifier`, etc. functions. If not specified, the _estimator_type attr is passed through to the underlying estimator.

PrePredict Estimators

Occassionally it is useful to be able to use predictions made during an inferencing workflow that does not involve Yellowbrick, for example when the inferencing process requires extra compute resources such as a cluster or when the model takes a very long time to train and inference. In other instances there are models that Yellowbrick simply does not support, even with the *third-party estimator wrapper* or the results may have been collected from some source out of your control.

Some Yellowbrick visualizers are still able to create visual diagnostics with predictions already made using the contrib library `PrePredict` estimator, which is a simple wrapper around some data and an estimator type. Although not quite as straight forward as a scikit-learn metric in the form `metric(y_true, y_pred)`, this estimator allows Yellowbrick to be used in the cases described above, an example is below:

```
# Import the prepredict estimator and a Yellowbrick visualizer
from yellowbrick.contrib.prepredict import PrePredict, CLASSIFIER
from yellowbrick.classifier import classification_report

# Instantiate the estimator with the pre-predicted data
model = PrePredict(y_pred, CLASSIFIER)

# Use the visualizer, setting X to None since it is not required
oz = classification_report(model, None, y_test)
oz.show()
```

Warning: Many Yellowbrick visualizers inspect the estimator for learned attributes in order to deliver rich diagnostics. You may run into visualizers that cannot use the prepredict method, or you can manually set attributes on the `PrePredict` estimator with the learned attributes the visualizer requires.

In the case where you've saved pre-predicted data from disk, the `PrePredict` estimator can load it using `np.load`. A full workflow is described below:

```
# Phase one: fit your estimator, make inferences, and save the inferences to disk
np.save("y_pred.npy", y_pred)

# Import the prepredict estimator and a Yellowbrick visualizer
from yellowbrick.contrib.prepredict import PrePredict, REGRESSOR
from yellowbrick.regressor import prediction_error

# Instantiate the estimator with the pre-predicted data and pass a path to where
# the data has been saved on disk.
model = PrePredict("y_pred.npy", REGRESSOR)

# Use the visualizer, setting X to None since it is not required
oz = prediction_error(model, X_test, y_test)
oz.show()
```

The `PrePredict` estimator can use a callable function to return pre-predicted data, a `str`, file-like object, or `pathlib.Path` to load from disk using `np.load`, otherwise it simply returns the data it wraps. See the API reference for more details.

API Reference

PrePredict estimator allows Yellowbrick to work with results produced by an estimator prior to the visual diagnostic workflow, particularly for inferences that require extensive time or compute resources.

class yellowbrick.contrib.prepredict.**PrePredict**(*data*, *estimator_type=None*)

Bases: BaseEstimator

The Passthrough estimator allows users to specify pre-predicted results to Yellowbrick without the need to input the original estimator. Note that Yellowbrick often uses the learned attributes of the estimator to produce rich visual diagnostics, so this estimator may not work for all Yellowbrick visualizers.

The passthrough estimator can accept data either in memory as a numpy array or it can accept a string, which it interprets as a path on disk to load the data from.

Currently passthrough does not support predict_proba or decision_function methods, which it could if it was passed predicted data as 2D array instead of a 1D array.

Parameters

data
[array-like, func, or file-like object, string, or pathlib.Path] The predicted values wrapped by the estimator and returned on predict() and used by the score function. The default expectation is that data is a 1D numpy array of y_hat or y_pred values produced by some other estimator. Data can also be a func, which is called and returned, or a file-like object, string, or pathlib.Path at which point the data is loaded from disk using `np.load`.

estimator_type
[str, optional] One of "classifier", "regressor", "clusterer", "DensityEstimator", or "outlier_detector" that allows the contrib estimator to pass the scikit-learn `is_classifier`, etc. functions. If not specified, the Yellowbrick visualizer you're trying to use may error.

fit(*X*, *y=None*)

Fit is a no-op, simply returning self per the scikit-learn API.

predict(*X*)

Predict returns the embedded data but does not perform any checks on the validity of X (e.g. that it has the same shape as the internal data).

score(*X*, *y=None*)

Score uses an appropriate metric for the estimator type and compares the input y values with the pre-predicted values.

statsmodels Visualizers

statsmodels is a Python library that provides utilities for the estimation of several statistical models and includes extensive results and metrics for each estimator. In particular, statsmodels excels at generalized linear models (GLMs) which are far superior to scikit-learn's implementation of ordinary least squares.

This contrib module allows statsmodels users to take advantage of Yellowbrick visualizers by creating a wrapper class that implements the scikit-learn `BaseEstimator`. Using the wrapper class, statsmodels can be passed directly to many visualizers, customized for the scoring and metric functionality required.

Warning: The statsmodel wrapper is currently a prototype and as such is currently a bit trivial. Many options and extra functionality such as weights are not currently handled. We are actively looking for statsmodels users to contribute to this package!

Using the statsmodels wrapper:

```
import statsmodels.api as sm

from functools import partial
from yellowbrick.regressor import ResidualsPlot
from yellowbrick.contrib.statsmodels import StatsModelsWrapper

glm_gaussian_partial = partial(sm.GLM, family=sm.families.Gaussian())
model = StatsModelsWrapper(glm_gaussian_partial)

viz = ResidualsPlot(model)
viz.fit(X_train, y_train)
viz.score(X_test, y_test)
viz.show()
```

You can also use fitted estimators with the wrapper to avoid having to pass a partial function:

```
from yellowbrick.regressor import prediction_error

# Create the OLS model
model = sm.OLS(y, X)

# Get the detailed results
results = model.fit()
print(results.summary())

# Visualize the prediction error
prediction_error(StatsModelWrapper(model), X, y, is_fitted=True)
```

This example also shows the use of a Yellowbrick oneliner, which is often more suited to the analytical style of statsmodels.

API Reference

A basic wrapper for statsmodels that emulates a scikit-learn estimator.

class yellowbrick.contrib.statsmodels.base.**StatsModelsWrapper**(*glm_partial*, *stated_estimator_type='regressor'*, *scorer=<function r2_score>*)

Bases: BaseEstimator

Wrap a statsmodels GLM as a sklearn (fake) BaseEstimator for YellowBrick.

Notes

Note: This wrapper is trivial, options and extra things like weights are not currently handled.

Examples

First import the external libraries and helper utilities:

```
>>> import statsmodels.api as sm
>>> from functools import partial
```

Instantiate a partial with the statsmodels API:

```
>>> glm_gaussian_partial = partial(sm.GLM, family=sm.families.Gaussian())
>>> sm_est = StatsModelsWrapper(glm_gaussian_partial)
```

Create a Yellowbrick visualizer to visualize prediction error:

```
>>> visualizer = PredictionError(sm_est)
>>> visualizer.fit(X_train, y_train)
>>> visualizer.score(X_test, y_test)
```

For statsmodels usage, calling .summary() etc:

```
>>> gaussian_model = glm_gaussian_partial(y_train, X_train)
```

fit(*X*, *y*)

Pretend to be a sklearn estimator, fit is called on creation

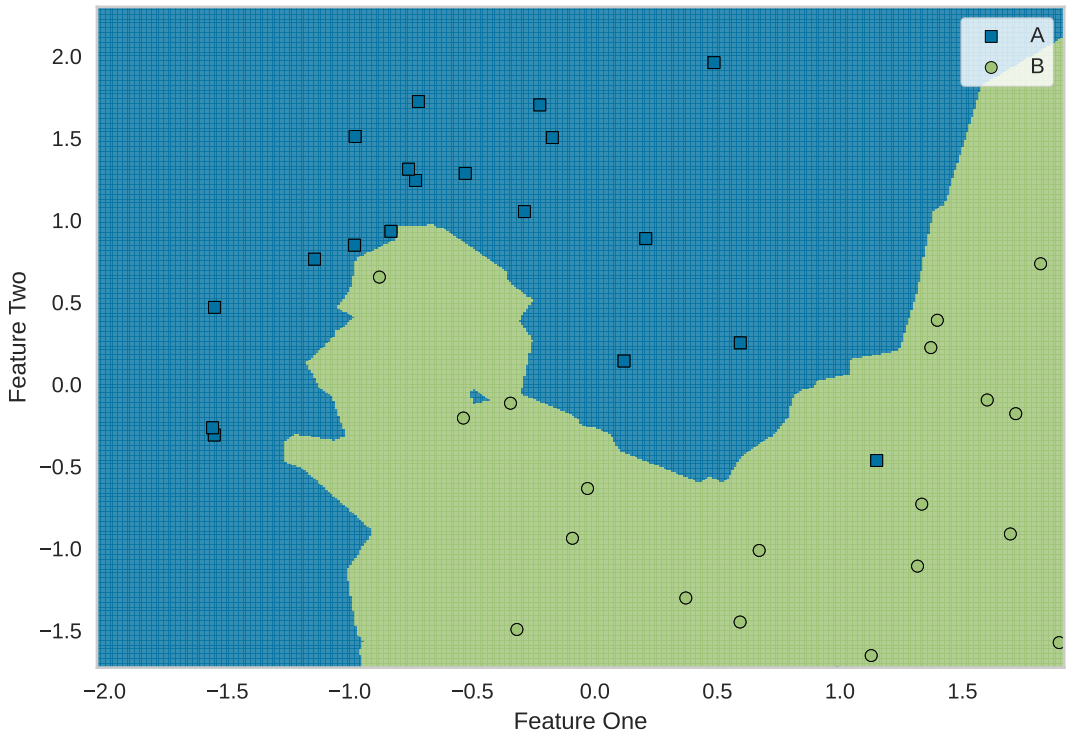
predict(*X*)

score(*X*, *y*)

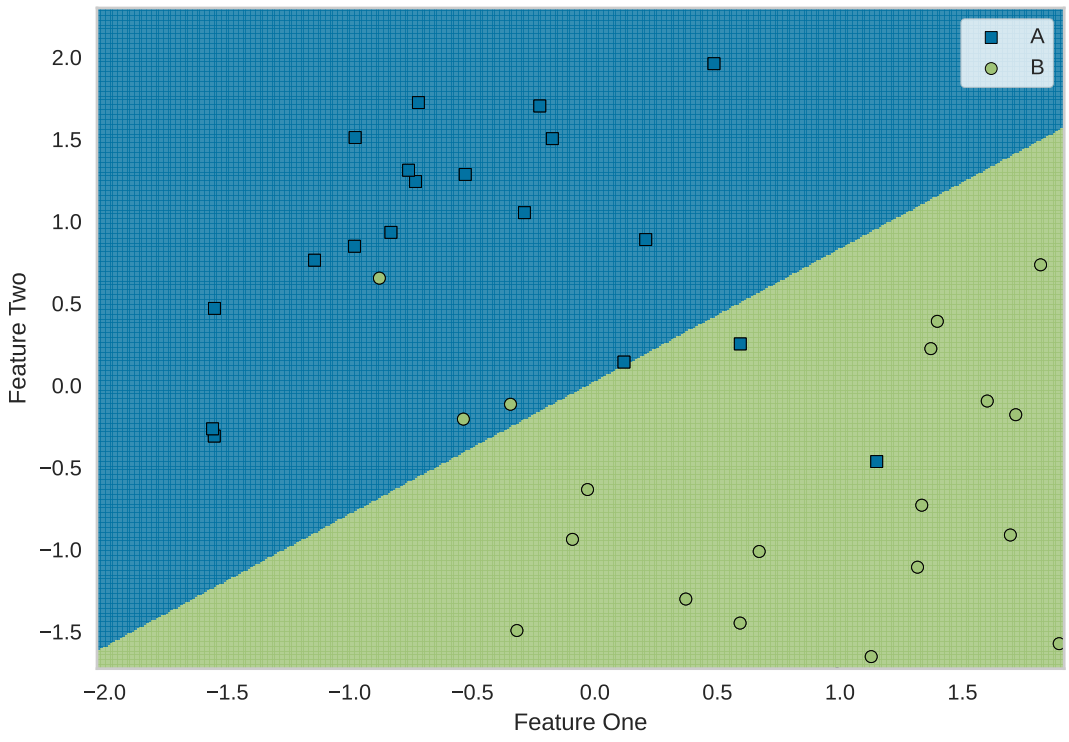
DecisionBoundaries Vizualizer

The DecisionBoundariesVisualizer is a bivariate data visualization algorithm that plots the decision boundaries of each class.

```
from sklearn.model_selection import train_test_split as tts
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons
from sklearn.neighbors import KNeighborsClassifier
from yellowbrick.contrib.classifier import DecisionViz

data_set = make_moons(noise=0.3, random_state=0)

X, y = data_set
X = StandardScaler().fit_transform(X)
X_train, X_test, y_train, y_test = tts(X, y, test_size=.4, random_state=42)

viz = DecisionViz(
    KNeighborsClassifier(3), title="Nearest Neighbors",
    features=['Feature One', 'Feature Two'], classes=['A', 'B']
)
viz.fit(X_train, y_train)
viz.draw(X_test, y_test)
viz.show()
```



```
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split as tts
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons
from sklearn.neighbors import KNeighborsClassifier
from yellowbrick.contrib.classifier import DecisionViz

data_set = make_moons(noise=0.3, random_state=0)

X, y = data_set
X = StandardScaler().fit_transform(X)
X_train, X_test, y_train, y_test = tts(X, y, test_size=.4, random_state=42)


viz = DecisionViz(
    SVC(kernel="linear", C=0.025), title="Linear SVM",
    features=['Feature One', 'Feature Two'], classes=['A', 'B']
)
viz.fit(X_train, y_train)
viz.draw(X_test, y_test)
viz.show()
```



API Reference

class yellowbrick.contrib.classifier.boundaries.**DecisionBoundariesVisualizer**(*estimator*, *ax=None*, *x=None*, *y=None*, *features=None*, *classes=None*, *show_scatter=True*, *step_size=0.0025*, *markers=None*, *pcolormesh_alpha=0.8*, *scatter_alpha=1.0*, *encoder=None*, *is_fitted='auto'*, *force_model=False*, ***kwargs*)

Bases: `ClassificationScoreVisualizer`

DecisionBoundariesVisualizer is a bivariate data visualization algorithm that plots the decision boundaries of each class.

Parameters

estimator
[estimator] A scikit-learn estimator that should be a classifier. If the model is not a classifier, an exception is raised. If the internal model is not fitted, it is fit when the visualizer is fitted, unless otherwise specified by `is_fitted`.

ax
[matplotlib Axes, default: None] The axes to plot the figure on. If not specified the current axes will be used (or generated if required).

x
[string, default: None] The feature name that corresponds to a column name or index postion in the matrix that will be plotted against the x-axis

y
[string, default: None] The feature name that corresponds to a column name or index postion in the matrix that will be plotted against the y-axis

features
[list of strings, default: None] The names of the features or columns

classes
[list of str, defult: None] The class labels to use for the legend ordered by the index of the sorted classes discovered in the `fit()` method. Specifying classes in this manner is used to change the class names to a more specific format or to label encoded integer classes. Some visualizers may also use this field to filter the visualization for specific classes. For more advanced usage specify an encoder rather than class labels.

show_scatter
[boolean, default: True] If boolean is True, then a scatter plot with points will be drawn on top of the decision boundary graph

step_size
[float percentage, default: 0.0025] Determines the step size for creating the numpy meshgrid that will later become the foundation of the decision boundary graph. The default value of 0.0025 means that the step size for constructing the meshgrid will be 0.25%% of differenes of the max and min of x and y for each feature.

markers
[iterable of strings, default: ,od*vh+] Matplotlib style markers for points on the scatter plot points

pcolormesh_alpha
[float, default: 0.8] Sets the alpha transparency for the meshgrid of model boundaries

scatter_alpha
[float, default: 1.0] Sets the alpha transparency for the scatter plot points

encoder
[dict or LabelEncoder, default: None] A mapping of classes to human readable labels. Often there is a mismatch between desired class labels and those contained in the target variable passed to `fit()` or `score()`. The encoder disambiguates this mismatch ensuring that classes are labeled correctly in the visualization.

is_fitted
[bool or str, default="auto"] Specify if the wrapped estimator is already fitted. If False, the estimator will be fit when the visualizer is fit, otherwise, the estimator will not be modified. If "auto" (default), a helper method will check if the estimator is fitted before fitting it again.

force_model
[bool, default: False] Do not check to ensure that the underlying estimator is a classifier. This will prevent an exception when the visualizer is initialized but may result in unexpected or unintended behavior.

kwargs
[dict] Keyword arguments passed to the visualizer base classes.

`draw`(*X*, *y=None*, ***kwargs*)

Called from the fit method, this method creates a decision boundary plot, and if self.scatter is True, it will scatter plot that draws each instance as a class or target colored point, whose location is determined by the feature data set.

`finalize`(***kwargs*)

Sets the title and axis labels and adds a legend.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

`fit`(*X*, *y=None*, ***kwargs*)

The fit method is the primary drawing input for the decision boundaries visualization since it has both the X and y data required for the viz and the transform method does not.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with m features

y
[ndarray or Series of length n] An array or series of target or class values

kwargs
[dict] Pass generic arguments to the drawing method

Returns

self
[instance] Returns the instance of the visualizer

fit_draw(*X*, *y=None*, ***kwargs*)

Fits a transformer to X and y then returns visualization of features or fitted model.

fit_draw_show(*X*, *y=None*, ***kwargs*)

Fits a transformer to X and y then returns visualization of features or fitted model. Then calls show to finalize.

Scatter Plot Visualizer

Sometimes for feature analysis you simply need a scatter plot to determine the distribution of data. Machine learning operates on high dimensional data, so the number of dimensions has to be filtered. As a result these visualizations are typically used as the base for larger visualizers; however you can also use them to quickly plot data during ML analysis.

A scatter visualizer simply plots two features against each other and colors the points according to the target. This can be useful in assessing the relationship of pairs of features to an individual target.

```
from yellowbrick.contrib.scatter import ScatterVisualizer
from yellowbrick.datasets import load_occupancy

# Load the classification dataset
X, y = load_occupancy()

# Specify the target classes
classes = ["unoccupied", "occupied"]

# Instantiate the visualizer
visualizer = ScatterVisualizer(x="light", y="CO2", classes=classes)

visualizer.fit(X, y)           # Fit the data to the visualizer
visualizer.transform(X)        # Transform the data
visualizer.show()              # Finalize and render the figure
```

API Reference

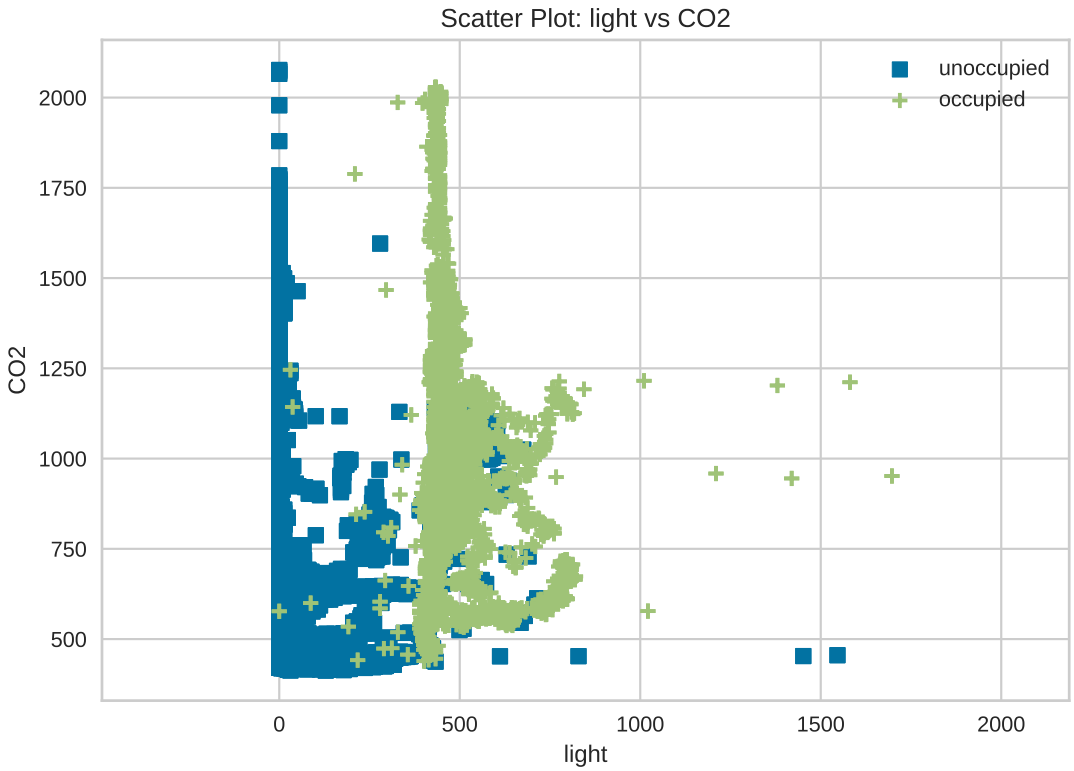
Implements a 2D scatter plot for feature analysis.

class yellowbrick.contrib.scatter.**ScatterVisualizer**(*ax=None*, *x=None*, *y=None*, *features=None*, *classes=None*, *color=None*, *colormap=None*, *markers=None*, *alpha=1.0*, ***kwargs*)

Bases: DataVisualizer

ScatterVisualizer is a bivariate feature data visualization algorithm that plots using the Cartesian coordinates of each point.

Parameters



ax
: [a matplotlib plot, default: None] The axis to plot the figure on.

x
: [string, default: None] The feature name that corresponds to a column name or index postion in the matrix that will be plotted against the x-axis

y
: [string, default: None] The feature name that corresponds to a column name or index postion in the matrix that will be plotted against the y-axis

features
: [a list of two feature names to use, default: None] List of two features that correspond to the columns in the array. The order of the two features correspond to X and Y axes on the graph. More than two feature names or columns will raise an error. If a DataFrame is passed to fit and features is None, feature names are selected that are the columns of the DataFrame.

classes
: [a list of class names for the legend, default: None] If classes is None and a y value is passed to fit then the classes are selected from the target vector.

color
: [optional list or tuple of colors to colorize points, default: None] Use either color to colorize the points on a per class basis or colormap to color them on a continuous scale.

colormap
: [optional string or matplotlib cmap to colorize points, default: None] Use either color to colorize the points on a per class basis or colormap to color them on a continuous scale.

markers
[iterable of strings, default: ,+o*vhd] Matplotlib style markers for points on the scatter plot points

alpha
[float, default: 1.0] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

kwargs
[keyword arguments passed to the super class.]

These parameters can be influenced later on in the visualization process, but can and should be set as early as possible.

`draw`(*X*, *y*, ***kwargs*)

Called from the fit method, this method creates a scatter plot that draws each instance as a class or target colored point, whose location is determined by the feature data set.

`finalize`(***kwargs*)

Adds a title and a legend and ensures that the axis labels are set as the feature names being visualized.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

`fit`(*X*, *y=None*, ***kwargs*)

The fit method is the primary drawing input for the parallel coords visualization since it has both the X and y data required for the viz and the transform method does not.

Parameters

X
[ndarray or DataFrame of shape n x m] A matrix of n instances with 2 features

y
[ndarray or Series of length n] An array or series of target or class values

kwargs
[dict] Pass generic arguments to the drawing method

Returns

self
[instance] Returns the instance of the transformer/visualizer

Missing Values

MissingValues visualizers are a variant of feature visualizers that specifically show places in a dataset that have missing values (numpy NaN).

- *MissingValues Bar*: visualize the count of missing values by feature.
- *MissingValues Dispersion*: visualize the position of missing values by position in the index.

MissingValues Bar

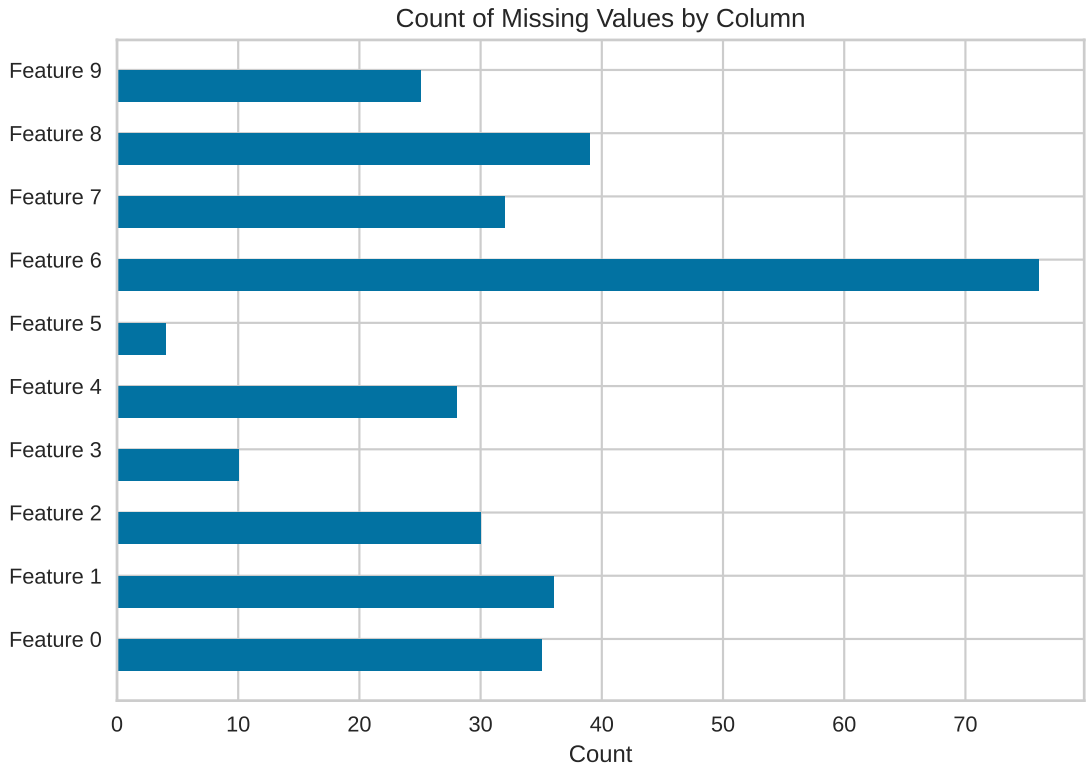
The MissingValues Bar visualizer creates a bar graph that counts the number of missing values per feature column. If the target `y` is supplied to fit, a stacked bar chart is produced.

Without Targets Supplied

```
import numpy as np

from sklearn.datasets import make_classification
from yellowbrick.contrib.missing import MissingValuesBar

# Make a classification dataset
X, y = make_classification(
    n_samples=400, n_features=10, n_informative=2, n_redundant=3,
    n_classes=2, n_clusters_per_class=2, random_state=854
)

# Assign NaN values
X[X > 1.5] = np.nan
features = ["Feature {}".format(str(n)) for n in range(10)]

# Instantiate the visualizer
visualizer = MissingValuesBar(features=features)

visualizer.fit(X)        # Fit the data to the visualizer
visualizer.show()        # Finalize and render the figure
```

With Targets (y) Supplied

```
import numpy as np

from sklearn.datasets import make_classification
from yellowbrick.contrib.missing import MissingValuesBar

# Make a classification dataset
X, y = make_classification(
    n_samples=400, n_features=10, n_informative=2, n_redundant=3,
    n_classes=2, n_clusters_per_class=2, random_state=854
)

# Assign NaN values
X[X > 1.5] = np.nan
features = ["Feature {}".format(str(n)) for n in range(10)]

# Instantiate the visualizer
visualizer = MissingValuesBar(features=features)

visualizer.fit(X, y=y)        # Supply the targets via y
visualizer.show()             # Finalize and render the figure
```



API Reference

Bar visualizer of missing values by column.

class yellowbrick.contrib.missing.bar.**MissingValuesBar**(*width=0.5*, *color=None*, *colors=None*, *classes=None*, ***kwargs*)

Bases: MissingDataVisualizer

The MissingValues Bar visualizer creates a bar graph that lists the total count of missing values for each selected feature column.

When y targets are supplied to fit, the output is a stacked bar chart where each color corresponds to the total NaNs for the feature in that column.

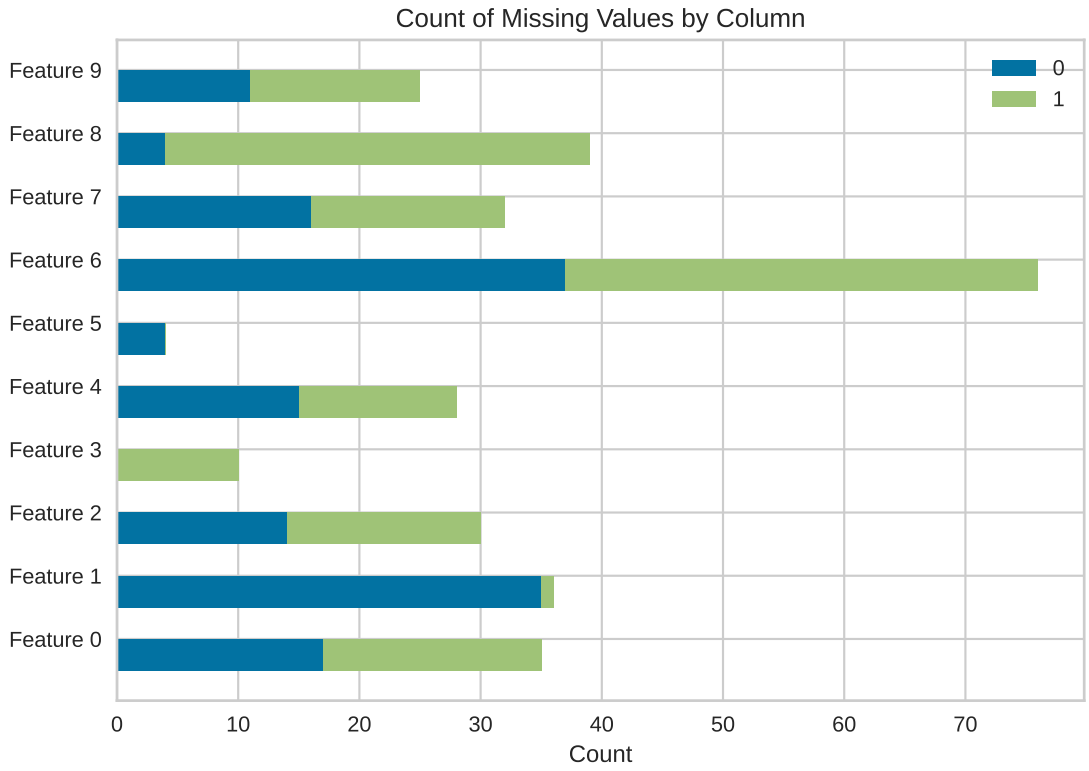
Parameters

alpha
[float, default: 0.5] A value for bending elments with the background.

marker
[matplotlib marker, default: |] The marker used for each element coordinate in the plot

color
[string, default: black] The color for drawing the bar chart when the y targets are not passed to fit.

colors
[list, default: None] The color palette for drawing a stack bar chart when the y targets are passed to fit.



classes
: [list, default: None] A list of class names for the legend. If classes is None and a y value is passed to fit then the classes are selected from the target vector.

kwargs
: [dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> from yellowbrick.contrib.missing import MissingValuesBar
>>> visualizer = MissingValuesBar()
>>> visualizer.fit(X, y=y)
>>> visualizer.show()
```

Attributes

features_
: [np.array] The feature labels ranked according to their importance

classes_
: [np.array] The class labels for each of the target values

draw(*X*, *y*, ***kwargs*)

Called from the fit method, this method generated a horizontal bar plot.

If y is none, then draws a simple horizontal bar chart. If y is not none, then draws a stacked horizontal bar chart for each nan count per target values.

draw_stacked_bar(*nan_col_counts*)

Draws a horizontal stacked bar chart with different colors for each count of nan values per label.

finalize(***kwargs*)

Sets a title and x-axis labels and adds a legend. Also ensures that the y tick values are correctly set to feature names.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

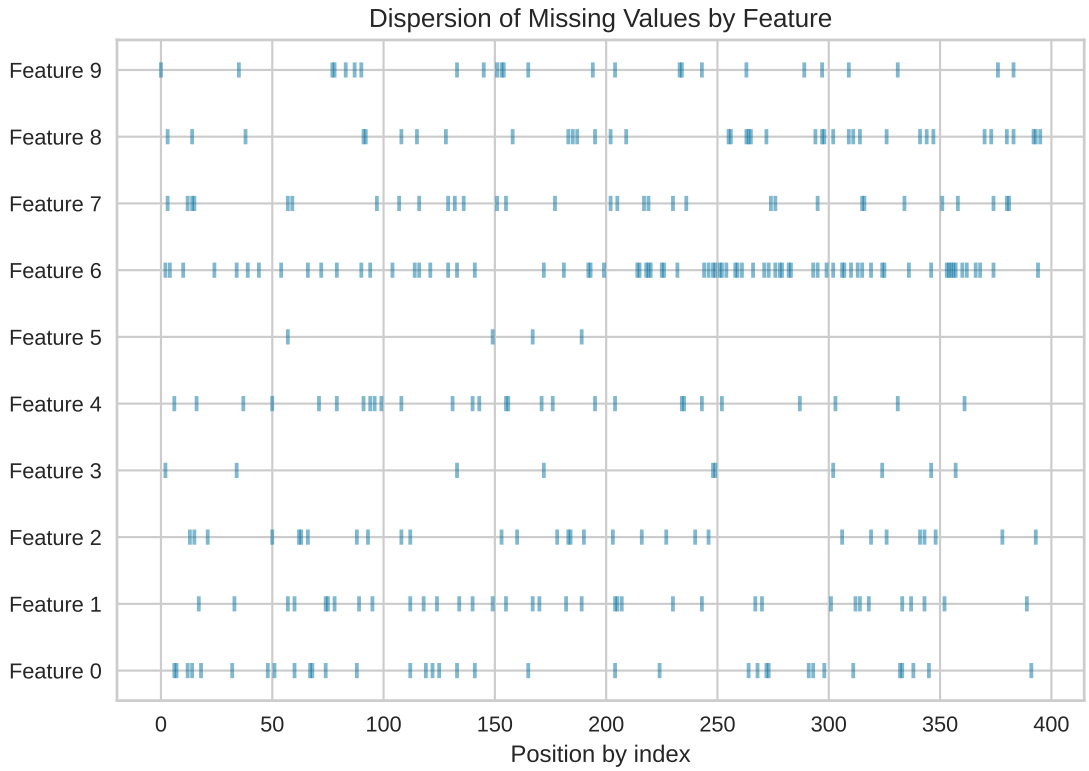
get_nan_col_counts(***kwargs*)

MissingValues Dispersion

The MissingValues Dispersion visualizer creates a chart that maps the position of missing values by the order of the index.

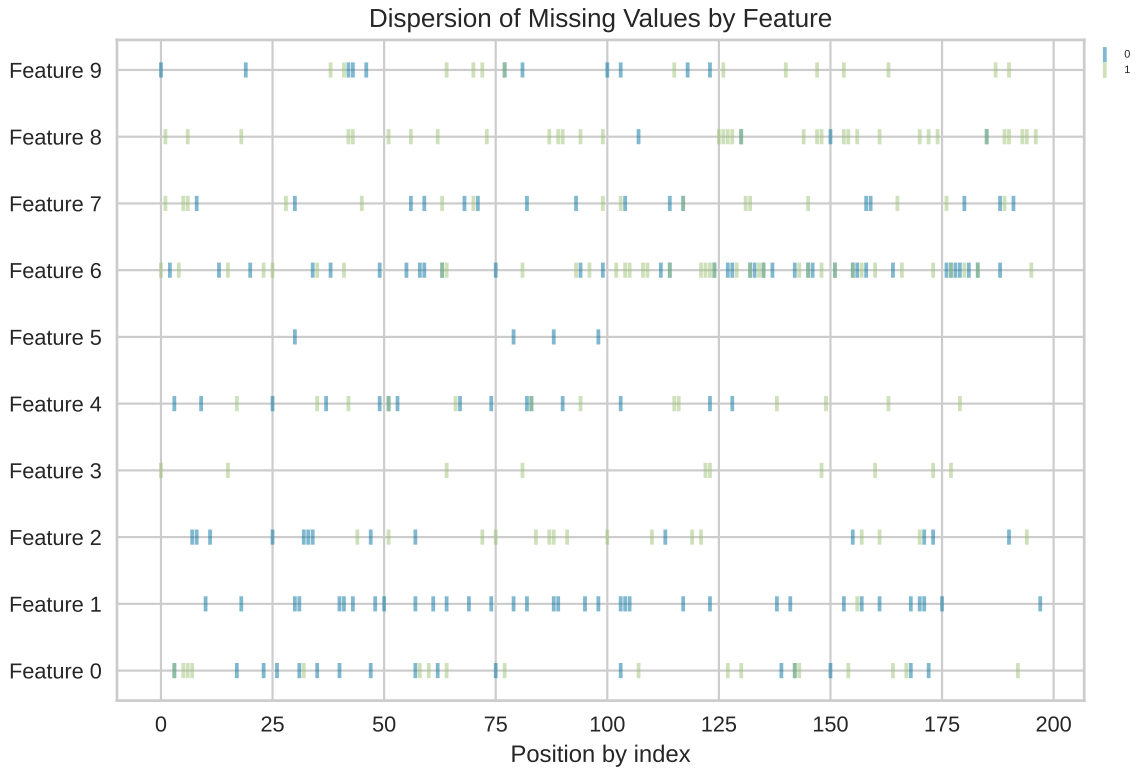
Without Targets Supplied

```
import numpy as np

from sklearn.datasets import make_classification
from yellowbrick.contrib.missing import MissingValuesDispersion

X, y = make_classification(
    n_samples=400, n_features=10, n_informative=2, n_redundant=3,
    n_classes=2, n_clusters_per_class=2, random_state=854
)

# assign some NaN values
X[X > 1.5] = np.nan
features = ["Feature {}".format(str(n)) for n in range(10)]

visualizer = MissingValuesDispersion(features=features)

visualizer.fit(X)
visualizer.show()
```



With Targets (y) Supplied

```
import numpy as np

from sklearn.datasets import make_classification
from yellowbrick.contrib.missing import MissingValuesDispersion

X, y = make_classification(
    n_samples=400, n_features=10, n_informative=2, n_redundant=3,
    n_classes=2, n_clusters_per_class=2, random_state=854
)

# assign some NaN values
X[X > 1.5] = np.nan
features = ["Feature {}".format(str(n)) for n in range(10)]

# Instantiate the visualizer
visualizer = MissingValuesDispersion(features=features)

visualizer.fit(X, y=y) # supply the targets via y
visualizer.show()
```



API Reference

Dispersion visualizer for locations of missing values by column against index position.

class yellowbrick.contrib.missing.dispersion.**MissingValuesDispersion**(*alpha=0.5*, *marker='|'*, *classes=None*, ***kwargs*)

Bases: `MissingDataVisualizer`

The Missing Values Dispersion visualizer shows the locations of missing (nan) values in the feature dataset by the order of the index.

When y targets are supplied to fit, the output dispersion plot is color coded according to the target y that the element refers to.

Parameters

alpha
[float, default: 0.5] A value for bending elments with the background.

marker
[matplotlib marker, default: |] The marker used for each element coordinate in the plot

classes
[list, default: None] A list of class names for the legend. If classes is None and a y value is passed to fit then the classes are selected from the target vector.

kwargs
[dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Examples

```
>>> from yellowbrick.contrib.missing import MissingValuesDispersion
>>> visualizer = MissingValuesDispersion()
>>> visualizer.fit(X, y=y)
>>> visualizer.show()
```

Attributes

features_
[np.array] The feature labels ranked according to their importance

classes_
[np.array] The class labels for each of the target values

draw(*X*, *y*, ***kwargs*)

Called from the fit method, this method creates a scatter plot that draws each instance as a class or target colored point, whose location is determined by the feature data set.

If y is not None, then it draws a scatter plot where each class is in a different color.

draw_multi_dispersion_chart(*nan_locs*)

Draws a multi dimensional dispersion chart, each color corresponds to a different target variable.

finalize(***kwargs*)

Sets the title and x-axis label and adds a legend. Also ensures that the y tick labels are set to the feature names.

Parameters

kwargs: generic keyword arguments.

Notes

Generally this method is called from show and not directly by the user.

get_nan_locs(***kwargs*)

Gets the locations of nans in feature data and returns the coordinates in the matrix

8.3.11 Colors and Style

Yellowbrick believes that visual diagnostics are more effective if visualizations are appealing. As a result, we have borrowed familiar styles from Seaborn and use the new matplotlib 2.0 styles. We hope that these out-of-the-box styles will make your visualizations publication ready, though you can also still customize your own look and feel by directly modifying the visualizations with matplotlib.

For most visualizers, Yellowbrick prioritizes color in its visualizations. There are two types of color sets that can be provided to a visualizer: a palette and a sequence. Palettes are discrete color values usually of a fixed length and are typically used for classification or clustering by showing each class, cluster, or topic. Sequences are continuous color values that do not have a fixed length but rather a range and are typically used for regression or clustering, showing all possible values in the target or distances between items in clusters.

In order to make the distinction easy, most matplotlib colors (both palettes and sequences) can be referred to by name. A complete listing can be imported as follows:

```
import matplotlib.pyplot as plt
from yellowbrick.style.palettes import PALETTES, SEQUENCES, color_palette
```

Palettes and sequences can be passed to visualizers as follows:

```
visualizer = Visualizer(color="bold")
```

Refer to the API listing of each visualizer for specifications on how the color argument is handled. In the next two sections, we will show every possible color palette and sequence currently available in Yellowbrick.

Color Palettes

Color palettes are discrete color lists that have a fixed length. The most common palettes are ordered as "blue", "green", "red", "maroon", "yellow", "cyan", and an optional "key". This allows you to specify these named colors by the first character, e.g. 'bgrmyck' for matplotlib visualizations.

To change the global color palette, use the *set_palette* function as follows:

```
from yellowbrick.style import set_palette
set_palette('flatui')
```

Color palettes are most often used for classifiers to show the relationship between discrete class labels. They can also be used for clustering algorithms to show membership in discrete clusters.

A complete listing of the Yellowbrick color palettes can be visualized as follows:

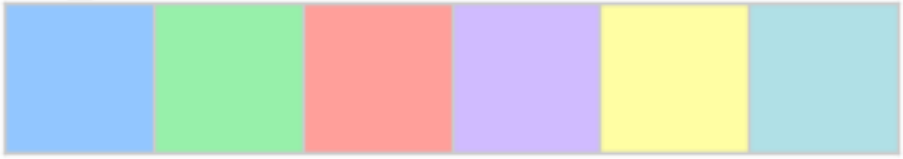
```
# ['blue', 'green', 'red', 'maroon', 'yellow', 'cyan']
for palette in PALETTES.keys():
    color_palette(palette).plot()
    plt.title(palette, loc='left')
```

bold

flatui

muted

sns_pastel



sns_deep

accent

sns_dark

dark

paired

sns_muted

sns_colorblind

set1

yellowbrick

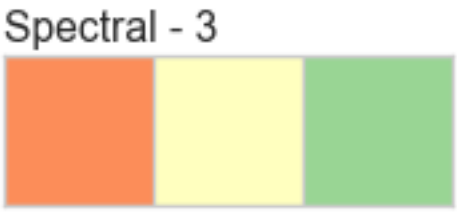
Color Sequences

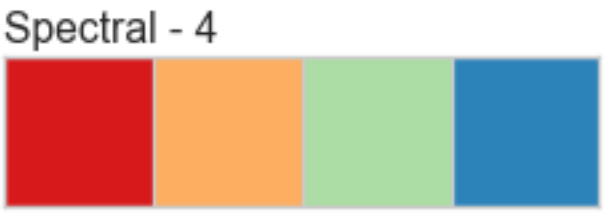
Color sequences are continuous representations of color and are usually defined as a fixed number of steps between a minimum and maximal value. Sequences must be created with a total number of bins (or length) before plotting to ensure that the values are assigned correctly. In the listing below, each sequence is shown with varying lengths to describe the range of colors in detail.

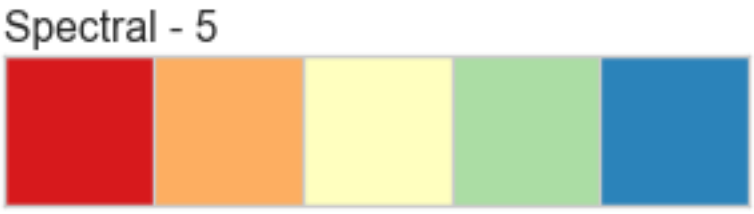
Color sequences are most often used in regressions to show the distribution in the range of target values. They can also be used in clustering and distribution analysis to show distance or histogram data.

Below is a complete listing of all the sequence names available in Yellowbrick:

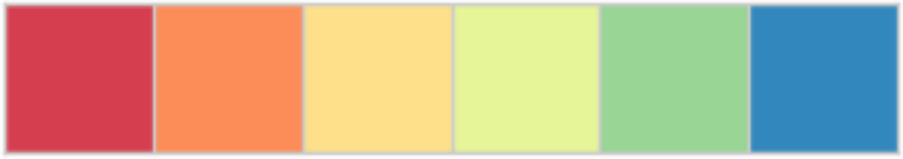
```
for name, maps in SEQUENCES.items():
    for num, palette in maps.items():
        color_palette(palette).plot()
        plt.title("{} - {}".format(name, num), loc='left')
```



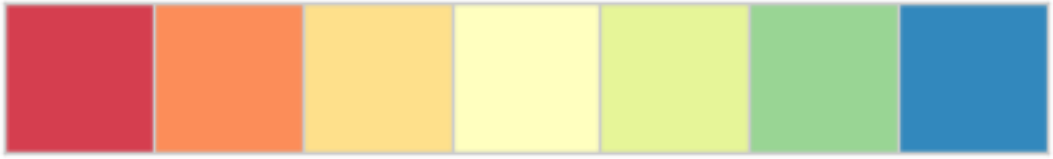




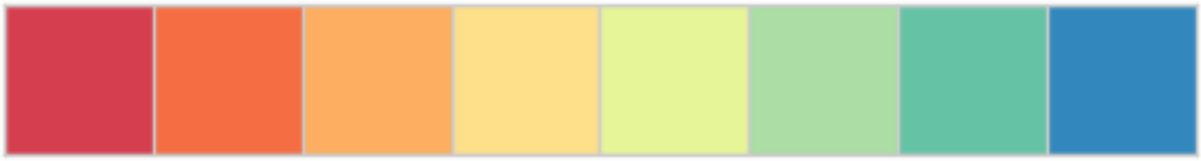
Spectral - 6



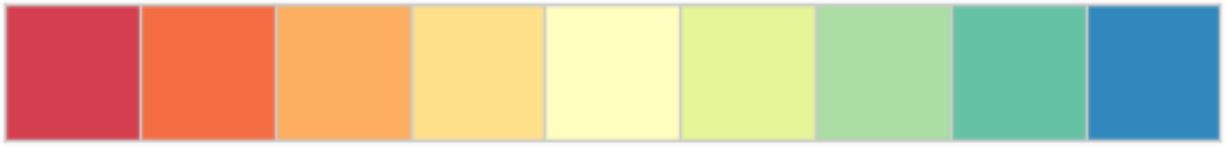
Spectral - 7



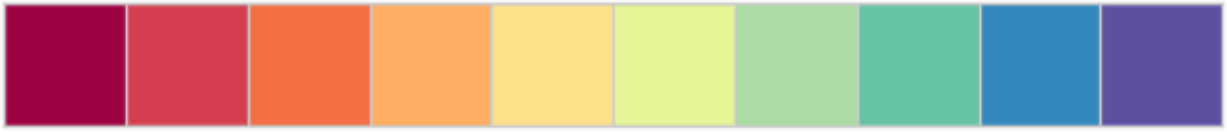
Spectral - 8



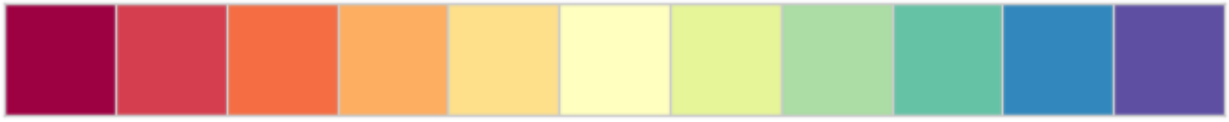
Spectral - 9



Spectral - 10

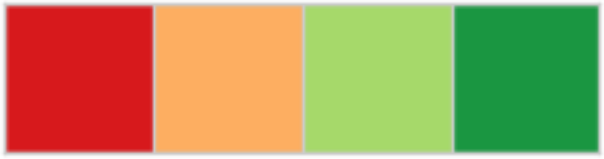


Spectral - 11

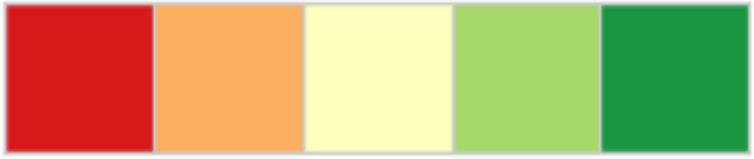


RdYlGn - 3

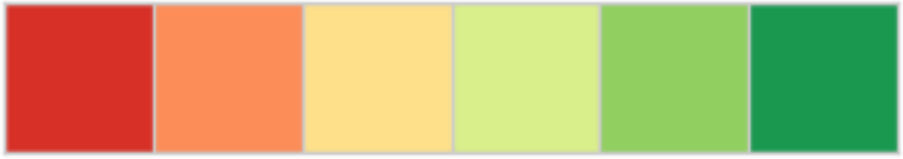
RdYlGn - 4



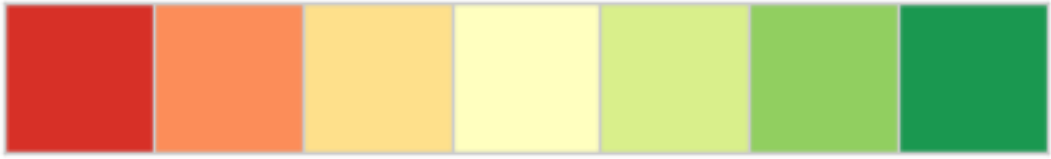
RdYlGn - 5



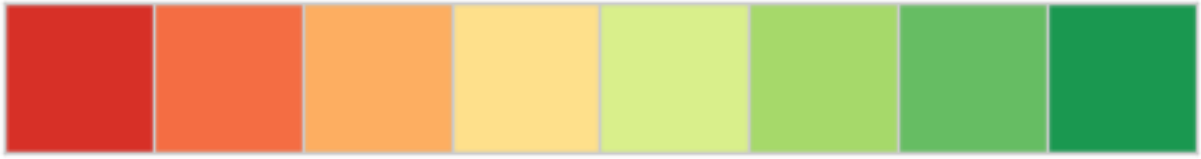
RdYlGn - 6



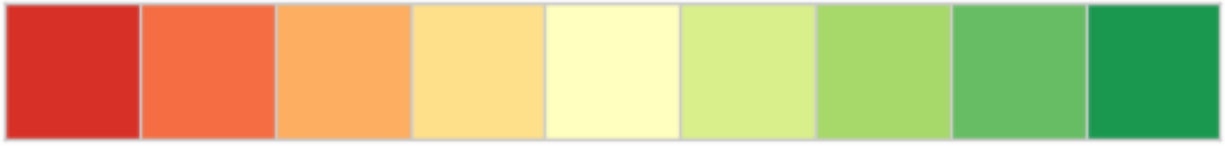
RdYlGn - 7



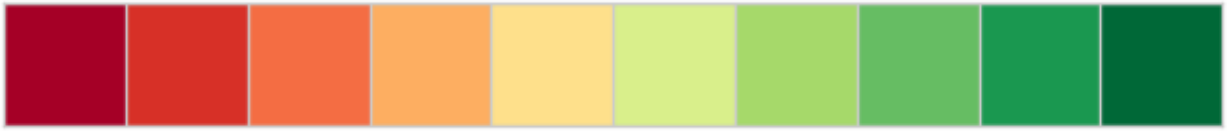
RdYlGn - 8



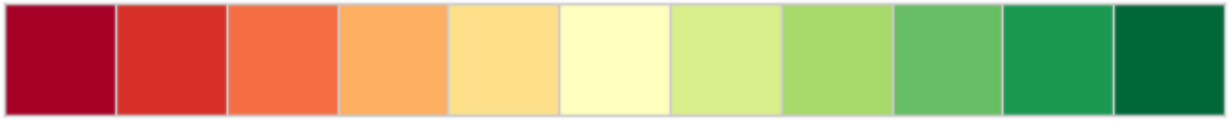
RdYlGn - 9



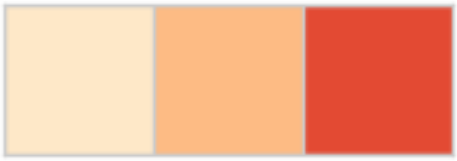
RdYlGn - 10



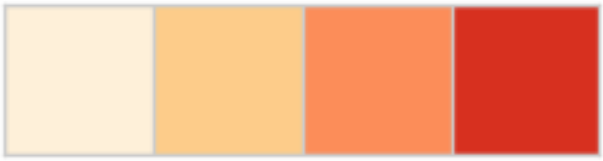
RdYlGn - 11



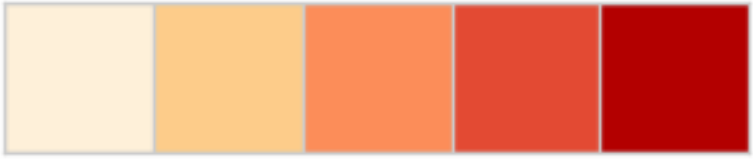
OrRd - 3



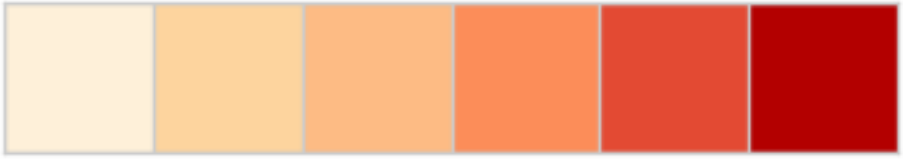
OrRd - 4



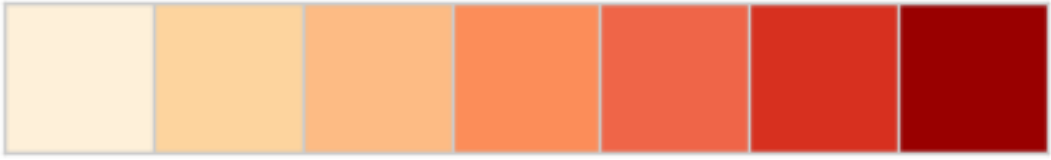
OrRd - 5



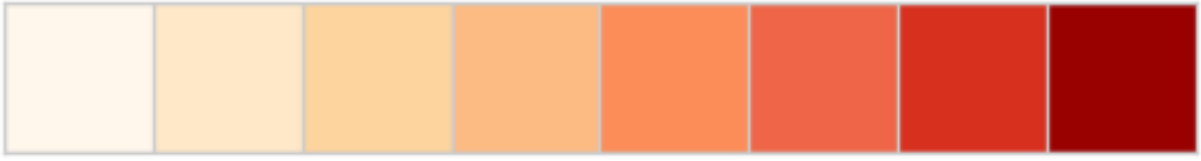
OrRd - 6



OrRd - 7



OrRd - 8



OrRd - 9

PuBu - 3

PuBu - 4

PuBu - 5

PuBu - 6

PuBu - 7

PuBu - 8

PuBu - 9

BuPu - 3

BuPu - 4

BuPu - 5

BuPu - 6

BuPu - 7

BuPu - 8

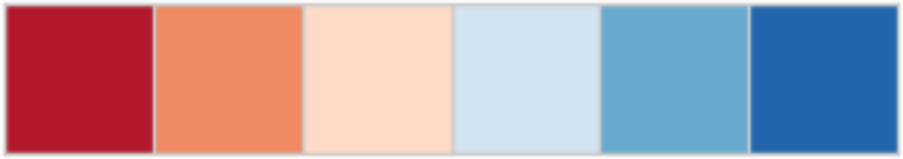
BuPu - 9

RdBu - 3

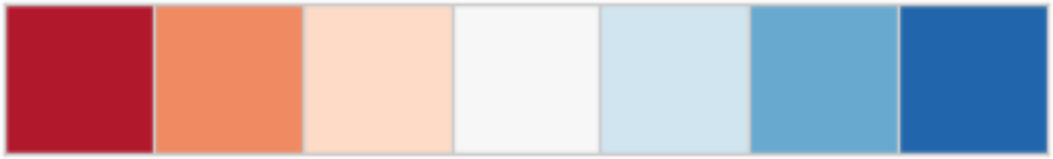
RdBu - 4

RdBu - 5

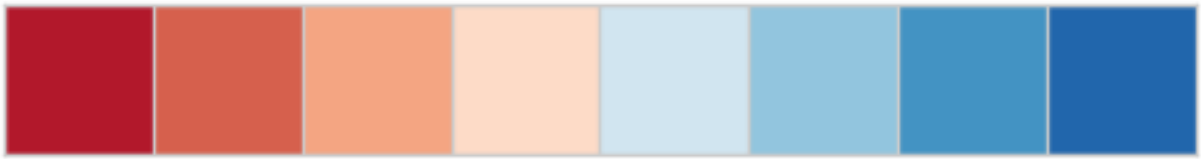
RdBu - 6



RdBu - 7

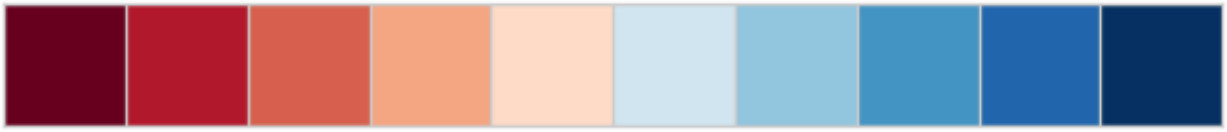


RdBu - 8

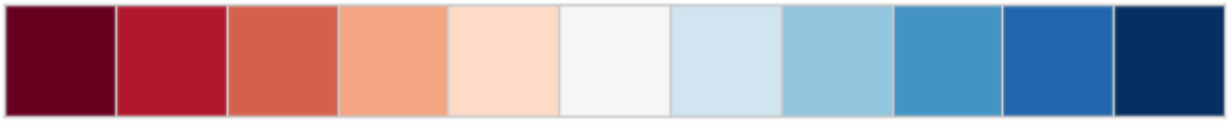


RdBu - 9

RdBu - 10

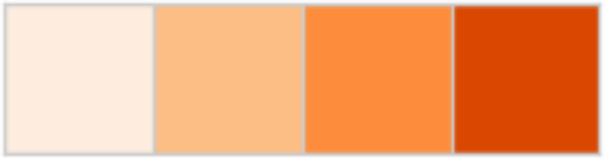


RdBu - 11



Oranges - 3

Oranges - 4



Oranges - 5

Oranges - 6

Oranges - 7

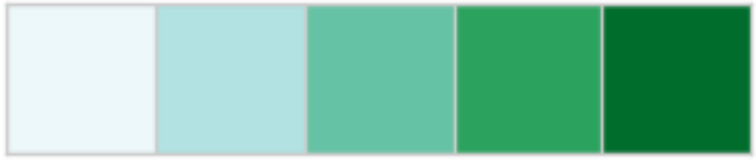
Oranges - 8

Oranges - 9

BuGn - 3

BuGn - 4

BuGn - 5



BuGn - 6

BuGn - 7

BuGn - 8

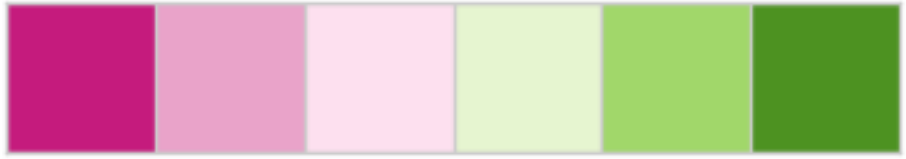
BuGn - 9

PiYG - 3

PiYG - 4

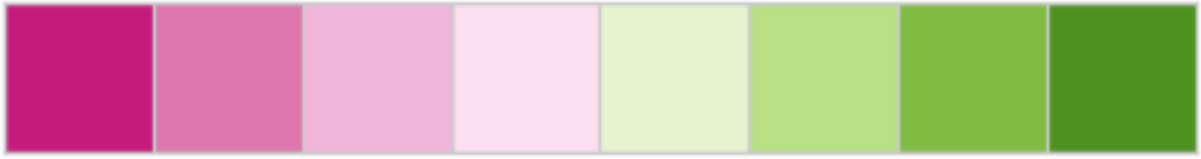
PiYG - 5

PiYG - 6



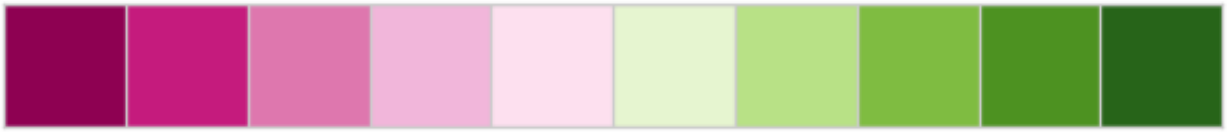
PiYG - 7

PiYG - 8

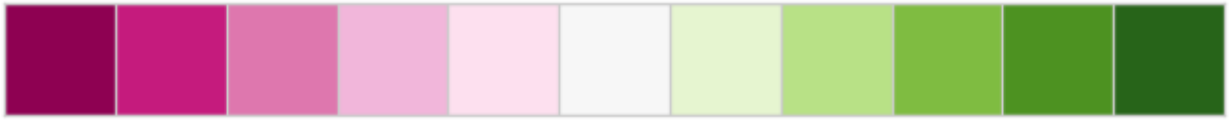


PiYG - 9

PiYG - 10

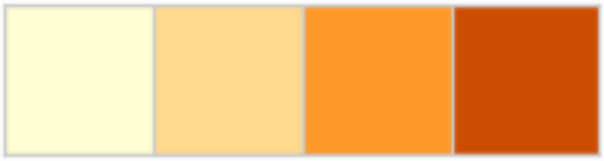


PiYG - 11



YlOrBr - 3

YlOrBr - 4



YlOrBr - 5

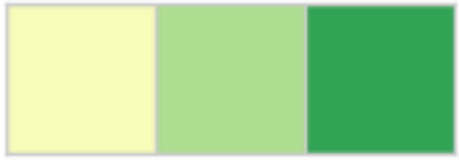
YlOrBr - 6

YlOrBr - 7

YlOrBr - 8

YlOrBr - 9

YlGn - 3



YlGn - 4

YlGn - 5

YlGn - 6

YlGn - 7

YlGn - 8

YlGn - 9

RdPu - 3

RdPu - 4

RdPu - 5

RdPu - 6

RdPu - 7

RdPu - 8

RdPu - 9

Greens - 3

Greens - 4

Greens - 5

Greens - 6

Greens - 7

Greens - 8

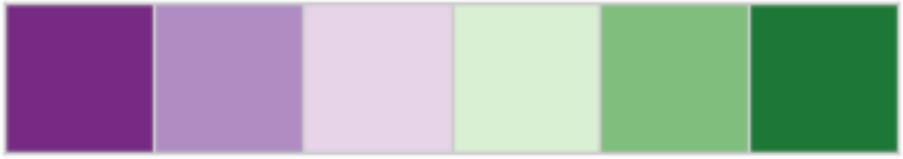
Greens - 9

PRGn - 3

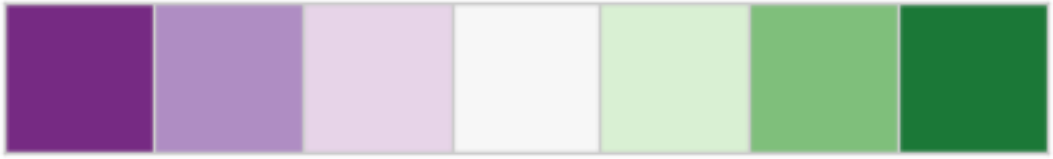
PRGn - 4

PRGn - 5

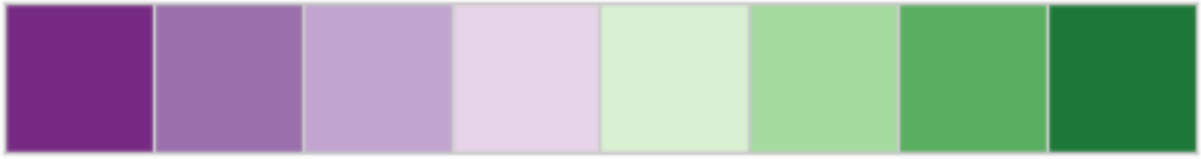
PRGn - 6



PRGn - 7

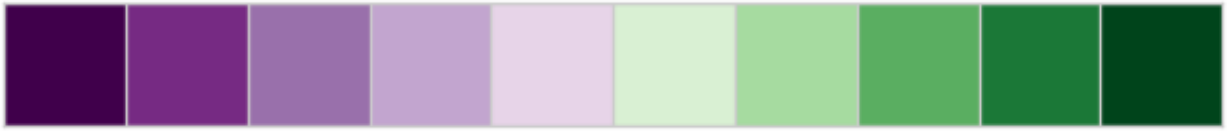


PRGn - 8

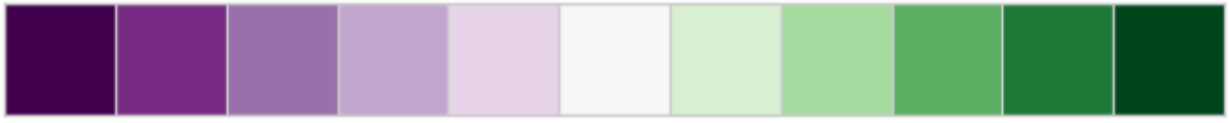


PRGn - 9

PRGn - 10



PRGn - 11



YlGnBu - 3

YlGnBu - 4

YlGnBu - 5

YlGnBu - 6

YlGnBu - 7

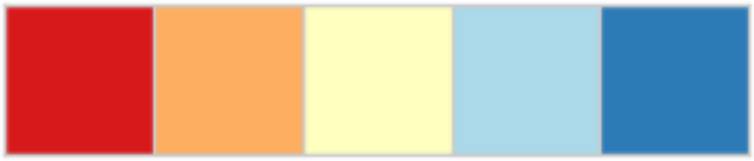
YlGnBu - 8

YlGnBu - 9

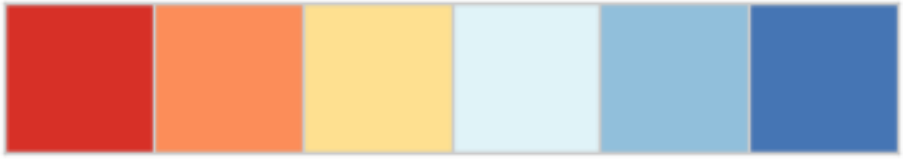
RdYlBu - 3

RdYlBu - 4

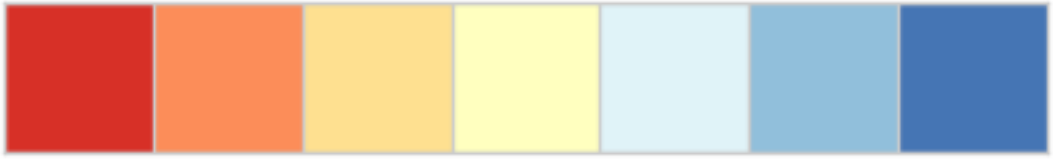
RdYlBu - 5



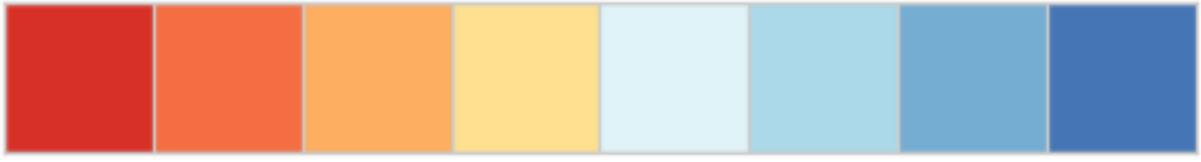
RdYlBu - 6



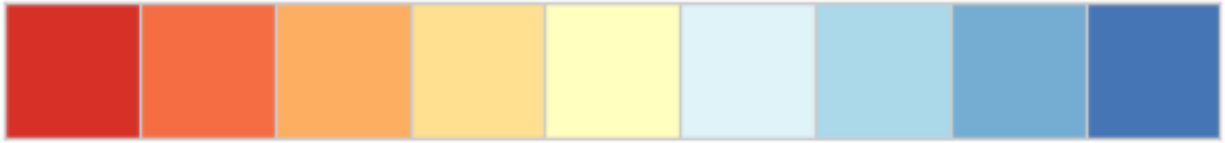
RdYlBu - 7



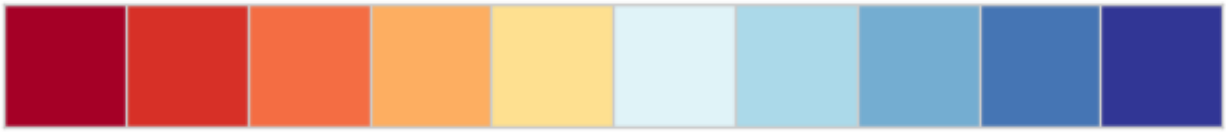
RdYlBu - 8



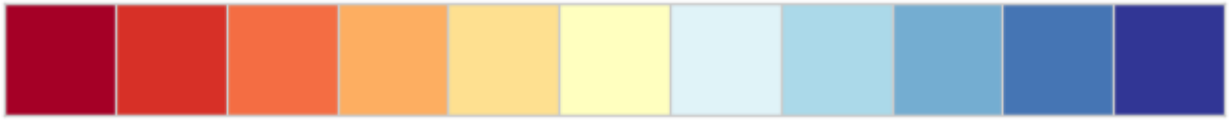
RdYlBu - 9



RdYlBu - 10



RdYlBu - 11

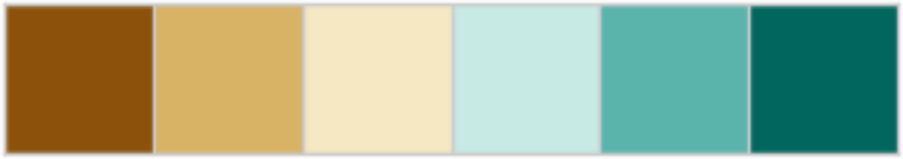


BrBG - 3

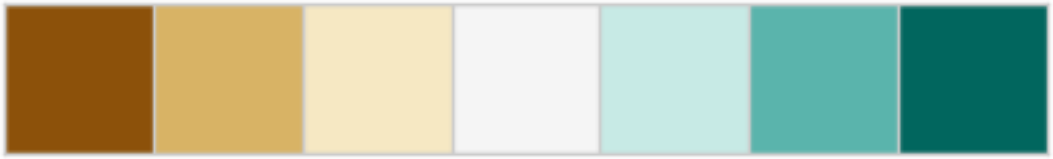
BrBG - 4

BrBG - 5

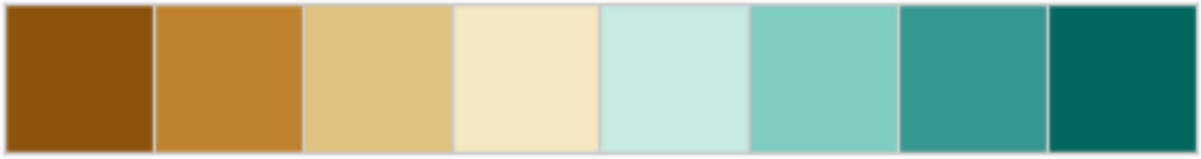
BrBG - 6



BrBG - 7

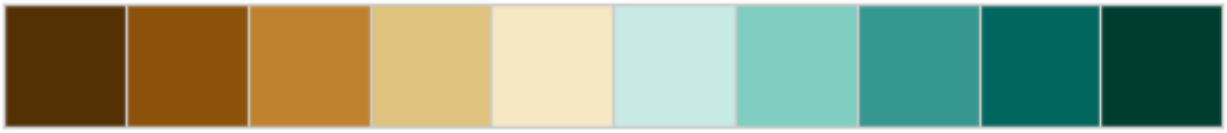


BrBG - 8

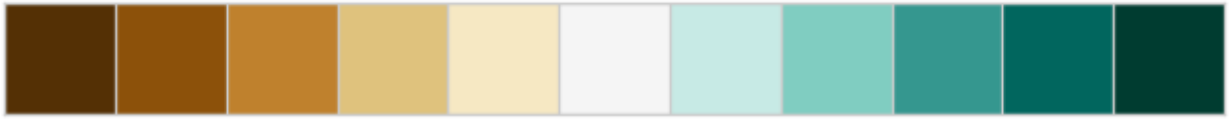


BrBG - 9

BrBG - 10



BrBG - 11



Purples - 3

Purples - 4

Purples - 5

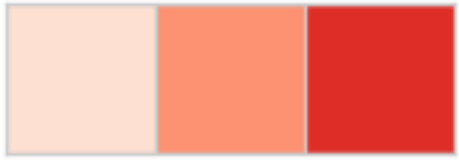
Purples - 6

Purples - 7

Purples - 8

Purples - 9

Reds - 3



Reds - 4

Reds - 5

Reds - 6

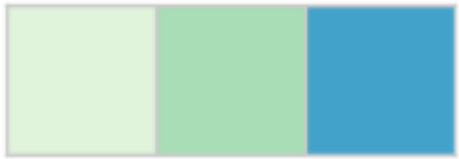
Reds - 7

Reds - 8

Reds - 9

ddl_heat - 12

GnBu - 3

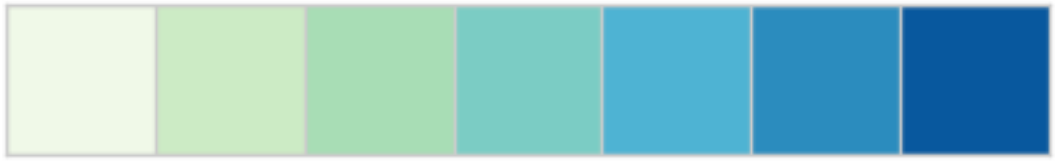


GnBu - 4

GnBu - 5

GnBu - 6

GnBu - 7



GnBu - 8

GnBu - 9

Greys - 3

Greys - 4

Greys - 5

Greys - 6

Greys - 7

Greys - 8

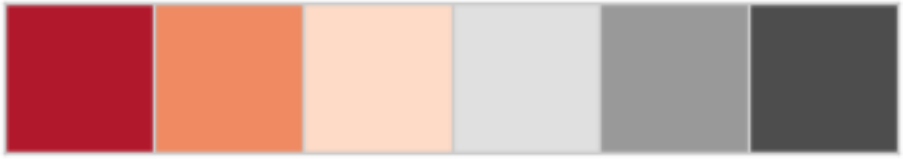
Greys - 9

RdGy - 3

RdGy - 4

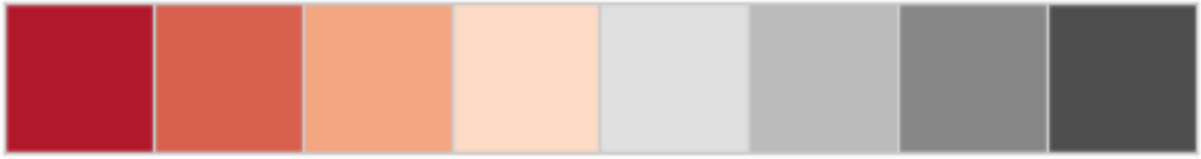
RdGy - 5

RdGy - 6



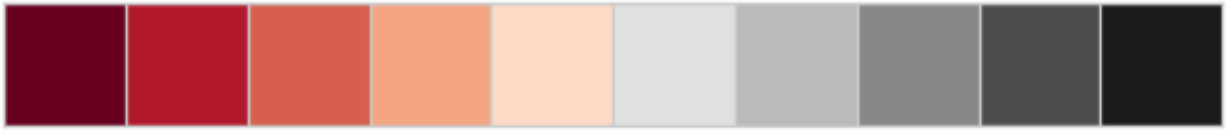
RdGy - 7

RdGy - 8

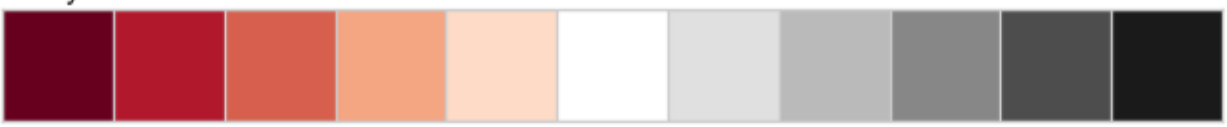


RdGy - 9

RdGy - 10



RdGy - 11



YlOrRd - 3

YlOrRd - 4

YlOrRd - 5

YlOrRd - 6

YlOrRd - 7

YlOrRd - 8

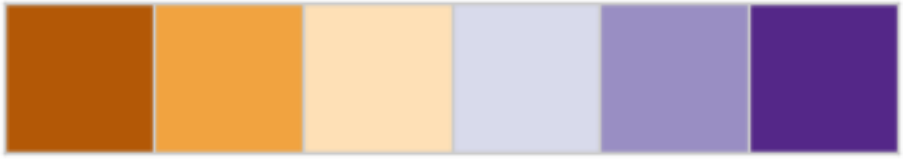
YlOrRd - 9

PuOr - 3

PuOr - 4

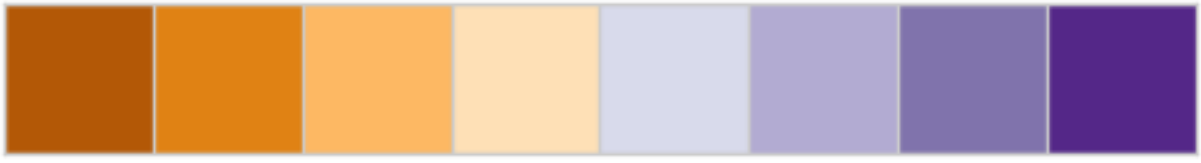
PuOr - 5

PuOr - 6

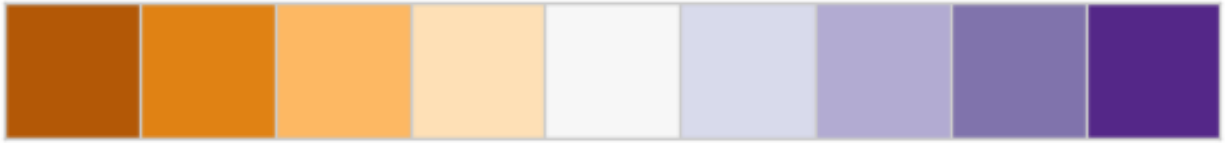


PuOr - 7

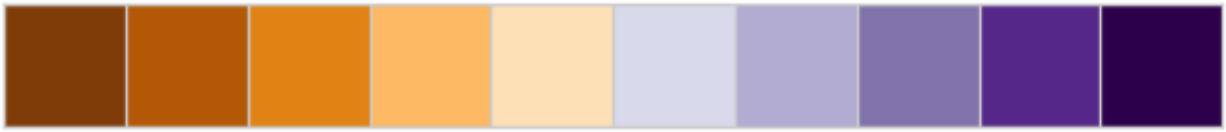
PuOr - 8



PuOr - 9



PuOr - 10



PuOr - 11

PuRd - 3

PuRd - 4

PuRd - 5

PuRd - 6

PuRd - 7

PuRd - 8

PuRd - 9

Blues - 3

Blues - 4

Blues - 5

Blues - 6

Blues - 7

Blues - 8

Blues - 9

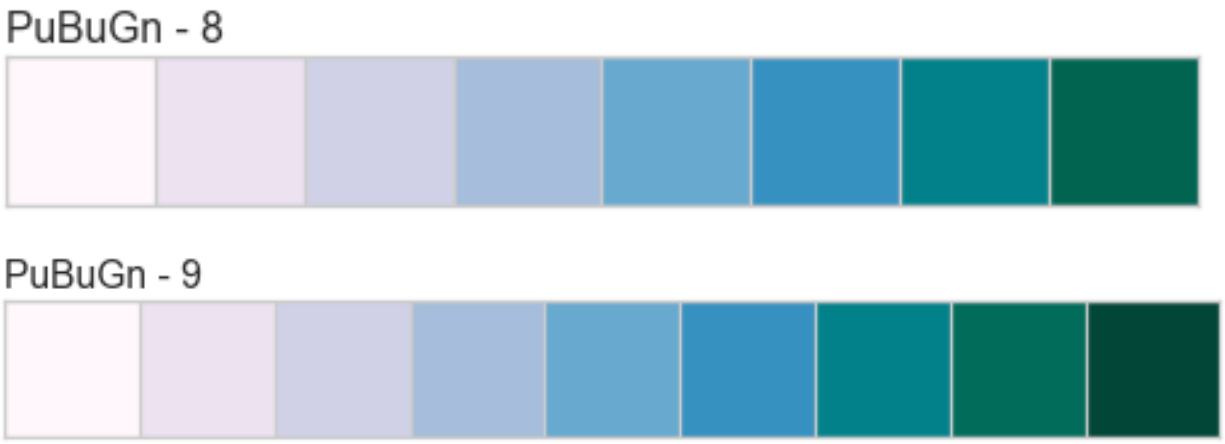
PuBuGn - 3

PuBuGn - 4

PuBuGn - 5

PuBuGn - 6

PuBuGn - 7



API Reference

yellowbrick.style.colors module

Colors and color helpers brought in from an alternate library. See https://bl.ocks.org/mbostock/5577023

class yellowbrick.style.colors.**ColorMap**(*colors='flatui'*, *shuffle=False*)

Bases: object

A helper for mapping categorical values to colors on demand.

property colors

yellowbrick.style.colors.**get_color_cycle**()

Returns the current color cycle from matplotlib.

yellowbrick.style.colors.**resolve_colors**(*n_colors=None*, *colormap=None*, *colors=None*)

Generates a list of colors based on common color arguments, for example the name of a colormap or palette or another iterable of colors. The list is then truncated (or multiplied) to the specific number of requested colors.

Parameters

n_colors
[int, default: None] Specify the length of the list of returned colors, which will either truncate or multiple the colors available. If None the length of the colors will not be modified.

colormap
[str, yellowbrick.style.palettes.ColorPalette, matplotlib.cm, default: None] The name of the matplotlib color map with which to generate colors.

colors
[iterable, default: None] A collection of colors to use specifically with the plot. Overrides colormap if both are specified.

Returns

colors
[list] A list of colors that can be used in matplotlib plots.

Notes

This function was originally based on a similar function in the pandas plotting library that has been removed in the new version of the library.

yellowbrick.style.palettes module

Implements the variety of colors that yellowbrick allows access to by name. This code was originally based on Seaborn's rcmody.py but has since been cleaned up to be Yellowbrick-specific and to dereference tools we don't use. Note that these functions alter the matplotlib rc dictionary on the fly.

`yellowbrick.style.palettes.color_palette`(*palette=None*, *n_colors=None*)

Return a color palette object with color definition and handling.

Calling this function with `palette=None` will return the current matplotlib color cycle.

This function can also be used in a `with` statement to temporarily set the color cycle for a plot or set of plots.

Parameters

palette
[None or str or sequence] Name of a palette or `None` to return the current palette. If a sequence the input colors are used but possibly cycled.

Available palette names from `yellowbrick.colors.palettes` are:

- `accent`
- `dark`
- `paired`
- `pastel`
- `bold`
- `muted`
- `colorblind`
- `sns_colorblind`
- `sns_deep`
- `sns_muted`
- `sns_pastel`
- `sns_bright`
- `sns_dark`
- `flatui`
- `neural_paint`

n_colors
[None or int] Number of colors in the palette. If `None`, the default will depend on how `palette` is specified. Named palettes default to 6 colors which allow the use of the names "bgrmyck", though others do have more or less colors; therefore reducing the size of the list can only be done by specifying this parameter. Asking for more colors than exist in the palette will cause it to cycle.

Returns

list(tuple)
Returns a ColorPalette object, which behaves like a list, but can be used as a context manager and possesses functions to convert colors.

See also:

set_palette()
Set the default color cycle for all plots.

set_color_codes()
Reassign color codes like `"b"`, `"g"`, etc. to colors from one of the yellowbrick palettes.

`colors.resolve_colors()`
Resolve a color map or listed sequence of colors.

yellowbrick.style.palettes.**set_color_codes**(*palette='accent'*)

Change how matplotlib color shorthands are interpreted.

Calling this will change how shorthand codes like "b" or "g" are interpreted by matplotlib in subsequent plots.

Parameters

palette
[str] Named yellowbrick palette to use as the source of colors.

See also:

`set_palette`
Color codes can also be set through the function that sets the matplotlib color cycle.

yellowbrick.style.rcmod module

Modifies the matplotlib rcParams in order to make yellowbrick more appealing. This has been modified from Seaborn's rcmod.py: github.com/mwaskom/seaborn in order to alter the matplotlib rc dictionary on the fly.

NOTE: matplotlib 2.0 styles mean we can simply convert this to a stylesheet!

yellowbrick.style.rcmod.**reset_defaults**()

Restore all RC params to default settings.

yellowbrick.style.rcmod.**reset_orig**()

Restore all RC params to original settings (respects custom rc).

yellowbrick.style.rcmod.**set_aesthetic**(*palette='yellowbrick'*, *font='sans-serif'*, *font_scale=1*, *color_codes=True*, *rc=None*)

Set aesthetic parameters in one step.

Each set of parameters can be set directly or temporarily, see the referenced functions below for more information.

Parameters

palette
[string or sequence] Color palette, see `color_palette()`

font
[string] Font family, see matplotlib font manager.

font_scale
[float, optional] Separate scaling factor to independently scale the size of the font elements.

color_codes
[bool] If `True` and `palette` is a yellowbrick palette, remap the shorthand color codes (e.g. "b", "g", "r", etc.) to the colors from this palette.

rc
[dict or None] Dictionary of rc parameter mappings to override the above.

yellowbrick.style.rcmod.**set_palette**(*palette*, *n_colors=None*, *color_codes=False*)

Set the matplotlib color cycle using a seaborn palette.

Parameters

palette
[yellowbrick color palette | seaborn color palette (with `sns_` prepended)] Palette definition. Should be something that `color_palette()` can process.

n_colors
[int] Number of colors in the cycle. The default number of colors will depend on the format of `palette`, see the `color_palette()` documentation for more information.

color_codes
[bool] If `True` and `palette` is a seaborn palette, remap the shorthand color codes (e.g. "b", "g", "r", etc.) to the colors from this palette.

`yellowbrick.style.rcmod.`**`set_style`**(*style=None*, *rc=None*)

Set the aesthetic style of the plots.

This affects things like the color of the axes, whether a grid is enabled by default, and other aesthetic elements.

Parameters

style
[dict, None, or one of {darkgrid, whitegrid, dark, white, ticks}] A dictionary of parameters or the name of a preconfigured set.

rc
[dict, optional] Parameter mappings to override the values in the preset seaborn style dictionaries. This only updates parameters that are considered part of the style definition.

8.3.12 Figures and Axes

This document is an open letter to the PyData community, particularly those that are involved in matplotlib development. We'd like to get some advice on the API choice we've made and thoughts about our use of the matplotlib Axes objects.

One of the most complex parts of designing a visualization library around matplotlib is working with figures and axes. As defined in The Lifecycle of a Plot, these central objects of matplotlib plots are as follows:

- A Figure is the final image that may contain 1 or more Axes.
- An Axes represents an individual plot

Based on these definitions and and the advice to "try to use the object-oriented interface over the pyplot interface", the Yellowbrick interface is designed to wrap a matplotlib `axes.Axes`. We propose the following general use case for most visualizers:

```
import matplotlib.pyplot as plt
from yellowbrick import Visualizer, quick_visualizer

fig, ax = plt.subplots()

# Object oriented approach
viz = Visualizer(ax=ax)
viz.fit(X, y)
viz.show()

# Quick method approach
viz = quick_visualizer(X, y, ax=ax)
viz.show()
```

This design allows users to more directly control the size, style, and interaction with the plot (though YB does provide some helpers for these as well). For example, if a user wanted to generate a report with multiple visualizers for a classification problem, it may looks something like:

```
import matplotlib.pyplot as plt

from yellowbrick.features import FeatureImportances
from yellowbrick.classifier import ConfusionMatrix, ClassificationReport, ROCAUC
from sklearn.linear_model import LogisticRegression

fig, axes = plot.subplots(2, 2)

model = LogisticRegression()
visualgrid = [
    FeatureImportances(ax=axes[0][0]),
    ConfusionMatrix(model, ax=axes[0][1]),
    ClassificationReport(model, ax=axes[1][0]),
    ROCAUC(model, ax=axes[1][1]),
]

for viz in visualgrid:
    viz.fit(X_train, y_train)
    viz.score(X_test, y_test)
    viz.finalize()

plt.show()
```

This is a common use case and we're working on the idea of "visual pipelines" to support this type of development because, for machine learning, users generally want a suite of visualizers or a report, not just a single visualization. The API requirement to support this has therefore been that visualizers use the `ax` object passed to them and not `plt`. If the user does not pass a specific `ax` then the global current axes is used via `plt.gca`. Generally, visualizers should behave as though they are a plot that as part of a larger figure.

Visualizers are getting more complex, however, and some are becoming multi-axes plots in their own right. For example:

- The ResidualsPlot has a scatter plot axes and a histogram axes
- The JointPlot has a scatter plot and two histogram axes
- Data driven scatter plot axes often have colorbar axes
- The PCA plot has scatter plot, color bar, and heatmap axes
- The confusion matrix probability histogram is a grid of axes for each class pair
- The ICDM has an inset axes that acts as a dynamic legend

Although it would have been easier to simply embed the figure into the visualizer and use a `GridSpec` or other layout tool, the focus on ensuring visualizers are individual plots that wrap an Axes has made us bend over backward to adjust the plot inside of the axes area that was originally supplied, primarily by using `make_axes_locateable`, which is part of the AxesGrid toolkit.

Generally, it appears that the AxesGrid Toolkit is the right tool for Yellowbrick - many of the examples shown are similar to the things that Yellowbrick is trying to do. However, this package is not fully documented with examples and some helper utilities that would be useful, for example the `ImageGrid`, still require a `figure.Figure` object.

At this point we are left with some important questions about Yellowbrick's development roadmap:

1. Like Seaborn, should YB have two classes of visualizer, one that wraps an axes and one that wraps a figure?
2. Should we go all in on the AxesGrid toolkit and continue to restrict our use of the figure, will this method be supported in the long run?

Other notes and discussion:

- Create equal aspect (square) plot with multiple axes when data limits are different?

Note: Many examples utilize data from the UCI Machine Learning repository. In order to run the accompanying code, make sure to follow the instructions in *Example Datasets* to download and load the required data.

A guide to finding the visualizer you're looking for: generally speaking, visualizers can be data visualizers which visualize instances relative to the model space; score visualizers which visualize model performance; model selection visualizers which compare multiple model forms against each other; and application specific-visualizers. This can be a bit confusing, so we've grouped visualizers according to the type of analysis they are well suited for.

Feature analysis visualizers are where you'll find the primary implementation of data visualizers. Regression, classification, and clustering analysis visualizers can be found in their respective libraries. Finally, visualizers for text analysis are also available in Yellowbrick! Other utilities, such as styles, best fit lines, and Anscombe's visualization, can also be found in the links above.

8.4 Oneliners

Yellowbrick's quick methods are visualizers in a single line of code!

Yellowbrick is designed to give you as much control as you would like over the plots you create, offering parameters to help you customize everything from color, size, and title to preferred evaluation or correlation measure, optional bestfit lines or histograms, and cross validation techniques. To learn more about how to customize your visualizations using those parameters, check out the *Visualizers and API*.

But... sometimes you just want to build a plot with a single line of code!

On this page we'll explore the Yellowbrick quick methods (aka "oneliners"), which return a fully fitted, finalized visualizer object in only a single line.

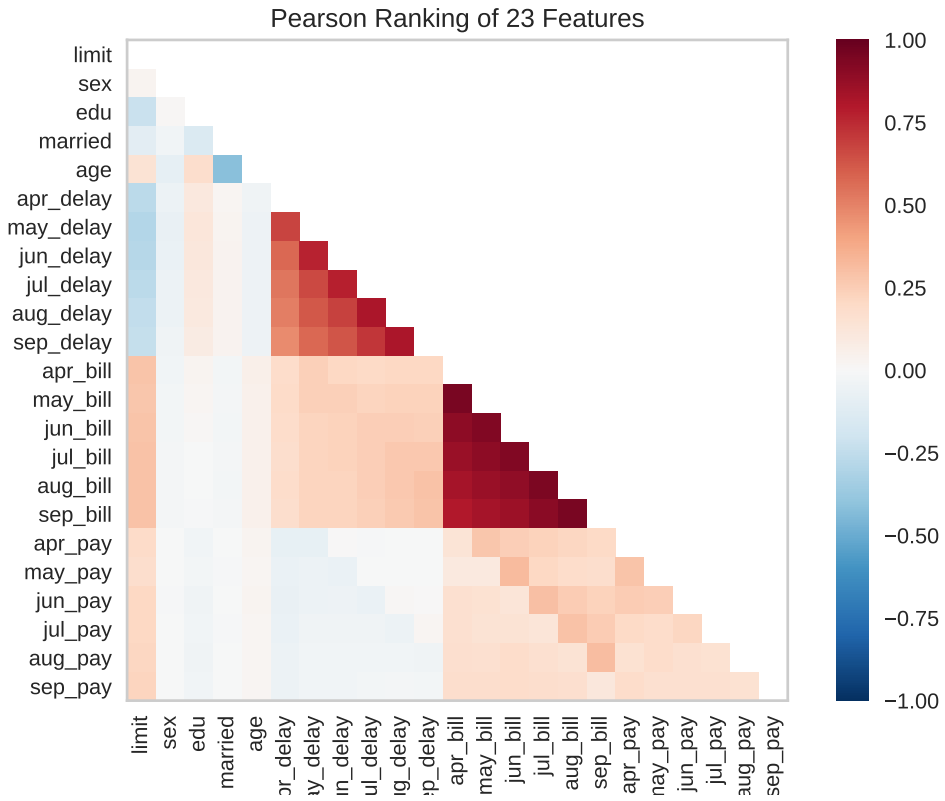
Note: This page illustrates oneliners for some of our most popular visualizers for feature analysis, classification, regression, clustering, and target evaluation, but is not a comprehensive list. Nearly every Yellowbrick visualizer has an associated quick method!

8.4.1 Feature Analysis

Rank2D

The `rank1d` and `rank2d` plots show pairwise rankings of features to help you detect relationships. More about *Rank Features*.

```
from yellowbrick.features import rank2d
from yellowbrick.datasets import load_credit


X, _ = load_credit()
visualizer = rank2d(X)
```

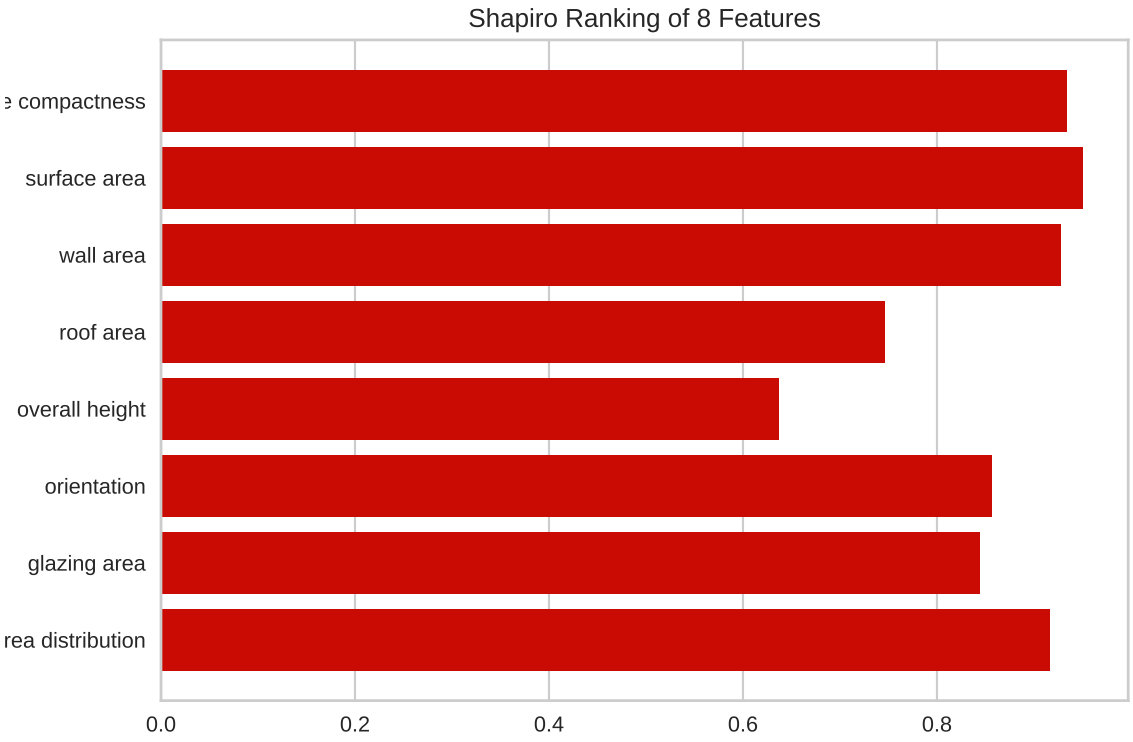



```
from yellowbrick.features import rank1d
from yellowbrick.datasets import load_energy


X, _ = load_energy()
visualizer = rank1d(X, color="r")
```

Parallel Coordinates

The `parallel_coordinates` plot is a horizontal visualization of instances, disaggregated by the features that describe them. More about *Parallel Coordinates*.

```
from sklearn.datasets import load_wine
from yellowbrick.features import parallel_coordinates


X, y = load_wine(return_X_y=True)
visualizer = parallel_coordinates(X, y, normalize="standard")
```



Radial Visualization

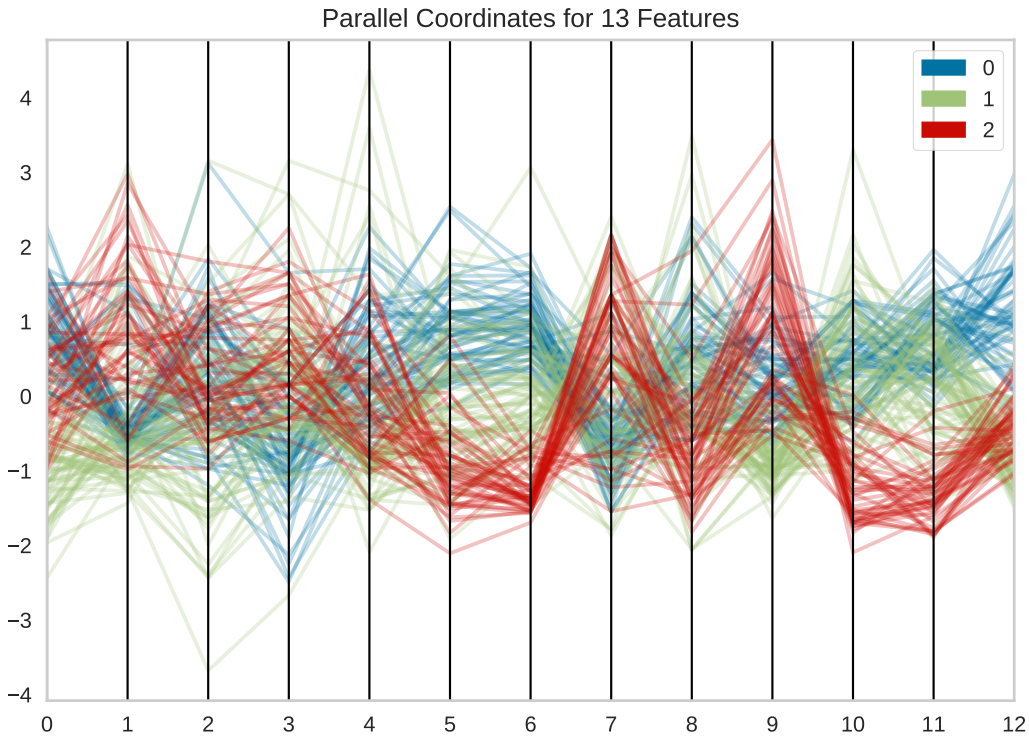
The `radviz` plot shows the separation of instances around a unit circle. More about *RadViz Visualizer*.

```
from yellowbrick.features import radviz
from yellowbrick.datasets import load_occupancy


X, y = load_occupancy()
visualizer = radviz(X, y, colors=["maroon", "gold"])
```

PCA

A `pca_decomposition` is a projection of instances based on principal components. More about *PCA Projection*.

```
from yellowbrick.datasets import load_spam
from yellowbrick.features import pca_decomposition


X, y = load_spam()
visualizer = pca_decomposition(X, y)
```



Manifold

The `manifold_embedding` plot is a high dimensional visualization with manifold learning, which can show nonlinear relationships in the features. More about *Manifold Visualization*.

```
from sklearn.datasets import load_iris
from yellowbrick.features import manifold_embedding


X, y = load_iris(return_X_y=True)
visualizer = manifold_embedding(X, y)
```

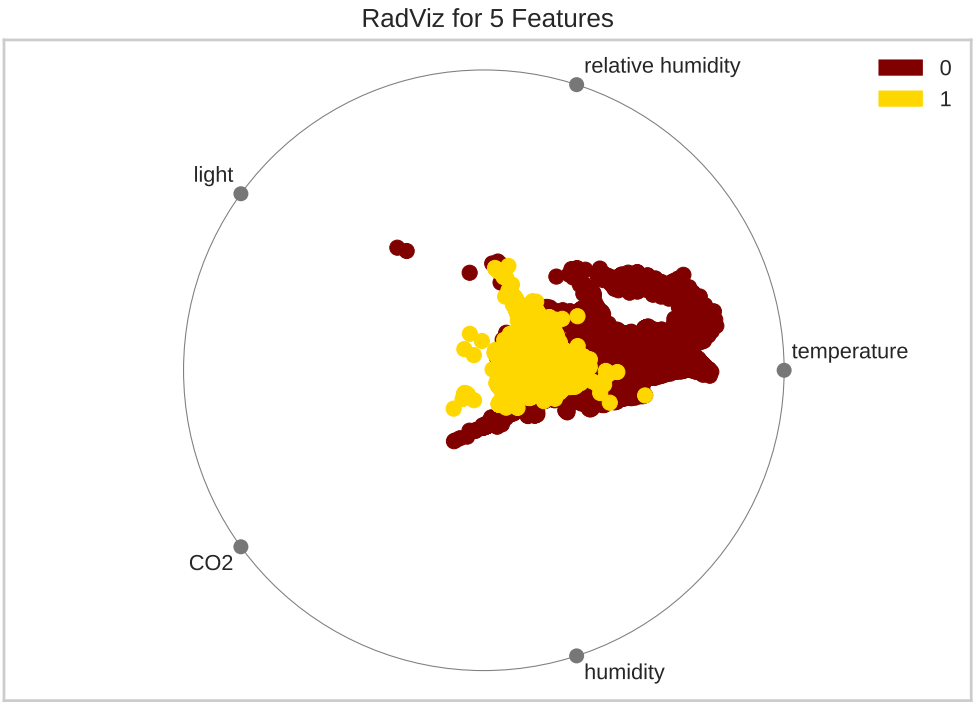
8.4.2 Classification

Class Prediction Error

A `class_prediction_error` plot illustrates the error and support in a classification as a bar chart. More about *Class Prediction Error*.

```
from yellowbrick.datasets import load_game
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import RandomForestClassifier
from yellowbrick.classifier import class_prediction_error
```

(continues on next page)



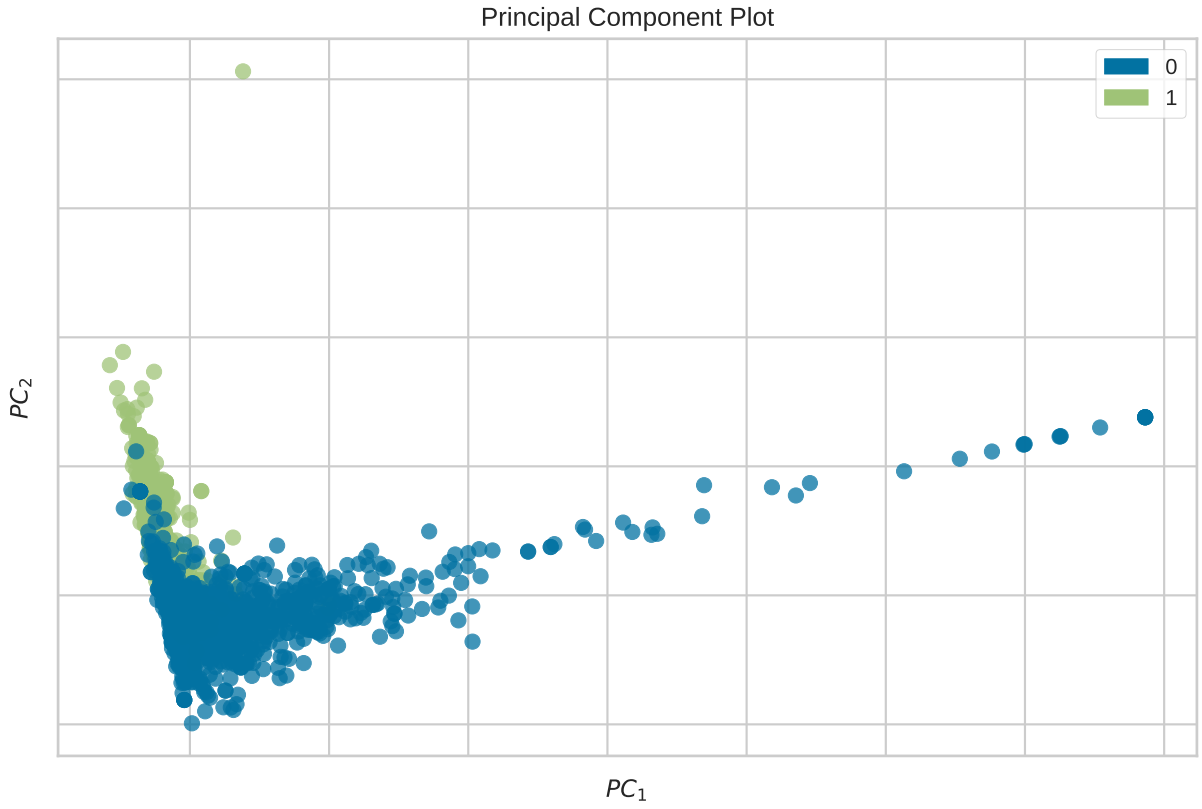
(continued from previous page)

```
X, y = load_game()
X = OneHotEncoder().fit_transform(X)
visualizer = class_prediction_error(
    RandomForestClassifier(n_estimators=10), X, y
)
```

Classification Report

A `classification_report` is a visual representation of precision, recall, and F1 score. More about *Classification Report*.

```
from yellowbrick.datasets import load_credit
from sklearn.ensemble import RandomForestClassifier
from yellowbrick.classifier import classification_report


X, y = load_credit()
visualizer = classification_report(
    RandomForestClassifier(n_estimators=10), X, y
)
```



Confusion Matrix

A `confusion_matrix` is a visual description of per-class decision making. More about *Confusion Matrix*.

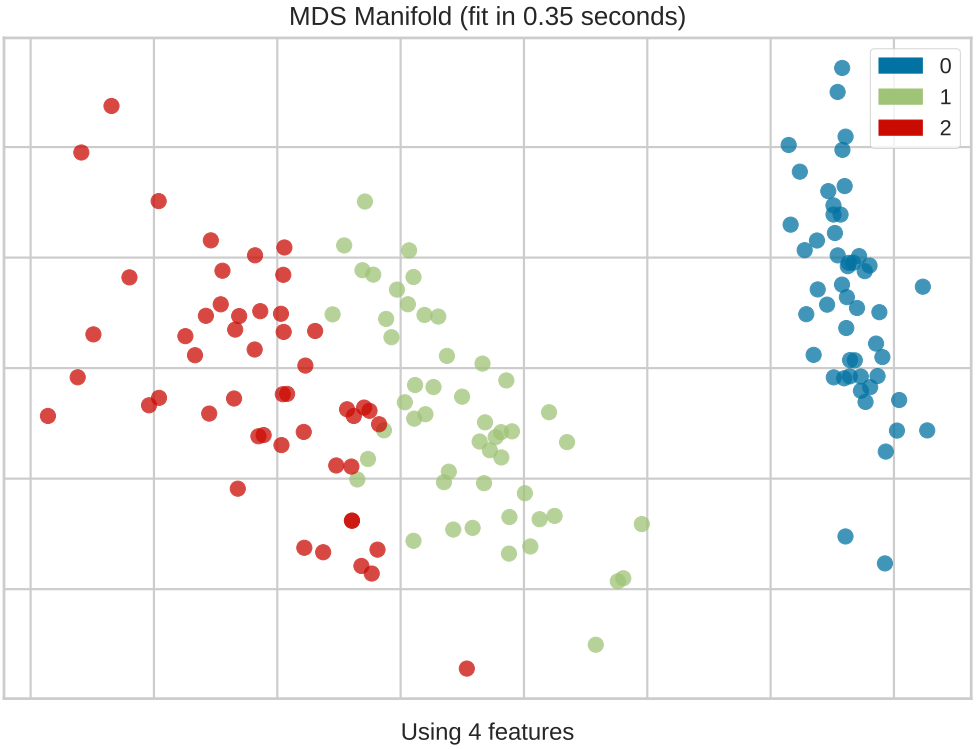
```
from yellowbrick.datasets import load_game
from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import RidgeClassifier
from yellowbrick.classifier import confusion_matrix


X, y = load_game()
X = OneHotEncoder().fit_transform(X)
visualizer = confusion_matrix(RidgeClassifier(), X, y, cmap="Greens")
```

Precision Recall

A `precision_recall_curve` shows the tradeoff between precision and recall for different probability thresholds. More about *Precision-Recall Curves*.

```
from sklearn.naive_bayes import GaussianNB
from yellowbrick.datasets import load_occupancy
from yellowbrick.classifier import precision_recall_curve
```

(continues on next page)



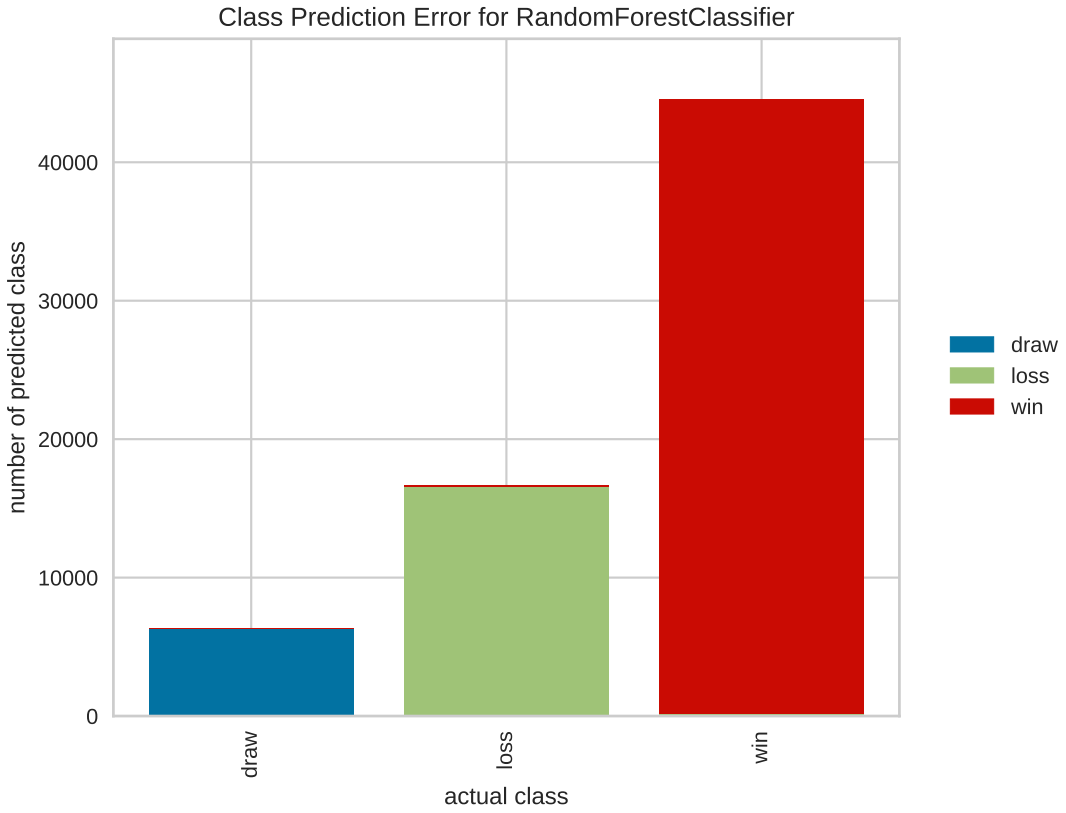
(continued from previous page)

```
X, y = load_occupancy()
visualizer = precision_recall_curve(GaussianNB(), X, y)
```

ROCAUC

A roc_auc plot shows the receiver operator characteristics and area under the curve. More about *ROCAUC*.

```
from yellowbrick.classifier import roc_auc
from yellowbrick.datasets import load_spam
from sklearn.linear_model import LogisticRegression


X, y = load_spam()
visualizer = roc_auc(LogisticRegression(), X, y)
```



Discrimination Threshold

A `discrimination_threshold` plot can help find a threshold that best separates binary classes. More about *Discrimination Threshold*.

```
from yellowbrick.classifier import discrimination_threshold
from sklearn.linear_model import LogisticRegression
from yellowbrick.datasets import load_spam

X, y = load_spam()
visualizer = discrimination_threshold(
    LogisticRegression(multi_class="auto", solver="liblinear"), X, y
)
```

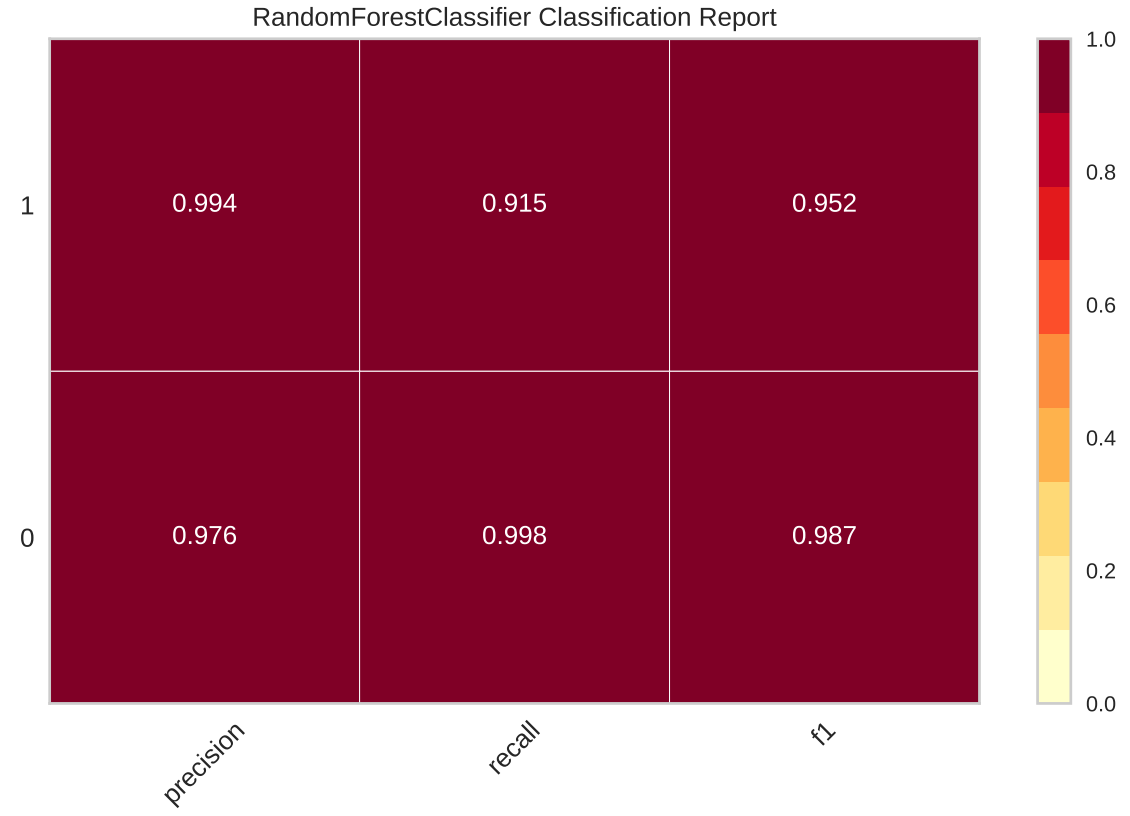
8.4.3 Regression

Residuals Plot

A `residuals_plot` shows the difference in residuals between the training and test data. More about *Residuals Plot*.

```
from sklearn.linear_model import Ridge
from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import residuals_plot
```

(continues on next page)



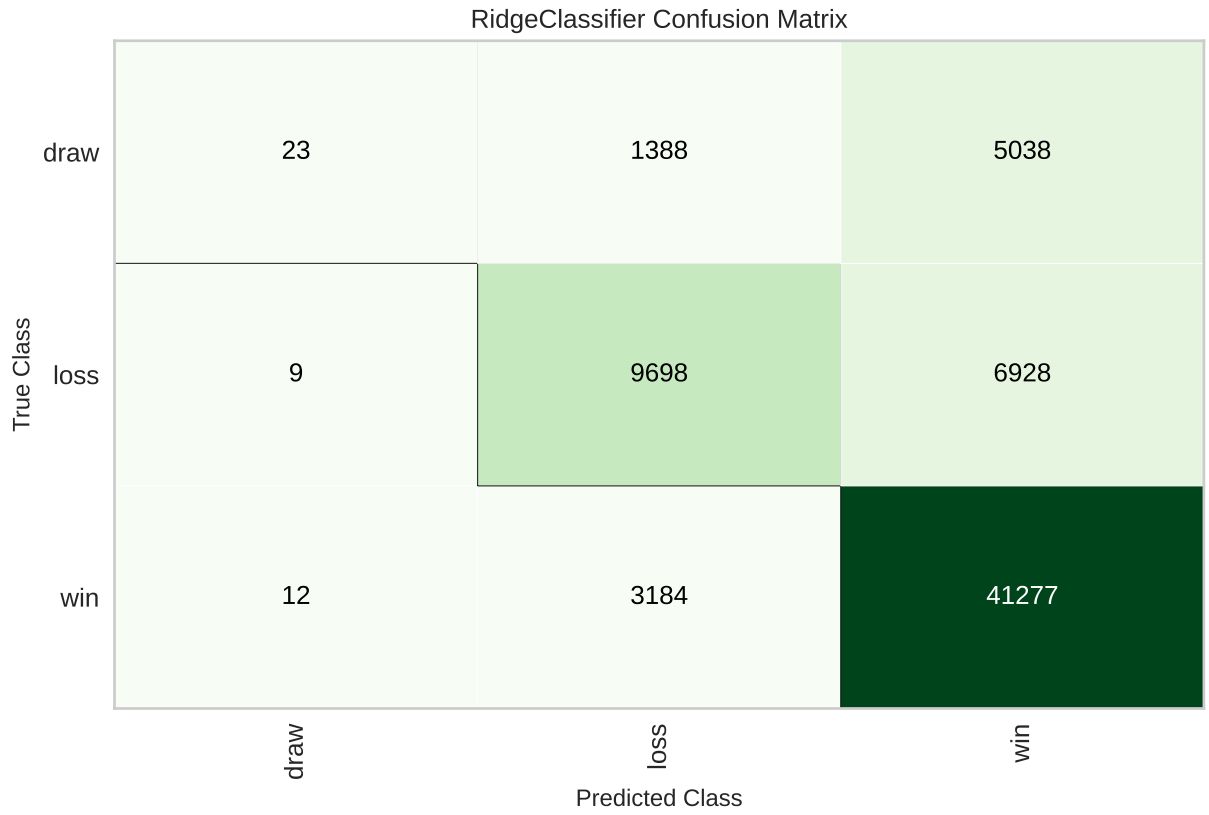
(continued from previous page)

```
X, y = load_concrete()
visualizer = residuals_plot(
    Ridge(), X, y, train_color="maroon", test_color="gold"
)
```

Prediction Error

A `prediction_error` helps find where the regression is making the most errors. More about *Prediction Error Plot*.

```
from sklearn.linear_model import Lasso
from yellowbrick.datasets import load_bikeshare
from yellowbrick.regressor import prediction_error


X, y = load_bikeshare()
visualizer = prediction_error(Lasso(), X, y)
```



Cooks Distance

A `cooks_distance` plot shows the influence of instances on linear regression. More about *Cook's Distance*.

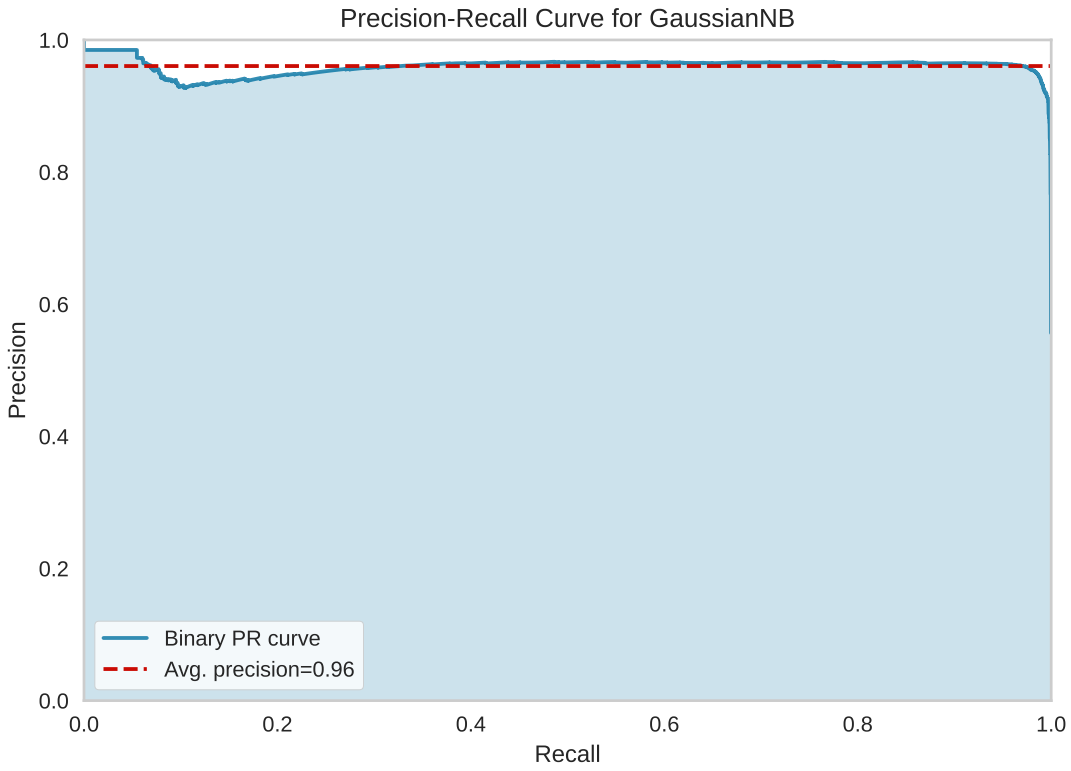
```
from sklearn.datasets import load_diabetes
from yellowbrick.regressor import cooks_distance


X, y = load_diabetes(return_X_y=True)
visualizer = cooks_distance(X, y)
```

8.4.4 Clustering

Silhouette Scores

A `silhouette_visualizer` can help you select `k` by visualizing silhouette coefficient values. More about *Silhouette Visualizer*.

```
from sklearn.cluster import KMeans
from yellowbrick.datasets import load_nfl
from yellowbrick.cluster import silhouette_visualizer

X, y = load_nfl()
visualizer = silhouette_visualizer(KMeans(5, random_state=42), X)
```



Intercluster Distance

A `intercluster_distance` shows size and relative distance between clusters. More about *Intercluster Distance Maps*.

```
from yellowbrick.datasets import load_nfl
from sklearn.cluster import MiniBatchKMeans
from yellowbrick.cluster import intercluster_distance


X, y = load_nfl()
visualizer = intercluster_distance(MiniBatchKMeans(5, random_state=777), X)
```

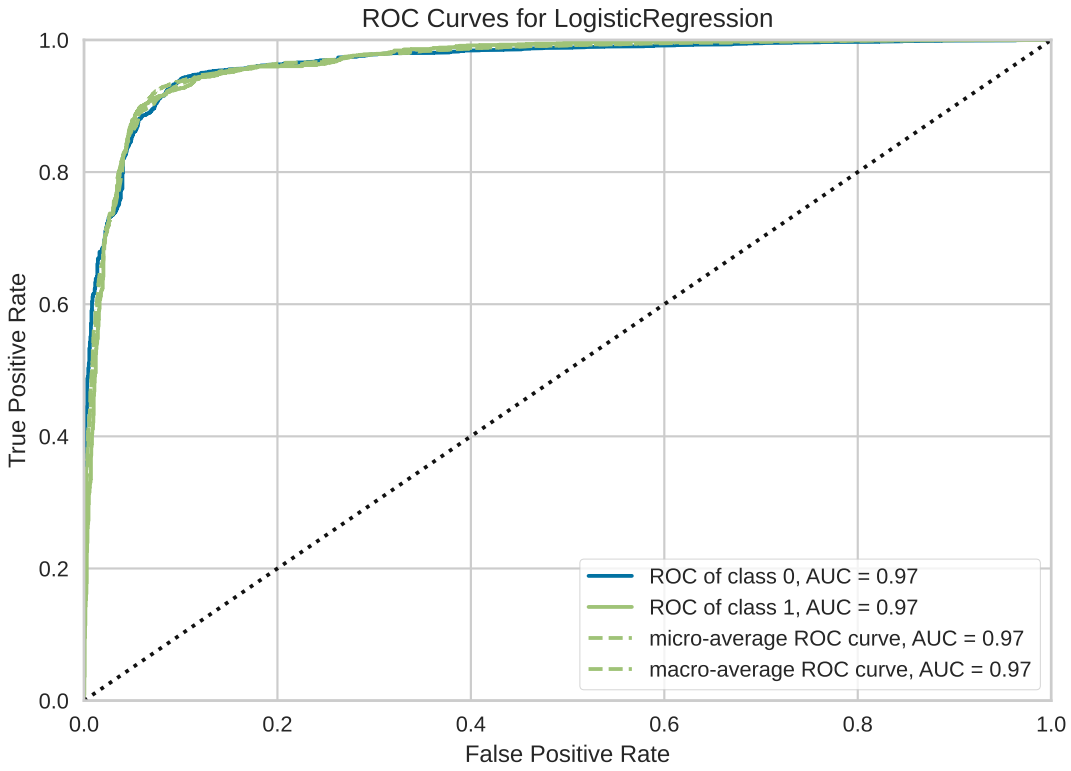
8.4.5 Target Analysis

ClassBalance

The `class_balance` plot can make it easier to see how the distribution of classes may affect the model. More about *Class Balance*.

```
from yellowbrick.datasets import load_game
from yellowbrick.target import class_balance

```

(continues on next page)



(continued from previous page)

```
X, y = load_game()
visualizer = class_balance(y, labels=["draw", "loss", "win"])
```

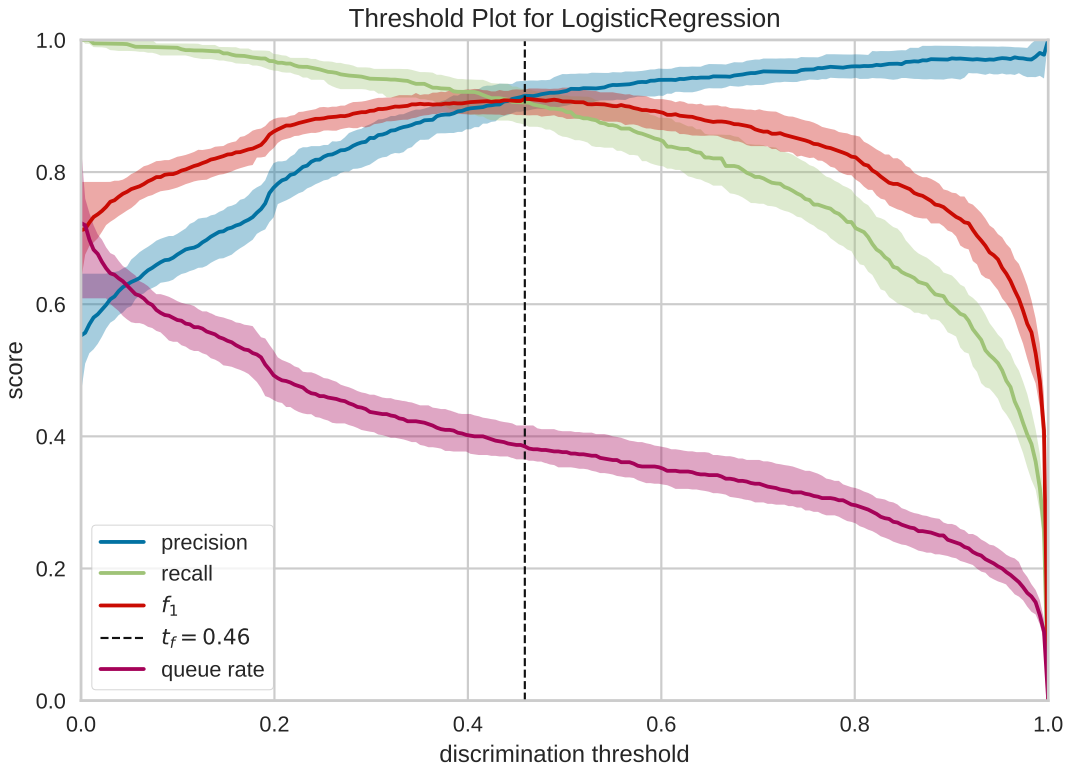
8.5 Contributing

Yellowbrick is an open source project that is supported by a community who will gratefully and humbly accept any contributions you might make to the project. Large or small, any contribution makes a big difference; and if you've never contributed to an open source project before, we hope you will start with Yellowbrick!

Principally, Yellowbrick development is about the addition and creation of *visualizers* — objects that learn from data and create a visual representation of the data or model. Visualizers integrate with scikit-learn estimators, transformers, and pipelines for specific purposes and as a result, can be simple to build and deploy. The most common contribution is a new visualizer for a specific model or model family. We'll discuss in detail how to build visualizers later.

Beyond creating visualizers, there are many ways to contribute:

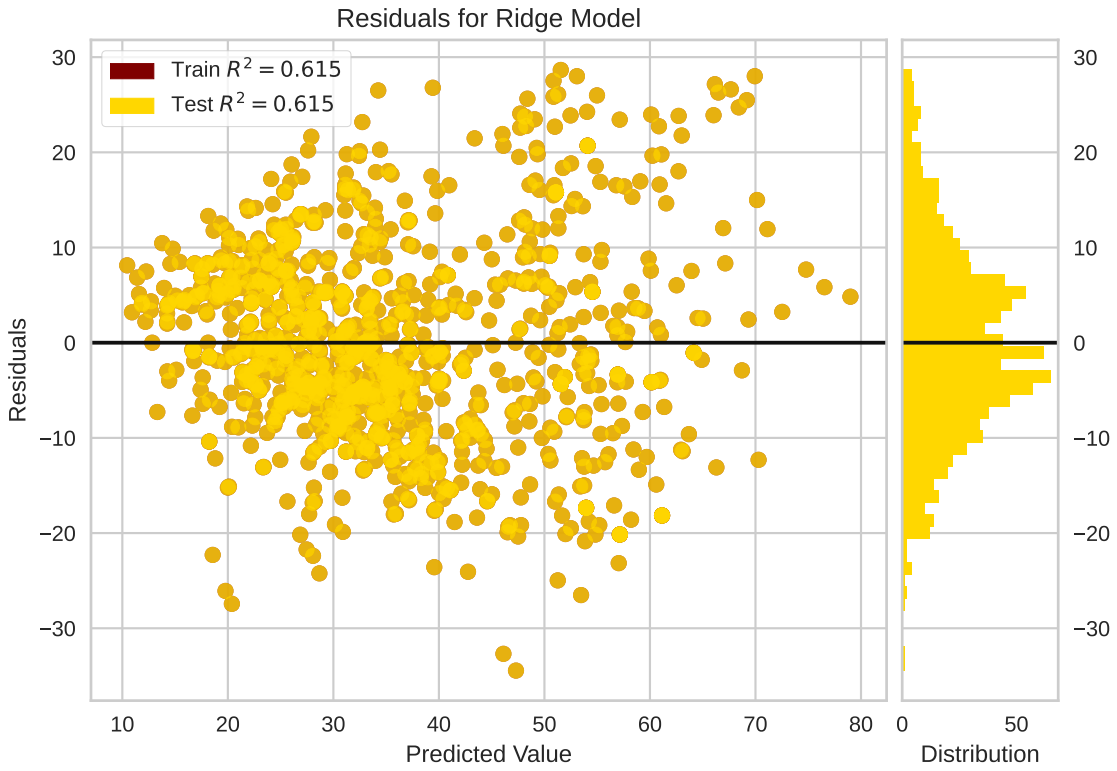
- Submit a bug report or feature request on GitHub issues.
- Contribute an Jupyter notebook to our examples gallery.
- Assist us with *user testing*.
- Add to the documentation or help with our website, scikit-yb.org



- Write unit or integration tests for our project.
- Answer questions on our GitHub issues, mailing list, Stack Overflow, and Twitter.
- Translate our documentation into another language.
- Write a blog post, tweet, or share our project with others.
- Teach someone how to use Yellowbrick.

As you can see, there are lots of ways to get involved and we would be very happy for you to join us! The only thing we ask is that you abide by the principles of openness, respect, and consideration of others as described in our *Code of Conduct*.

Note: If you're unsure where to start, perhaps the best place is to drop the maintainers a note via our mailing list: http://bit.ly/yb-listserv.



8.5.1 Getting Started on GitHub

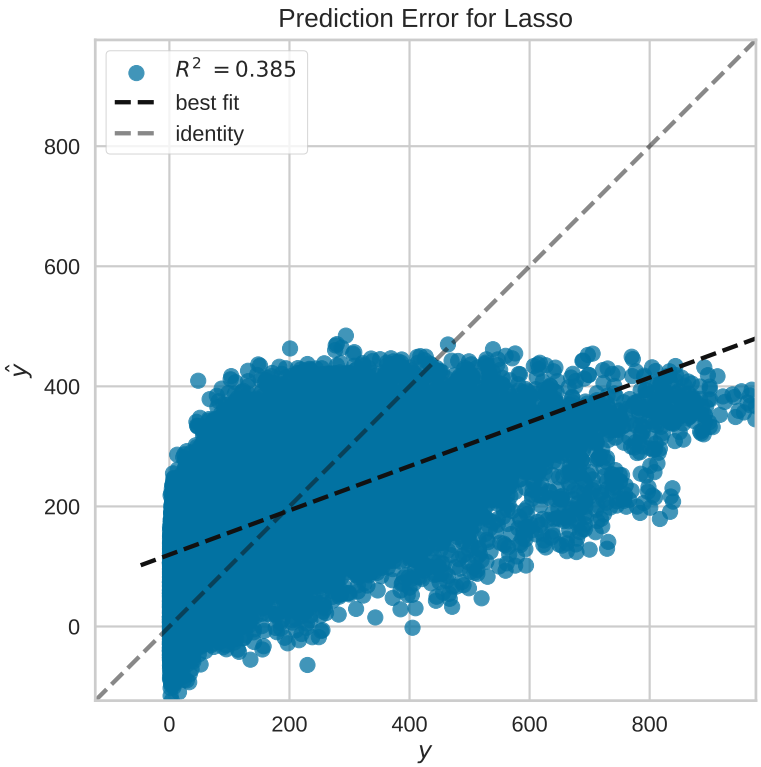
Yellowbrick is hosted on GitHub at https://github.com/DistrictDataLabs/yellowbrick.

The typical workflow for a contributor to the codebase is as follows:

1. **Discover** a bug or a feature by using Yellowbrick.
2. **Discuss** with the core contributors by adding an issue.
3. **Fork** the repository into your own GitHub account.
4. Create a **Pull Request** first thing to connect with us about your task.
5. **Code** the feature, write the tests and documentation, add your contribution.
6. **Review** the code with core contributors who will guide you to a high quality submission.
7. **Merge** your contribution into the Yellowbrick codebase.

We believe that *contribution is collaboration* and therefore emphasize *communication* throughout the open source process. We rely heavily on GitHub's social coding tools to allow us to do this. For instance, we use GitHub's milestone feature to focus our development efforts for each Yellowbrick semester, so be sure to check out the issues associated with our current milestone!

Once you have a good sense of how you are going to implement the new feature (or fix the bug!), you can reach out for feedback from the maintainers by creating a pull request. Ideally, any pull request should be capable of resolution within 6 weeks of being opened. This timeline helps to keep our pull request queue small and allows Yellowbrick to maintain a robust release schedule to give our users the best experience possible. However, the most important thing is



to keep the dialogue going! And if you're unsure whether you can complete your idea within 6 weeks, you should still go ahead and open a PR and we will be happy to help you scope it down as needed.

If we have comments or questions when we evaluate your pull request and receive no response, we will also close the PR after this period of time. Please know that this does not mean we don't value your contribution, just that things go stale. If in the future you want to pick it back up, feel free to address our original feedback and to reference the original PR in a new pull request.

Note: Please note that if we feel your solution has not been thought out in earnest, or if the PR is not aligned with our current milestone goals, we may reach out to ask that you close the PR so that we can prioritize reviewing the most critical feature requests and bug fixes.

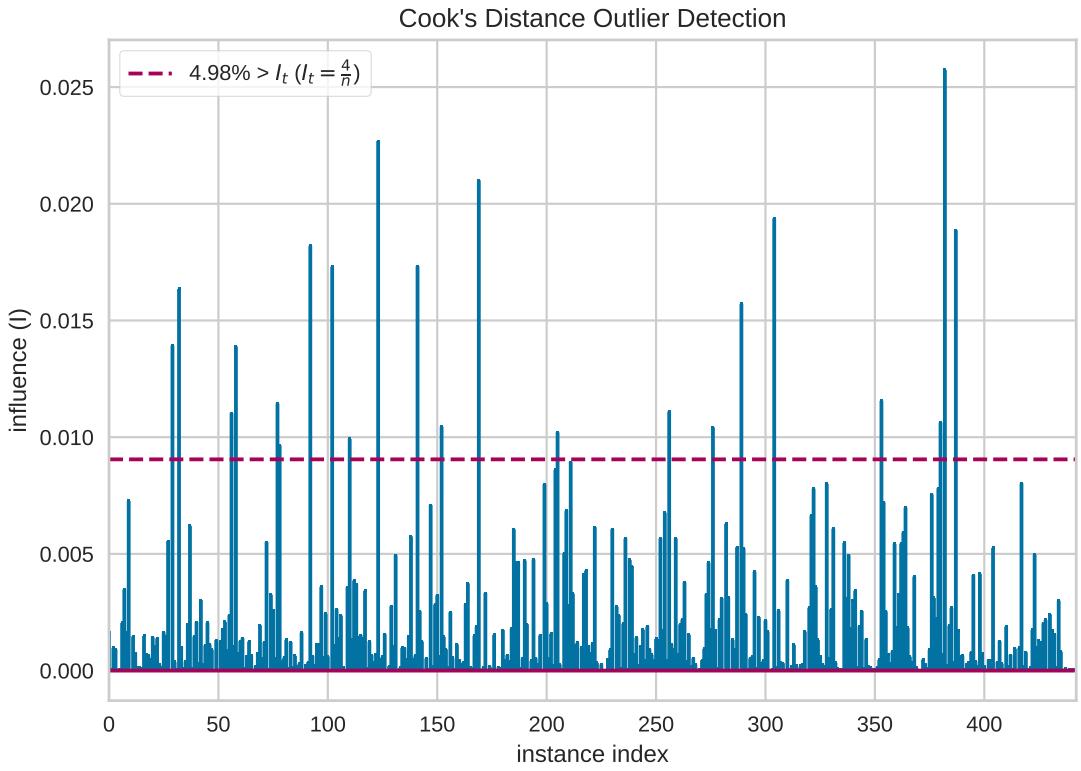
Forking the Repository

The first step is to fork the repository into your own account. This will create a copy of the codebase that you can edit and write to. Do so by clicking the **"fork"** button in the upper right corner of the Yellowbrick GitHub page.

Once forked, use the following steps to get your development environment set up on your computer:

1. Clone the repository.

   After clicking the fork button, you should be redirected to the GitHub page of the repository in your user account. You can then clone a copy of the code to your local machine.:




```
$ git clone https://github.com/[YOURUSERNAME]/yellowbrick
$ cd yellowbrick
```

2. Create a virtual environment.

   Yellowbrick developers typically use virtualenv (and virtualenvwrapper), pyenv or conda envs in order to manage their Python version and dependencies. Using the virtual environment tool of your choice, create one for Yellowbrick. Here's how with virtualenv:

   ```
   $ virtualenv venv
   ```

   To develop with a conda environment, the conda-forge channel is needed to install some testing dependencies. The following command adds the channel with the highest priority:

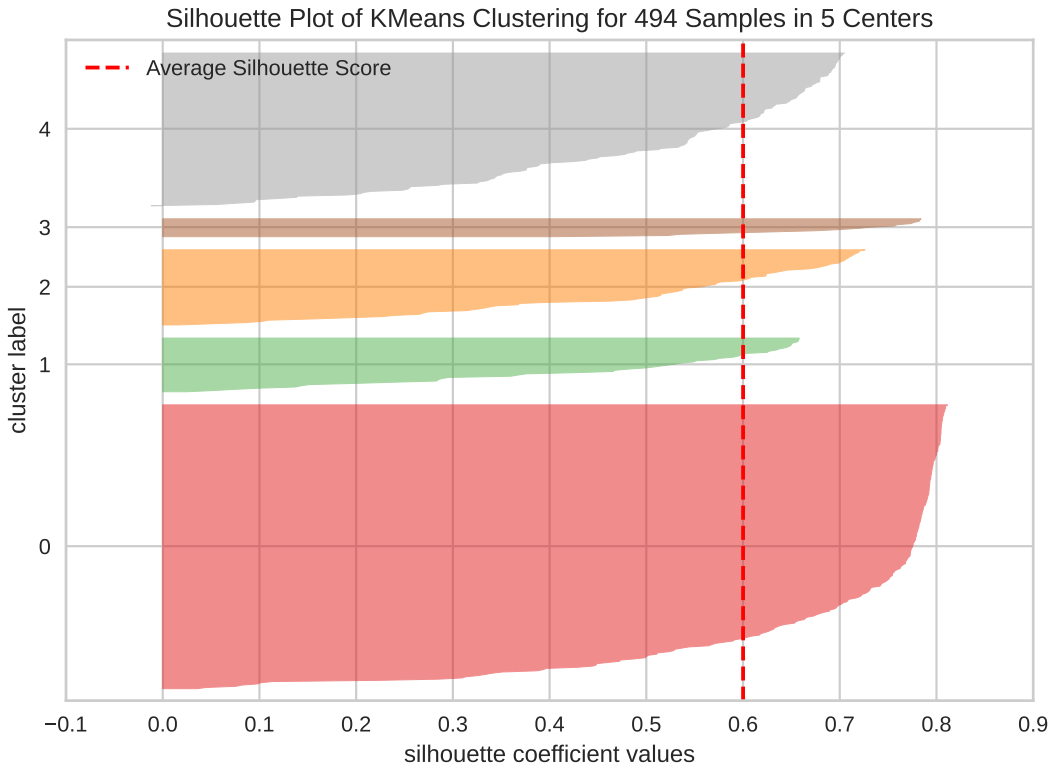
   ```
   $ conda config --add channels conda-forge
   ```

3. Install dependencies.

   Yellowbrick's dependencies are in the `requirements.txt` document at the root of the repository. Open this file and uncomment any dependencies marked as for development only. Then install the package in editable mode:

   ```
   $ pip install -e .
   ```

   This will add Yellowbrick to your PYTHONPATH so that you don't need to reinstall it each time you make a change during development.



Note that there may be other dependencies required for development and testing; you can simply install them with `pip`. For example to install the additional dependencies for building the documentation or to run the test suite, use the `requirements.txt` files in those directories:

```
$ pip install -r tests/requirements.txt
$ pip install -r docs/requirements.txt
```

4. (Optional) Set up pre-commit hooks.

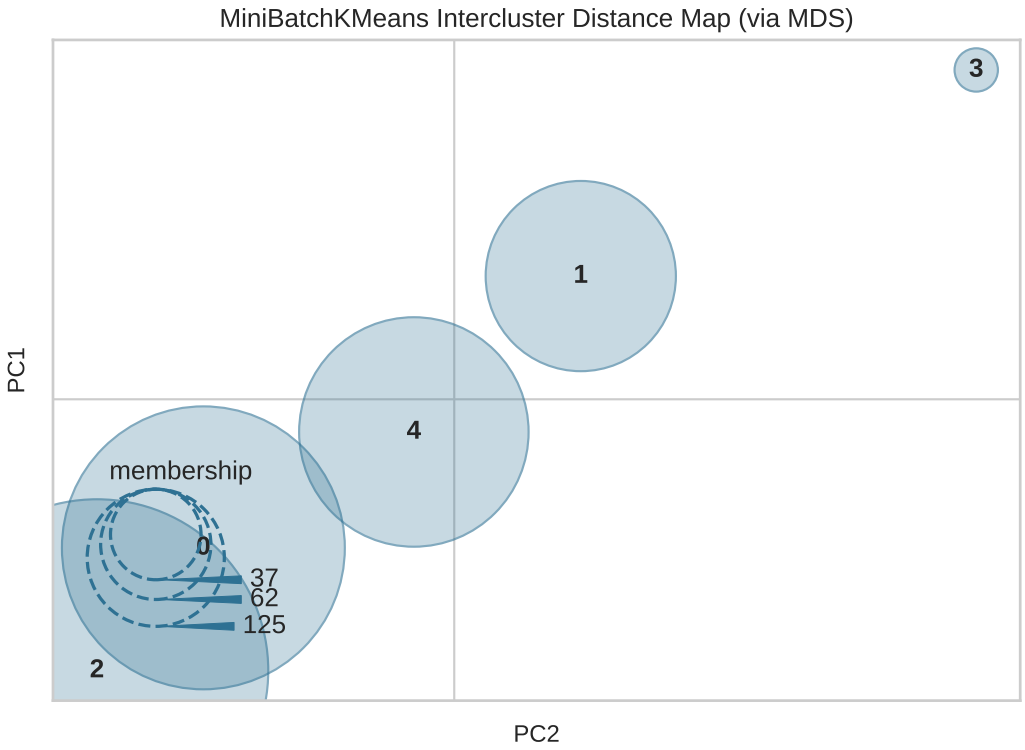
   When opening a PR in the Yellowbrick repository, a series of checks will be run on your contribution, some of which lint and look at the formatting of your code. These may indicate some changes that need to be made before your contribution can be reviewed. You can set up pre-commit hooks to run these checks locally upon running `git commit` to ensure your contribution will pass formatting and linting checks. To set this up, you will need to uncomment the pre-commit line in `requirements.txt` and then run the following commands:

   ```
   $ pip install -r requirements.txt
   $ pre-commit install
   ```

   The next time you run `git commit` in the Yellowbrick repository, the checks will automatically run.

5. Switch to the develop branch.

   The Yellowbrick repository has a `develop` branch that is the primary working branch for contributions. It is probably already the branch you're on, but you can make sure and switch to it as follows:




```
$ git fetch
$ git checkout develop
```

At this point you're ready to get started writing code.

Branching Convention

The Yellowbrick repository is set up in a typical production/release/development cycle as described in "A Successful Git Branching Model." The primary working branch is the `develop` branch. This should be the branch that you are working on and from, since this has all the latest code. The `master` branch contains the latest stable version and release, which is pushed to PyPI. No one but core contributors will generally push to master.

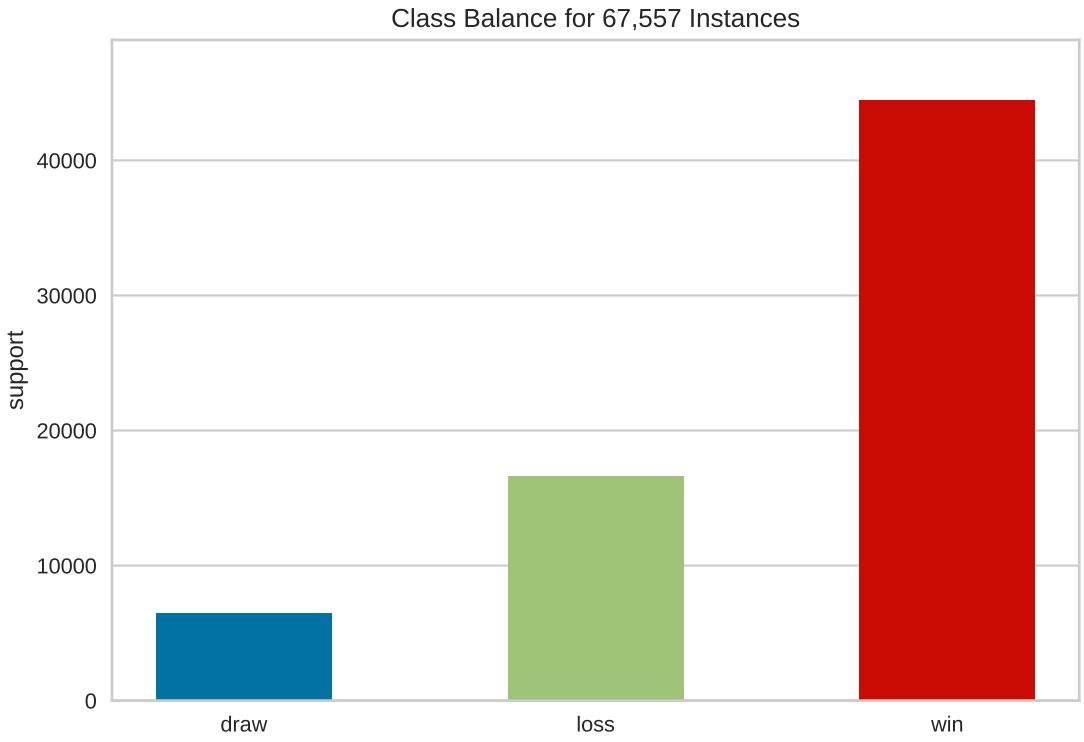
You should work directly in your fork. In order to reduce the number of merges (and merge conflicts) we kindly request that you utilize a feature branch off of `develop` to work in:

```
$ git checkout -b feature-myfeature develop
```

We also recommend setting up an `upstream` remote so that you can easily pull the latest development changes from the main Yellowbrick repository (see configuring a remote for a fork). You can do that as follows:

```
$ git remote add upstream https://github.com/DistrictDataLabs/yellowbrick.git
$ git remote -v
origin    https://github.com/YOUR_USERNAME/YOUR_FORK.git (fetch)
origin    https://github.com/YOUR_USERNAME/YOUR_FORK.git (push)
```

(continues on next page)



(continued from previous page)

```
upstream  https://github.com/DistrictDataLabs/yellowbrick.git (fetch)
upstream  https://github.com/DistrictDataLabs/yellowbrick.git (push)
```

When you're ready, request a code review for your pull request.

Pull Requests

A pull request (PR) is a GitHub tool for initiating an exchange of code and creating a communication channel for Yellowbrick maintainers to discuss your contribution. In essence, you are requesting that the maintainers merge code from your forked repository into the develop branch of the primary Yellowbrick repository. Once completed, your code will be part of Yellowbrick!

When starting a Yellowbrick contribution, *open the pull request as soon as possible*. We use your PR issue page to discuss your intentions and to give guidance and direction. Every time you push a commit into your forked repository, the commit is automatically included with your pull request, therefore we can review as you code. The earlier you open a PR, the more easily we can incorporate your updates, we'd hate for you to do a ton of work only to discover someone else already did it or that you went in the wrong direction and need to refactor.

Note: For a great example of a pull request for a new feature visualizer, check out this one by Carlo Morales.

Opening a Pull Request

When you open a pull request, ensure it is from your forked repository to the develop branch of github.com/districtdatalabs/yellowbrick; we will not merge a PR into the master branch. Title your Pull Request so that it is easy to understand what you're working on at a glance. Also be sure to include a reference to the issue that you're working on so that correct references are set up.

Note: All pull requests should be into the `yellowbrick/develop` branch from your forked repository.

After you open a PR, you should get a message from one of the maintainers. Use that time to discuss your idea and where best to implement your work. Feel free to go back and forth as you are developing with questions in the comment thread of the PR. Once you are ready, please ensure that you explicitly ping the maintainer to do a code review. Before code review, your PR should contain the following:

1. Your code contribution
2. Tests for your contribution
3. Documentation for your contribution
4. A PR comment describing the changes you made and how to use them
5. A PR comment that includes an image/example of your visualizer

At this point your code will be formally reviewed by one of the contributors. We use GitHub's code review tool, starting a new code review and adding comments to specific lines of code as well as general global comments. Please respond to the comments promptly, and don't be afraid to ask for help implementing any requested changes! You may have to go back and forth a couple of times to complete the code review.

When the following is true:

1. Code is reviewed by at least one maintainer
2. Continuous Integration tests have passed
3. Code coverage and quality have not decreased
4. Code is up to date with the yellowbrick develop branch

Then we will "Squash and Merge" your contribution, combining all of your commits into a single commit and merging it into the `develop` branch of Yellowbrick. Congratulations! Once your contribution has been merged into master, you will be officially listed as a contributor.

After Your Pull Request is Merged

After your pull request is merged, you should update your local fork, either by pulling from `upstream develop`:

```
$ git checkout develop
$ git pull upstream develop
$ git push origin develop
```

or by manually merging your feature into your fork's `develop` branch.:

```
$ git checkout develop
$ git merge --no-ff feature-myfeature
$ git push origin develop
```

Then you can safely delete the old feature branch, both locally and on GitHub. Now head back to the backlog and checkout another issue!

8.5.2 Developing Visualizers

In this section, we'll discuss the basics of developing visualizers. This of course is a big topic, but hopefully these simple tips and tricks will help make sense. First thing though, check out this presentation that we put together on yellowbrick development, it discusses the expected user workflow, our integration with scikit-learn, our plans and roadmap, etc:

One thing that is necessary is a good understanding of scikit-learn and Matplotlib. Because our API is intended to integrate with scikit-learn, a good start is to review "APIs of scikit-learn objects" and "rolling your own estimator". In terms of matplotlib, use Yellowbrick's guide *Effective Matplotlib*. Additional resources include Nicolas P. Rougier's Matplotlib tutorial and Chris Moffitt's Effectively Using Matplotlib.

Visualizer API

There are two basic types of Visualizers:

- **Feature Visualizers** are high dimensional data visualizations that are essentially transformers.
- **Score Visualizers** wrap a scikit-learn regressor, classifier, or clusterer and visualize the behavior or performance of the model on test data.

These two basic types of visualizers map well to the two basic estimator objects in scikit-learn:

- **Transformers** take input data and return a new data set.
- **Models** are fit to training data and can make predictions.

The scikit-learn API is object oriented, and estimators are initialized with parameters by instantiating their class. Hyperparameters can also be set using the `set_attrs()` method and retrieved with the corresponding `get_attrs()` method. All scikit-learn estimators have a `fit(X, y=None)` method that accepts a two dimensional data array, `X`, and optionally a vector `y` of target values. The `fit()` method trains the estimator, making it ready to transform data or make predictions. Transformers have an associated `transform(X)` method that returns a new dataset, `Xprime` and models have a `predict(X)` method that returns a vector of predictions, `yhat`. Models may also have a `score(X, y)` method that evaluate the performance of the model.

Visualizers interact with scikit-learn objects by intersecting with them at the methods defined above. Specifically, visualizers perform actions related to `fit()`, `transform()`, `predict()`, and `score()` then call a `draw()` method which initializes the underlying figure associated with the visualizer. The user calls the visualizer's `show()` method, which in turn calls a `finalize()` method on the visualizer to draw legends, titles, etc. and then `show()` renders the figure. The Visualizer API is therefore:

- `draw()`: add visual elements to the underlying axes object
- `finalize()`: prepare the figure for rendering, adding final touches such as legends, titles, axis labels, etc.
- `show()`: render the figure for the user (or saves it to disk).

Creating a visualizer means defining a class that extends `Visualizer` or one of its subclasses, then implementing several of the methods described above. A barebones implementation is as follows:

```
import matplotlib.pyplot as plt

from yellowbrick.base import Visualizer

class MyVisualizer(Visualizer):

```

(continues on next page)

(continued from previous page)

```
    def __init__(self, ax=None, **kwargs):
        super(MyVisualizer, self).__init__(ax, **kwargs)

    def fit(self, X, y=None):
        self.draw(X)
        return self

    def draw(self, X):
        self.ax.plt(X)
        return self.ax

    def finalize(self):
        self.set_title("My Visualizer")
```

This simple visualizer simply draws a line graph for some input dataset X, intersecting with the scikit-learn API at the `fit()` method. A user would use this visualizer in the typical style:

```
visualizer = MyVisualizer()
visualizer.fit(X)
visualizer.show()
```

Score visualizers work on the same principle but accept an additional required `estimator` argument. Score visualizers wrap the model (which can be either fitted or unfitted) and then pass through all attributes and methods through to the underlying model, drawing where necessary.

```
from yellowbrick.base import ScoreVisualizer

class MyScoreVisualizer(ScoreVisualizer):

    def __init__(self, estimator, ax=None, **kwargs):
        super(MyScoreVisualizer, self).__init__(estimator, ax=ax, **kwargs)

    def fit(self, X_train, y_train=None):
        # Fit the underlying model
        super(MyScoreVisualizer, self).fit(X_train, y_train)
        self.draw(X_train, y_train)
        return self

    def score(self, X_test, y_test):
        # Score the underlying model
        super(MyScoreVisualizer, self).fit(X_train, y_train)
        self.draw(X_test, y_test)
        return self.score_

    def draw(self, X, y):
        self.ax.scatter(X, c=y)
        return self.ax

    def finalize(self):
        self.set_title("My Score Visualizer")
```

Note that the calls to `super` in the above code ensure that the base functionality (e.g. fitting a model and computing the score) are required to ensure the visualizer is consistent with other visualizers.

Datasets

Yellowbrick gives easy access to several datasets that are used for the examples in the documentation and testing. These datasets are hosted in our CDN and must be downloaded for use. Typically, when a user calls one of the data loader functions, e.g. `load_bikeshare()` the data is automatically downloaded if it's not already on the user's computer. However, for development and testing, or if you know you will be working without internet access, it might be easier to simply download all the data at once.

The data downloader script can be run as follows:

```
$ python -m yellowbrick.download
```

This will download the data to the fixtures directory inside of the Yellowbrick site packages. You can specify the location of the download either as an argument to the downloader script (use `--help` for more details) or by setting the `$YELLOWBRICK_DATA` environment variable. This is the preferred mechanism because this will also influence how data is loaded in Yellowbrick.

Note that developers who have downloaded data from Yellowbrick versions earlier than v1.0 may experience some problems with the older data format. If this occurs, you can clear out your data cache as follows:

```
$ python -m yellowbrick.download --cleanup
```

This will remove old datasets and download the new ones. You can also use the `--no-download` flag to simply clear the cache without re-downloading data. Users who are having difficulty with datasets can also use this or they can uninstall and reinstall Yellowbrick using `pip`.

Testing

The test package mirrors the yellowbrick package in structure and also contains several helper methods and base functionality. To add a test to your visualizer, find the corresponding file to add the test case, or create a new test file in the same place you added your code.

Visual tests are notoriously difficult to create — how do you test a visualization or figure? Moreover, testing scikit-learn models with real data can consume a lot of memory. Therefore the primary test you should create is simply to test your visualizer from end to end and make sure that no exceptions occur. To assist with this, we have a helper, `VisualTestCase`. Create your tests as follows:

```
import pytest

from tests.base import VisualTestCase
from yellowbrick.datasets import load_occupancy

class MyVisualizerTests(VisualTestCase):

    def test_my_visualizer(self):
        """
        Test MyVisualizer on a real dataset
        """
        # Load the occupancy dataset
        X, y = load_occupancy()

        try:
            visualizer = MyVisualizer()
            assert visualizer.fit(X, y) is visualizer, "fit should return self"
```

(continues on next page)

(continued from previous page)

```
            visualizer.finalize()
        except Exception as e:
            pytest.fail("my visualizer didn't work: {}".format(e))
```

This simple test case is an excellent start to a larger test package and we recommend starting with this test as you develop your visualizer. Once you've completed the development and prototyping you can start to include *test fixtures* and test various normal use cases and edge cases with unit tests, then build *image similarity tests* to more thoroughly define the integration tests.

Note: In tests you should not call `visualizer.show()` because this will call `plt.show()` and trigger a matplotlib warning that the visualization cannot be displayed using the test backend `Agg`. Calling `visualizer.finalize()` instead should produce the full image and make the tests faster and more readable.

Running the Test Suite

To run the test suite, first install the testing dependencies that are located in the *tests* folder as follows:

```
$ pip install -r tests/requirements.txt
```

The required dependencies for the test suite include testing utilities and libraries such as pandas and nltk that are not included in the core dependencies.

Tests can be run as follows from the project root:

```
$ pytest
```

The `pytest` function is configured via `setup.cfg` with the correct arguments and runner, and therefore must be run from the project root. You can also use `make test` but this simply runs the `pytest` command.

The tests do take a while to run, so during normal development it is recommended that you only run the tests for the test file you're writing:

```
$ pytest tests/test_package/test_module.py
```

This will greatly simplify development and allow you to focus on the changes that you're making!

Image Comparison Tests

Writing an image based comparison test is only a little more difficult than the simple testcase presented above. We have adapted matplotlib's image comparison test utility into an easy to use assert method : `self.assert_images_similar(visualizer)`

The main consideration is that you must specify the "baseline", or expected, image in the `tests/baseline_images/` folder structure.

For example, create your test function located in `tests/test_regressor/test_myvisualizer.py` as follows:

```
from tests.base import VisualTestCase

class MyVisualizerTests(VisualTestCase):

```

(continues on next page)

(continued from previous page)

```
    def test_my_visualizer_output(self):
        visualizer = MyVisualizer()
        visualizer.fit(X)
        visualizer.finalize()
        self.assert_images_similar(visualizer)
```

The first time this test is run, there will be no baseline image to compare against, so the test will fail. Copy the output images (in this case `tests/actual_images/test_regressor/test_myvisualizer/test_my_visualizer_output.png`) to the correct subdirectory of baseline_images tree in the source directory (in this case `tests/baseline_images/test_regressor/test_myvisualizer/test_my_visualizer_output.png`). Put this new file under source code revision control (with git add). When rerunning the tests, they should now pass.

We also have a helper script, `tests/images.py` to clean up and manage baseline images automatically. It is run using the `python -m` command to execute a module as main, and it takes as an argument the path to your *test file*. To copy the figures as above:

```
$ python -m tests.images tests/test_regressor/test_myvisualizer.py
```

This will move all related test images from `actual_images` to `baseline_images` on your behalf (note you'll have had to run the tests at least once to generate the images). You can also clean up images from both actual and baseline as follows:

```
$ python -m tests.images -C tests/test_regressor/test_myvisualizer.py
```

This is useful particularly if you're stuck trying to get an image comparison to work. For more information on the images helper script, use `python -m tests.images --help`.

Finally, to reiterate a note from above; make sure that you do not call your visualizer's `show()` method, instead using `finalize()` directly. If you call `show()`, it will in turn call `plt.show()` which will issue a warning due to the fact that the tests use the `Agg` backend. When testing quick methods you should pass the `show=False` argument to the quick method as follows:

```
from tests.base import VisualTestCase

class MyVisualizerTests(VisualTestCase):

    def test_my_visualizer_quickmethod(self):
        visualizer = my_visualizer_quickmethod(X, show=False)
        self.assert_images_similar(visualizer)
```

Generally speaking, testing quick methods is identical to testing Visualizers except for the method of interaction because the quick method will return the visualizer or the axes object.

Test Fixtures

Often, you will need a controlled dataset to test your visualizer as specifically as possible. To do this, we recommend that you make use of pytest fixtures and scikit-learn's generated datasets. Together these tools ensure that you have complete control over your test fixtures and can test different user scenarios as precisely as possible. For example, consider the case where we want to test both a binary and a multiclass dataset for a classification score visualizer.

```
from tests.fixtures import Dataset, Split

from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split as tts

@pytest.fixture(scope="class")
def binary(request):
    """
    Creates a random binary classification dataset fixture
    """
    X, y = make_classification(
        n_samples=500,
        n_features=20,
        n_informative=8,
        n_redundant=2,
        n_classes=2,
        n_clusters_per_class=3,
        random_state=2001,
    )

    X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2, random_state=42)

    dataset = Dataset(Split(X_train, X_test), Split(y_train, y_test))
    request.cls.binary = dataset
```

In this example, we make use of `sklearn.datasets.make_classification()` to randomly generate exactly the dataset that we'd like, in this case a dataset with 2 classes and enough variability so as to be interesting. Because we're using this with a score visualizer, it is helpful to divide this into train and test splits. The `Dataset` and `Split` objects in `tests.fixtures` are namedtuples that allow you to easily access `X` and `y` properties on the dataset and `train` and `test` properties on the split. Creating a dataset this way means we can access `dataset.X.train` and `dataset.y.test` easily in our test functions.

Similarly, we can create a custom multiclass function as well:

```
@pytest.fixture(scope="class")
def multiclass(request):
    """
    Creates a random multiclass classification dataset fixture
    """
    X, y = make_classification(
        n_samples=500,
        n_features=20,
        n_informative=8,
        n_redundant=2,
        n_classes=6,
        n_clusters_per_class=3,
        random_state=87,
```

(continues on next page)

(continued from previous page)

```
    )

    X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2, random_state=93)

    dataset = Dataset(Split(X_train, X_test), Split(y_train, y_test))
    request.cls.multiclass = dataset
```

Note: Fixtures that are added to `conftest.py` are available to tests in the same directory or a subdirectory as `conftest.py`. This is special pytest magic since fixtures are identified by strings. Note that the two above example fixtures are in `tests/test_classifier/conftest.py` so you can use these exactly in the `tests/test_classifier` directory without having to create new fixtures.

To use these fixtures with a `VisualTestCase` you must decorate the test class with the fixture. Once done, the fixture will be *generated once per class* and stored in the `request.cls.<property>` variable. Here's how to use the above fixtures:

```
@pytest.mark.usefixtures("binary", "multiclass")
class TestMyScoreVisualizer(VisualTestCase):

    def test_binary(self):
        oz = MyScoreVisualizer()
        assert oz.fit(self.binary.X.train, self.binary.y.train) is oz
        assert 0.0 <= oz.score(self.binary.X.test, self.binary.y.test) <= 1.0
        oz.finalize()

        self.assert_images_similar(oz)
```

In the above test examples, we showed the use of the yellowbrick dataset loaders, e.g. `load_occupancy()`. You should feel free to use those datasets and the scikit-learn datasets for tests, particularly for integration tests (described next). The use of the generated datasets and fixtures allows a lot of control over what is being tested and ensures that the tests run as quickly as possible, therefore please use fixtures for the majority of test cases.

Integration Tests

The majority of test cases will use generated test fixtures as described above. But as a visualizer is concluded, it is important to create two "integration tests" that use real-world data in the form of Pandas and numpy arrays from the yellowbrick datasets loaders. These tests often take the following form:

```
try:
    import pandas as pd
except ImportError:
    pd = None

class MyVisualizerTests(VisualTestCase):

    @pytest.mark.skipif(pd is None, reason="test requires pandas")
    def test_pandas_integration(self):
        """
        Test with Pandas DataFrame and Series input
        """
```

(continues on next page)

(continued from previous page)

```
        X, y = load_occupancy(return_datset=True).to_pandas()
        oz = MyScoreVisualizer().fit(X, y)
        oz.finalize()
        self.assert_images_similar(oz)

    def test_numpy_integration(self):
        """
        Test with numpy arrays
        """
        X, y = load_occupancy(return_datset=True).to_numpy()
        oz = MyScoreVisualizer().fit(X, y)
        oz.finalize()
        self.assert_images_similar(oz)
```

These tests often offer the most complications with your visual test cases, so be sure to reserve them for the last tests you create!

Documentation

Yellowbrick uses Sphinx to build our documentation. The advantages of using Sphinx are many; we can more directly link to the documentation and source code of other projects like Matplotlib and scikit-learn using intersphinx. In addition, docstrings used to describe Yellowbrick visualizers can be automatically included when the documentation is built via autodoc.

To take advantage of these features, our documentation must be written in reStructuredText (or "rst"). reStructuredText is similar to markdown, but not identical, and does take some getting used to. For instance, styling for things like codeblocks, external hyperlinks, internal cross references, notes, and fixed-width text are all unique in rst.

If you would like to contribute to our documentation and do not have prior experience with rst, we recommend you make use of these resources:

- A reStructuredText Primer
- rst notes and cheatsheet
- Using the plot directive

Docstrings

The initial documentation for your visualizer will be a well structured docstring. Yellowbrick uses Sphinx to build documentation, therefore docstrings should be written in reStructuredText in numpydoc format (similar to scikit-learn). The primary location of your docstring should be right under the class definition, here is an example:

```
class MyVisualizer(Visualizer):
    """Short description of MyVisualizer

    This initial section should describe the visualizer and what
    it's about, including how to use it. Take as many paragraphs
    as needed to get as much detail as possible.

    In the next section describe the parameters to __init__.

    Parameters
```

(continues on next page)

(continued from previous page)

```
    ----------
    model : a scikit-learn regressor
        Should be an instance of a regressor, and specifically one whose name
        ends with "CV" otherwise a will raise a YellowbrickTypeError exception
        on instantiation. To use non-CV regressors see:
        ``ManualAlphaSelection``.

    ax : matplotlib Axes, default: None
        The axes to plot the figure on. If None is passed in the current axes
        will be used (or generated if required).

    kwargs : dict
        Keyword arguments that are passed to the base class and may influence
        the visualization as defined in other Visualizers.

    Attributes
    ----------
    score_ : float
        The coefficient of determination that is learned during the visual
        diagnostic, saved for reference after the image has been created.

    Examples
    --------
    >>> model = MyVisualizer()
    >>> model.fit(X)
    >>> model.show()

    Notes
    -----
    In the notes section specify any gotchas or other info.
    """
```

When your visualizer is added to the API section of the documentation, this docstring will be rendered in HTML to show the various options and functionality of your visualizer!

API Documentation Page

To add the visualizer to the documentation it needs to be added to the `docs/api` folder in the correct subdirectory. For example if your visualizer is a model score visualizer related to regression it would go in the `docs/api/regressor` subdirectory. Add your file named after your module, e.g. `docs/api/regressor/mymodule.rst`. If you have a question where your documentation should be located, please ask the maintainers via your pull request, we'd be happy to help!

There are quite a few examples in the documentation on which you can base your files of similar types. The primary format for the API section is as follows:

```
.. -*- mode: rst -*-

My Visualizer
=============

A brief introduction to my visualizer and how it is useful in the machine learning␣
```

(continues on next page)

(continued from previous page)

```
↪process.

.. plot::
    :context: close-figs
    :include-source: False
    :alt: Example using MyVisualizer

    visualizer = MyVisualizer(LinearRegression())

    visualizer.fit(X, y)
    g = visualizer.show()

Discussion about my visualizer and some interpretation of the above plot.


API Reference
-------------

.. automodule:: yellowbrick.regressor.mymodule
    :members: MyVisualizer
    :undoc-members:
    :show-inheritance:
```

This is a pretty good structure for a documentation page; a brief introduction followed by a code example with a visualization included using the plot directive. This will render the `MyVisualizer` image in the document along with links for the complete source code, the png, and the pdf versions of the image. It will also have the "alt-text" (for screen-readers) and will not display the source because of the `:include-source:` option. If `:include-source:` is omitted, the source will be included by default.

The primary section is wrapped up with a discussion about how to interpret the visualizer and use it in practice. Finally the `API Reference` section will use `automodule` to include the documentation from your docstring.

At this point there are several places where you can list your visualizer, but to ensure it is included in the documentation it *must be listed in the TOC of the local index*. Find the `index.rst` file in your subdirectory and add your rst file (without the `.rst` extension) to the `..toctree::` directive. This will ensure the documentation is included when it is built.

Building the Docs

Speaking of, you can build your documentation by changing into the `docs` directory and running `make html`, the documentation will be built and rendered in the `_build/html` directory. You can view it by opening `_build/html/index.html` then navigating to your documentation in the browser.

There are several other places that you can list your visualizer including:

- `docs/index.rst` for a high level overview of our visualizers
- `DESCRIPTION.rst` for inclusion on PyPI
- `README.md` for inclusion on GitHub

Please ask for the maintainer's advice about how to include your visualizer in these pages.

Generating the Gallery

In v1.0, we have adopted Matplotlib's plot directive which means that the majority of the images generated for the documentation are generated automatically. One exception is the gallery; the images for the gallery must still be generated manually.

If you have contributed a new visualizer as described in the above section, please also add it to the gallery, both to docs/gallery.py and to docs/gallery.rst. (Make sure you have already installed Yellowbrick in editable mode, from the top level directory: pip install -e .)

If you want to regenerate a single image (e.g. the elbow curve plot), you can do so as follows:

```
$ python docs/gallery.py elbow
```

If you want to regenerate them all (note: this takes a long time!)

```
$ python docs/gallery.py all
```

8.5.3 Advanced Development Topics

In this section we discuss more advanced contributing guidelines such as code conventions,the release life cycle or branch management. This section is intended for maintainers and core contributors of the Yellowbrick project. If you would like to be a maintainer please contact one of the current maintainers of the project.

Reviewing Pull Requests

We use several strategies when reviewing pull requests from contributors to Yellowbrick. If the pull request affects only a single file or a small portion of the code base, it is sometimes sufficient to review the code using GitHub's lightweight code review feature. However, if the changes impact a number of files or modify the documentation, our convention is to add the contributor's fork as a remote, pull, and check out their feature branch locally. From inside your fork of Yellowbrick, this can be done as follows:

```
$ git remote add contribsusername https://github.com/contribsusername/yellowbrick.git
$ git fetch contribsusername
$ git checkout -b contribsfeaturebranch contribsusername/contribsfeaturebranch
```

This will allow you to inspect their changes, run the tests, and build the docs locally. If the contributor has elected to allow reviewers to modify their feature branch, you will also be able to push changes directly to their branch:

```
$ git add filethatyouchanged.py
$ git commit -m "Adjusted tolerance levels to appease AppVeyor"
$ git push contribsusername contribsfeaturebranch
```

These changes will automatically go into the pull request, which can be useful for making small modifications (e.g. visual test tolerance levels) to get the PR over the finish line.

Visualizer Review Checklist

As the visualizer API has matured over time, we've realized that there are a number of routine items that must be in place to consider a visualizer truly complete and ready for prime time. This list is also extremely helpful for reviewing code submissions to ensure that visualizers are consistently implemented, tested, and documented. Though we do not expect these items to be checked off on every PR, the below list includes some guidance about what to look for when reviewing or writing a new Visualizer.

Note: The `contrib` module is a great place for work-in-progress Visualizers!

Code Conventions

- Ensure the visualizer API is met.

  The basic principle of the visualizer API is that scikit-learn methods such as `fit()`, `transform()`, `score()`, etc. perform interactions with scikit-learn or other computations and call the `draw()` method. Calls to matplotlib should happen only in `draw()` or `finalize()`.

- Create a quick method for the visualizer.

  In addition to creating the visualizer class, ensure there is an associated quick method that returns the visualizer and creates the visualization in one line of code!

- Subclass the correct visualizer.

  Ensure that the visualizer is correctly subclassed in the class hierarchy. If you're not sure what to subclass, please ping a maintainer, they'd be glad to help!

- Ensure numpy array comparisons are not ambiguous.

  Often there is code such as `if y:` where `y` is an array. However this is ambiguous when used with numpy arrays and other data containers. Change this code to `y is not None` or `len(y) > 0` or use `np.all` or `np.any` to test if the contents of the array are truthy/falsy.

- Add `random_state` argument to visualizer.

  If the visualizer uses/wraps a utility that also has `random_state`, then the visualizer itself needs to also have this argument which defaults to `None` and is passed to all internal stochastic behaviors. This ensures that image comparison testing will work and that users can get repeated behavior from visualizers.

- Use `np.unique` instead of *set*.

  If you need the unique values from a list or array, we prefer to use numpy methods wherever possible. We performed some limited benchmarking and believe that `np.unique` is a bit faster and more efficient.

- Use sklearn underscore suffix for learned parameters.

  Any parameters that are learned during `fit()` should only be added to the visualizer when `fit()` is called (this is also how we determine if a visualizer is fitted or not) and should be identified with an underscore suffix. For example, in classification visualizers, the classes can be either passed in by the user or determined when they are passed in via fit, therefore it should be `self.classes_`. This is also true for other learned parameters, e.g. `self.score_`, even though this is not created during `fit()`.

- Correctly set the title in finalize.

Use the `self.set_title()` method to set a default title; this allows the user to specify a custom title in the initialization arguments.

Testing Conventions

- Ensure there is an image comparison test.

  Ensure there is at least one image comparison test per visualizer. This is the primary regression testing of Yellowbrick and these tests catch a lot when changes occur in our dependencies or environment.

- Use pytest assertions rather than `unittest.TestCase` methods.

  We prefer `assert 2+2 == 4` rather than `self.assertEquals(2+2, 4)`. As a result, test classes should not extend `unittest.Testcase` but should extend the `VisualTestCase` in the tests package. Note that if you're writing tests that do not generate matplotlib figures you can simply extend `object`.

- Use test fixtures and sklearn dataset generators.

  Data is the key to testing with Yellowbrick; often the test package will have fixtures in `conftest.py` that can be directly used (e.g. binary vs. multiclass in the `test_classifier` package). If one isn't available feel free to use randomly generated datasets from the `sklearn.datasets` module e.g. `make_classification`, `make_regression`, or `make_blobs`. For integration testing, please feel free to use one of the Yellowbrick datasets.

- Fix all `random_state` arguments.

  Be on the lookout for any method (particularly sklearn methods) that have a `random_state` argument and be sure to fix them so that tests always pass!

- Test a variety of inputs.

  Machine learning can be done on a variety of inputs for `X` and `y`, ensure there is a test with numpy arrays, pandas DataFrame and Series objects, and with Python lists.

- Test that `fit()` returns self.

  When doing end-to-end testing, we like to `assert oz.fit() is oz` to ensure the API is maintained.

- Test that `score()` between zero and one.

  With visualizers that have a `score()` method, we like to `assert 0.0 <= oz.score() >=1.0` to ensure the API is maintained.

Documentation Conventions

- Visualizer DocString is correct.

  The visualizer docstring should be present under the class and contain a narrative about the visualizer and its arguments with the numpydoc style.

- API Documentation.

  All visualizers should have their own API page under `docs/api/[yb-module]`. This documentation should include an `automodule` statement. Generally speaking there is also an image generation script of the same name in this folder so that the documentation images can be generated on demand.

- Listing the visualizer.

  The visualizer should be listed in a number of places including: `docs/api/[yb-module]/index.rst`, `docs/api/index.rst`, `docs/index.rst`, `README.md`, and `DESCRIPTION.rst`.

- Include a gallery image.

  Please also add the visualizer image to the gallery!

- Update added to the changelog.

  To reduce the time it takes to put together the changelog, we'd like to update it when we add new features and visualizers rather than right before the release.

Merging Pull Requests

Our convention is that the person who performs the code review should merge the pull request (since reviewing is hard work and deserves due credit!). Only core contributors have write access to the repository and can merge pull requests. Some preferences for commit messages when merging in pull requests:

- Make sure to use the "Squash and Merge" option in order to create a Git history that is understandable.
- Keep the title of the commit short and descriptive; be sure it includes the PR #.
- Craft a commit message body that is 1-3 sentences, depending on the complexity of the commit; it should explicitly reference any issues being closed or opened using GitHub's commit message keywords.

Note: When merging a pull request, use the "squash and merge" option.

Releases

To ensure we get new code to our users as soon and as bug free as possible we periodically create major, minor, and hotfix version releases that are merged from the `develop` branch into `master` and pushed to PyPI and Anaconda Cloud. Our release cycle ensures that stable code can be found in the master branch and pip installed and that we can test our development code thoroughly before a release.

Note: The following steps must be taken by a maintainer with access to the primary (upstream) Yellowbrick repository. Any reference to `origin` refers to github.com/DistrictDataLabs/yellowbrick.

The first step is to create a release branch from develop - this allows us to do "release-work" (e.g. a version bump, changelog stuff, etc.) in a branch that is neither `develop` nor `master` and to test the release before deployment:

```
$ git checkout develop
$ git pull origin develop
$ git checkout -b release-x.x
```

This creates a release branch for version `x.x` where `x` is a digit. Release versions are described as a number `x.y.z` where `x` is the major version, `y` is the minor version and `z` is a patch version. Generally speaking most releases are minor version releases where `x.y` becomes `x.y+1`. Patch versions are infrequent but may also be needed where very little has changed or something quick has to be pushed to fix a critical bug, e.g.g `x.y` becomes `x.y.1`. Major version releases where `x.y` become `x+1.0` are rare.

At this point do the version bump by modifying `version.py` and the test version in `tests/__init__.py`. Make sure all tests pass for the release and that the documentation is up to date. To build the docs see the *documentation notes*. There may be style changes or deployment options that have to be done at this phase in the release branch. At this phase you'll also modify the `changelog` with the features and changes in the release that have not already been marked.

Note: Before merging the release to master make sure that the release checklist has been completed!

Once the release is ready for prime-time, merge into master:

```
$ git checkout master
$ git merge --no-ff --no-edit release-x.x
$ git push origin master
```

Tag the release in GitHub:

```
$ git tag -a vx.x
$ git push origin vx.x
```

Now go to the release page to convert the tag into a release and add a Markdown version of the changelog notes for those that are accessing the release directly from GitHub.

Deploying to PyPI

Deploying the release to PyPI is fairly straight forward. Ensure that you have valid PyPI login credentials in `~/.pypirc` and use the Makefile to deploy as follows:

```
$ make build
$ make deploy
```

The build process should create `build` and `dist` directories containing the wheel and source packages as well as a `.egg-info` file for deployment. The deploy command registers the version in PyPI and uploads it with Twine.

Deploying to Anaconda Cloud

These instructions follow the tutorial "Building conda packages with conda skeleton". To deploy release to Anaconda Cloud you first need to have Miniconda or Anaconda installed along with `conda-build` and `anaconda-client` (which can be installed using `conda`). Make sure that you run the `anaconda login` command using the credentials that allow access to the Yellowbrick channel. If you have an old skeleton directory, make sure to save it with a different name (e.g. yellowbrick.old) before running the skeleton command:

```
$ conda skeleton pypi yellowbrick
```

This should install the latest version of yellowbrick from PyPI - make sure the version matches the expected version of the release! There are some edits that must be made to the `yellowbrick/meta.yaml` that is generated as follows:

```
about:
    home: http://scikit-yb.org/
    license_file: LICENSE.txt
    doc_url: https://www.scikit-yb.org/en/latest/
    dev_url: https://github.com/DistrictDataLabs/yellowbrick
```

In addition, you must remove the entire `test:` section of the yaml file and add the following to the `requirements:` under both `host:` and `run:`. See example meta.yaml for a detailed version. Note that the description field in the metadata is pulled from the `DESCRIPTION.rst` in the root of the Yellowbrick project. However, Anaconda Cloud requires a Markdown description - the easiest thing to do is to copy it from the existing description.

With the `meta.yaml` file setup you can now run the build command for the various Python distributes that Yellowbrick supports:

```
$ conda build --python 3.6 yellowbrick
$ conda build --python 3.7 yellowbrick
```

After this command completes you should have build files in `$MINICONDA_HOME/conda-bld/[OS]/yellowbrick-x.x-py3.x_0.tar.bz2`. You can now run conda convert for each of the Python versions using this directory as follows:

```
$ conda convert --platform all [path to build] -o $MINICONDA_HOME/conda-bld
```

At this point you should have builds for all the versions of Python and all platforms Yellowbrick supports. Unfortunately at this point you have to upload them all to Anaconda Cloud:

```
$ anaconda upload $MINICONDA_HOME/conda-bld/[OS]/yellowbrick-x.x-py3.x_0.tar.bz2
```

Once uploaded, the Anaconda Cloud page should reflect the latest version, you may have to edit the description to make sure it's in Markdown format.

Finalizing the Release

The last steps in the release process are to check to make sure the release completed successfully. Make sure that the PyPI page and the Anaconda Cloud Page are correctly updated to the latest version. Also ensure that ReadTheDocs has correctly built the "latest" documentation on scikit-yb.org.

Make sure that you can update the package on your local machine, either in a virtual environment that does not include yellowbrick or in a Python install that is not used for development (e.g. not in the yellowbrick project directory):

```
$ pip install -U yellowbrick
$ python -c "import yellowbrick; print(yellowbrick.__version__)"
```

After verifying that the version has been correctly updated you can clean up the project directory:

```
$ make clean
```

After this, it's time to merge the release into develop so that we can get started on the next version!

```
$ git checkout develop
$ git merge --no-ff --no-edit release-x.x
$ git branch -d release-x.x
$ git push origin develop
```

Make sure to celebrate the release with the other maintainers and to tweet to everyone to let them know it's time to update Yellowbrick!

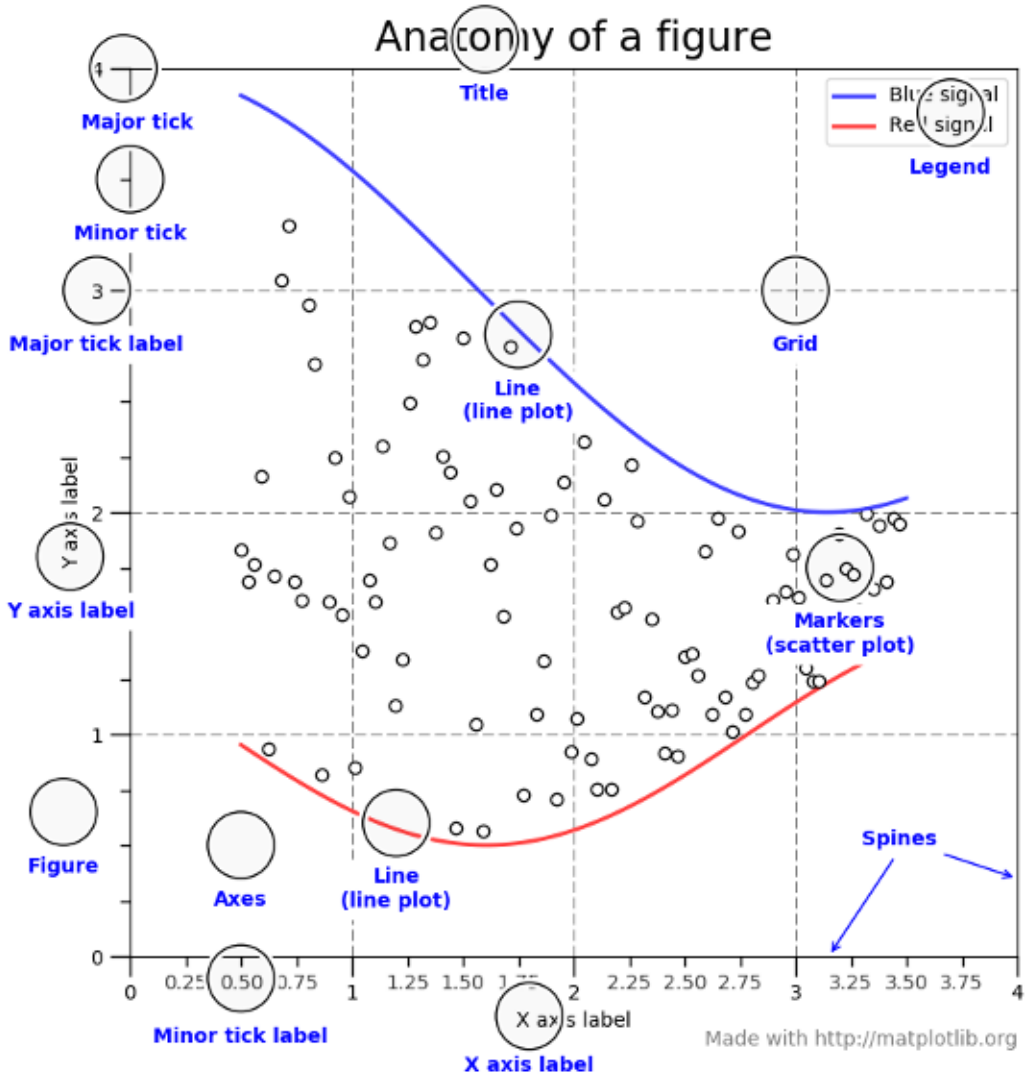
8.6 Effective Matplotlib

Yellowbrick generates visualizations by wrapping matplotlib, the most prominent Python scientific visualization library. Because of this, Yellowbrick is able to generate publication-ready images for a variety of GUI backends, image formats, and Jupyter notebooks. Yellowbrick strives to provide well-styled visual diagnostic tools and complete information. However, to customize figures or roll your own visualizers, a strong background in using matplotlib is required.

With permission, we have included part of Chris Moffitt's Effectively Using Matplotlib as a crash course into Matplotlib terminology and usage. For a complete example, please visit his excellent post on creating a visual sales analysis! This post was also adapated for matplotlib's documentation, The Lifecycle of a Plot, which gives another good perspective. Additionally we recommend Nicolas P. Rougier's Matplotlib tutorial for an in-depth dive.

8.6.1 Figures and Axes

This graphic from the matplotlib faq is gold. Keep it handy to understand the different terminology of a plot.



Most of the terms are straightforward but the main thing to remember is that the `Figure` is the final image that may contain 1 or more axes. The `Axes` represent an individual plot. Once you understand what these are and how to access them through the object oriented API, the rest of the process starts to fall into place.

The other benefit of this knowledge is that you have a starting point when you see things on the web. If you take the time to understand this point, the rest of the matplotlib API will start to make sense.

Matplotlib keeps a global reference to the global figure and axes objects which can be modified by the `pyplot` API. To access this import matplotlib as follows:

```
import matplotlib.pyplot as plt

axes = plt.gca()
```

The `plt.gca()` function gets the current axes so that you can draw on it directly. You can also directly create a figure and axes as follows:

```
fig = plt.figure()
ax = fig.add_subplot(111)
```

Yellowbrick will use `plt.gca()` by default to draw on. You can access the `Axes` object on a visualizer via its `ax` property:

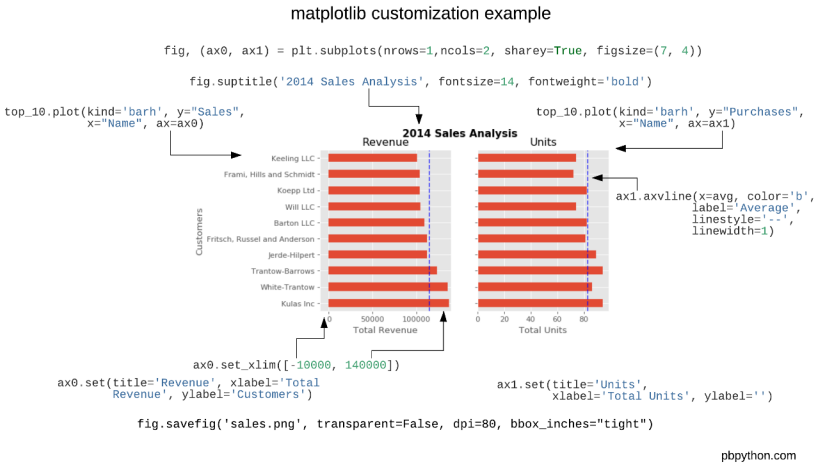
```
from sklearn.linear_model import LinearRegression
from yellowbrick.regressor import PredictionError

# Fit the visualizer
model = PredictionError(LinearRegression() )
model.fit(X_train, y_train)
model.score(X_test, y_test)

# Call finalize to draw the final yellowbrick-specific elements
model.finalize()

# Get access to the axes object and modify labels
model.ax.set_xlabel("measured concrete strength")
model.ax.set_ylabel("predicted concrete strength")
plt.savefig("peplot.pdf")
```

You can also pass an external `Axes` object directly to the visualizer:

```
model = PredictionError(LinearRegression(), ax=ax)
```

Therefore you have complete control of the style and customization of a Yellowbrick visualizer.

8.6.2 Creating a Custom Plot



The first step with any visualization is to plot the data. Often the simplest way to do this is using the standard pandas plotting function (given a `DataFrame` called `top_10`):

```
top_10.plot(kind='barh', y="Sales", x="Name")
```

The reason I recommend using pandas plotting first is that it is a quick and easy way to prototype your visualization. Since most people are probably already doing some level of data manipulation/analysis in pandas as a first step, go ahead and use the basic plots to get started.

Assuming you are comfortable with the gist of this plot, the next step is to customize it. Some of the customizations (like adding titles and labels) are very simple to use with the pandas plot function. However, you will probably find yourself needing to move outside of that functionality at some point. That's why it is recommended to create your own `Axes` first and pass it to the plotting function in Pandas:

```
fig, ax = plt.subplots()
top_10.plot(kind='barh', y="Sales", x="Name", ax=ax)
```

The resulting plot looks exactly the same as the original but we added an additional call to `plt.subplots()` and passed the `ax` to the plotting function. Why should you do this? Remember when I said it is critical to get access to the axes and figures in matplotlib? That's what we have accomplished here. Any future customization will be done via the `ax` or `fig` objects.

We have the benefit of a quick plot from pandas but access to all the power from matplotlib now. An example should show what we can do now. Also, by using this naming convention, it is fairly straightforward to adapt others' solutions to your unique needs.

Suppose we want to tweak the x limits and change some axis labels? Now that we have the axes in the `ax` variable, we have a lot of control:

```
fig, ax = plt.subplots()
top_10.plot(kind='barh', y="Sales", x="Name", ax=ax)
ax.set_xlim([-10000, 140000])
ax.set_xlabel('Total Revenue')
ax.set_ylabel('Customer');
```

Here's another shortcut we can use to change the title and both labels:

```
fig, ax = plt.subplots()
top_10.plot(kind='barh', y="Sales", x="Name", ax=ax)
ax.set_xlim([-10000, 140000])
ax.set(title='2014 Revenue', xlabel='Total Revenue', ylabel='Customer')
```

To further demonstrate this approach, we can also adjust the size of this image. By using the `plt.subplots()` function, we can define the `figsize` in inches. We can also remove the legend using `ax.legend().set_visible(False)`:

```
fig, ax = plt.subplots(figsize=(5, 6))
top_10.plot(kind='barh', y="Sales", x="Name", ax=ax)
ax.set_xlim([-10000, 140000])
ax.set(title='2014 Revenue', xlabel='Total Revenue')
ax.legend().set_visible(False)
```

There are plenty of things you probably want to do to clean up this plot. One of the biggest eye sores is the formatting of the Total Revenue numbers. Matplotlib can help us with this through the use of the `FuncFormatter` . This versatile function can apply a user defined function to a value and return a nicely formatted string to place on the axis.

Here is a currency formatting function to gracefully handle US dollars in the several hundred thousand dollar range:

```
def currency(x, pos):
    """
    The two args are the value and tick position
    """
    if x >= 1000000:
        return '${:1.1f}M'.format(x*1e-6)
    return '${:1.0f}K'.format(x*1e-3)
```

Now that we have a formatter function, we need to define it and apply it to the x axis. Here is the full code:

```
fig, ax = plt.subplots()
top_10.plot(kind='barh', y="Sales", x="Name", ax=ax)
ax.set_xlim([-10000, 140000])
ax.set(title='2014 Revenue', xlabel='Total Revenue', ylabel='Customer')
formatter = FuncFormatter(currency)
ax.xaxis.set_major_formatter(formatter)
ax.legend().set_visible(False)
```

That's much nicer and shows a good example of the flexibility to define your own solution to the problem.

The final customization feature I will go through is the ability to add annotations to the plot. In order to draw a vertical line, you can use `ax.axvline()` and to add custom text, you can use `ax.text()`.

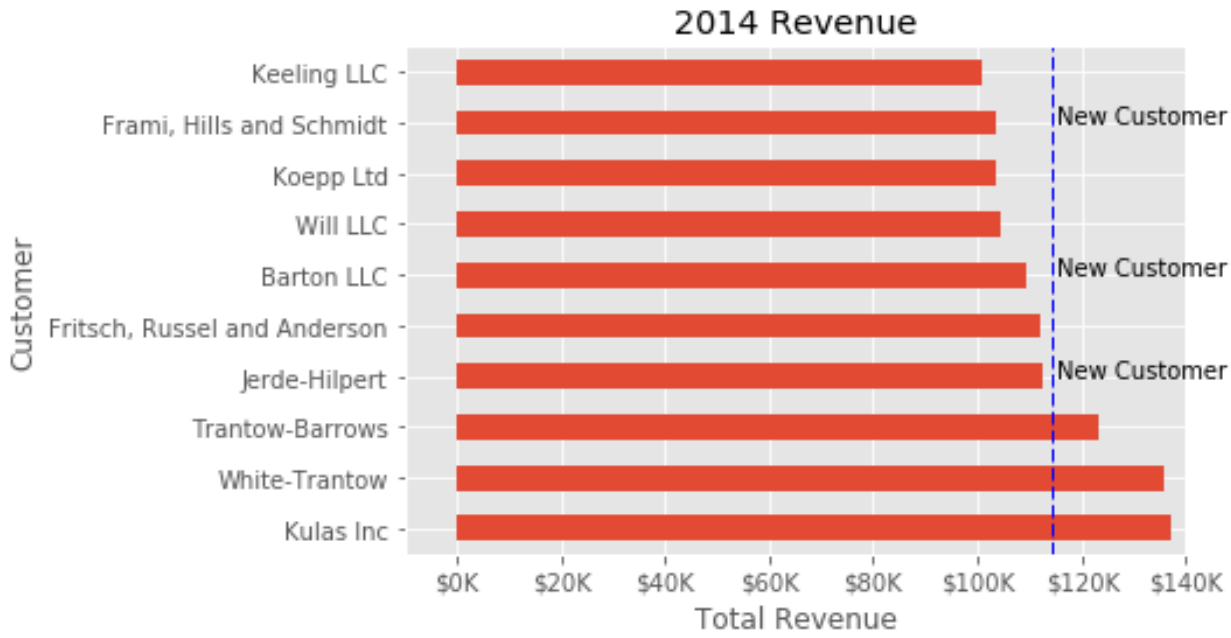
For this example, we'll draw a line showing an average and include labels showing three new customers. Here is the full code with comments to pull it all together.

```
# Create the figure and the axes
fig, ax = plt.subplots()

# Plot the data and get the average
top_10.plot(kind='barh', y="Sales", x="Name", ax=ax)
avg = top_10['Sales'].mean()

# Set limits and labels
ax.set_xlim([-10000, 140000])
```

(continues on next page)

(continued from previous page)

```
ax.set(title='2014 Revenue', xlabel='Total Revenue', ylabel='Customer')

# Add a line for the average
ax.axvline(x=avg, color='b', label='Average', linestyle='--', linewidth=1)

# Annotate the new customers
for cust in [3, 5, 8]:
ax.text(115000, cust, "New Customer")

# Format the currency
formatter = FuncFormatter(currency)
ax.xaxis.set_major_formatter(formatter)

# Hide the legend
ax.legend().set_visible(False)
```



While this may not be the most exciting plot it does show how much power you have when following this approach.

Up until now, all the changes we have made have been with the individual plot. Fortunately, we also have the ability to add multiple plots on a figure as well as save the entire figure using various options.

If we decided that we wanted to put two plots on the same figure, we should have a basic understanding of how to do it. First, create the figure, then the axes, then plot it all together. We can accomplish this using `plt.subplots()`:

```
fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, sharey=True, figsize=(7, 4))
```

In this example, I'm using `nrows` and `ncols` to specify the size because this is very clear to the new user. In sample code you will frequently just see variables like 1,2. I think using the named parameters is a little easier to interpret later on when you're looking at your code.

I am also using `sharey=True` so that the y-axis will share the same labels.

This example is also kind of nifty because the various axes get unpacked to `ax0` and `ax1`. Now that we have these axes, you can plot them like the examples above but put one plot on `ax0` and the other on `ax1`.

```
# Get the figure and the axes
fig, (ax0, ax1) = plt.subplots(nrows=1,ncols=2, sharey=True, figsize=(7, 4))
top_10.plot(kind='barh', y="Sales", x="Name", ax=ax0)
ax0.set_xlim([-10000, 140000])
ax0.set(title='Revenue', xlabel='Total Revenue', ylabel='Customers')

# Plot the average as a vertical line
avg = top_10['Sales'].mean()
ax0.axvline(x=avg, color='b', label='Average', linestyle='--', linewidth=1)

# Repeat for the unit plot
top_10.plot(kind='barh', y="Purchases", x="Name", ax=ax1)
avg = top_10['Purchases'].mean()
ax1.set(title='Units', xlabel='Total Units', ylabel='')
ax1.axvline(x=avg, color='b', label='Average', linestyle='--', linewidth=1)

# Title the figure
fig.suptitle('2014 Sales Analysis', fontsize=14, fontweight='bold');

# Hide the legends
ax1.legend().set_visible(False)
ax0.legend().set_visible(False)
```

When writing code in a Jupyter notebook you can take advantage of the `%matplotlib inline` or `%matplotlib notebook` directives to render figures inline. More often, however, you probably want to save your images to disk. Matplotlib supports many different formats for saving files. You can use `fig.canvas.get_supported_filetypes()` to see what your system supports:

```
fig.canvas.get_supported_filetypes()
```

```
{'eps': 'Encapsulated Postscript',
 'jpeg': 'Joint Photographic Experts Group',
 'jpg': 'Joint Photographic Experts Group',
 'pdf': 'Portable Document Format',
 'pgf': 'PGF code for LaTeX',
 'png': 'Portable Network Graphics',
 'ps': 'Postscript',
 'raw': 'Raw RGBA bitmap',
 'rgba': 'Raw RGBA bitmap',
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format'}
```

Since we have the fig object, we can save the figure using multiple options:

```
fig.savefig('sales.png', transparent=False, dpi=80, bbox_inches="tight")
```

This version saves the plot as a png with opaque background. I have also specified the dpi and bbox_inches=”tight” in order to minimize excess white space.

8.7 Yellowbrick for Teachers

For teachers and students of machine learning, Yellowbrick can be used as a framework for teaching and understanding a large variety of algorithms and methods. In fact, Yellowbrick grew out of teaching data science courses at Georgetown's School of Continuing Studies!

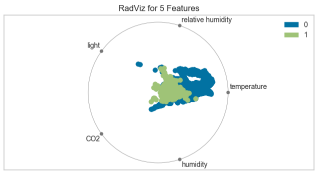
The following slide deck presents an approach to teaching students about the machine learning workflow (the model selection triple), including:

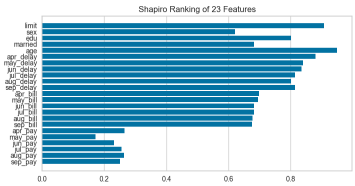
- feature analysis
- feature importances
- feature engineering
- algorithm selection
- model evaluation for classification and regression
- cross-validation
- hyperparameter tuning
- the scikit-learn API

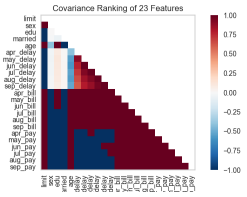
Teachers are welcome to download the slides via SlideShare as a PowerPoint deck, and to add them to their course materials to assist in teaching these important concepts.

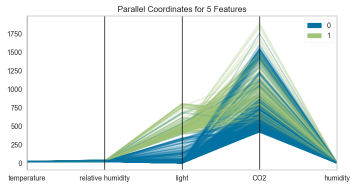
8.8 Gallery

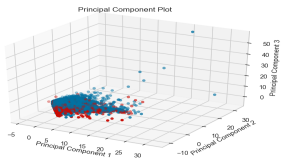
8.8.1 Feature Analysis

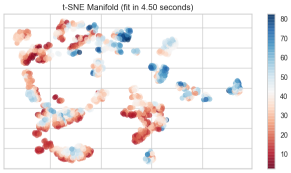


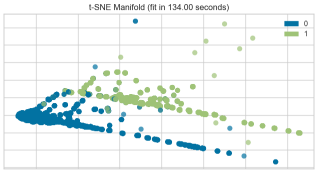


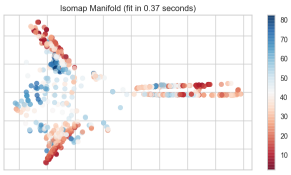


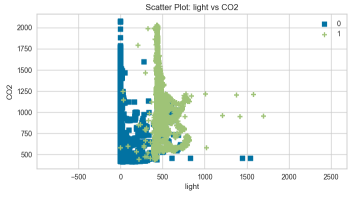

Parallel Coordinates for 5 Features
temperature
relative humidity
light
CO2
humidity

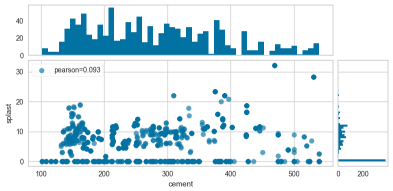

Principal Component Plot


t-SNE Manifold (fit in 4.50 seconds)

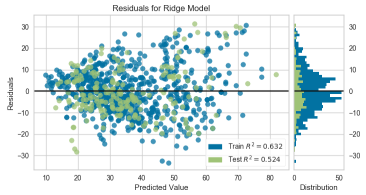

t-SNE Manifold (fit in 134.00 seconds)

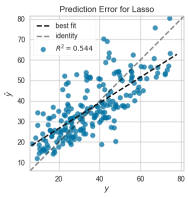

Isomap Manifold (fit in 0.37 seconds)

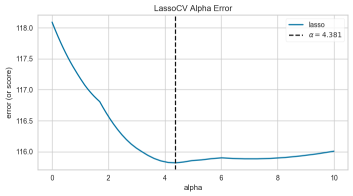

Scatter Plot: light vs CO2
light
CO2


pearson=0.093
cement
splast

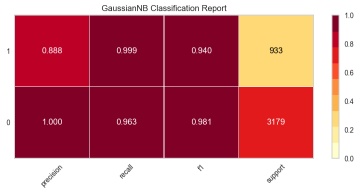
8.8.2 Regression Visualizers

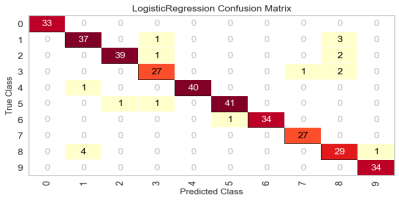


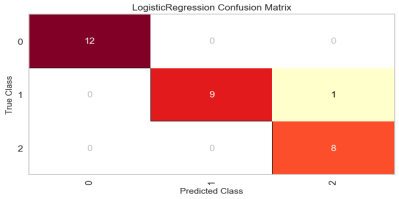


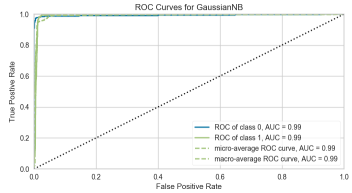


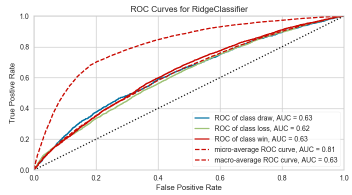
8.8.3 Classification Visualizers

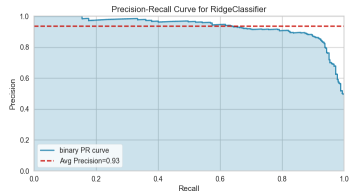


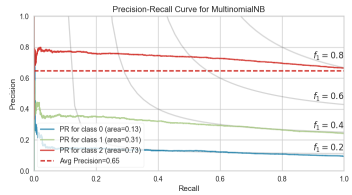


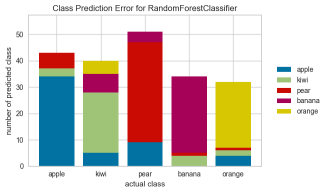


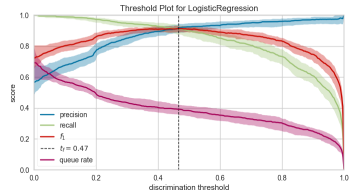




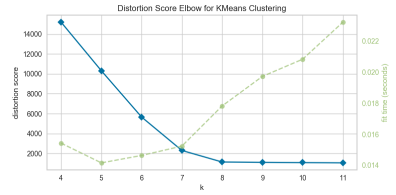


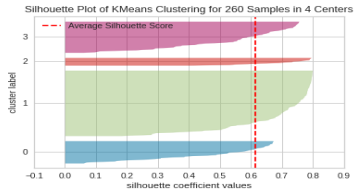


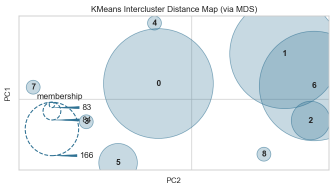




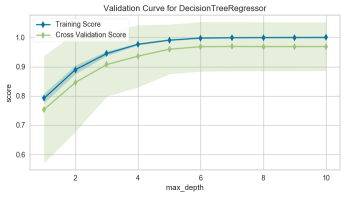
8.8.4 Clustering Visualizers

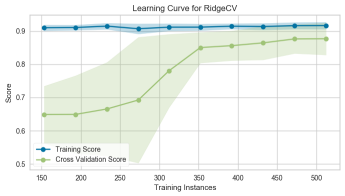


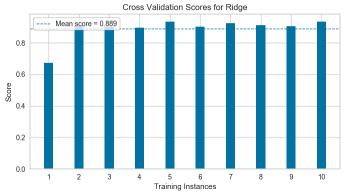


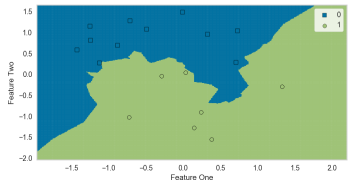


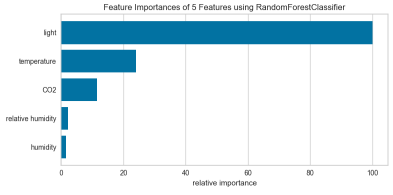
8.8.5 Model Selection Visualizers

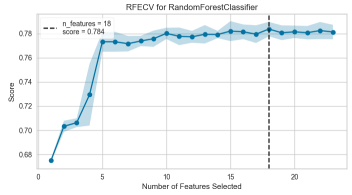




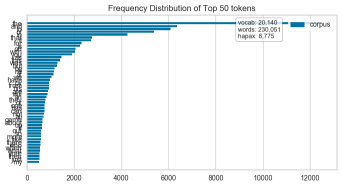


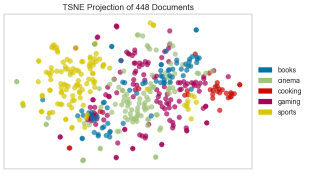


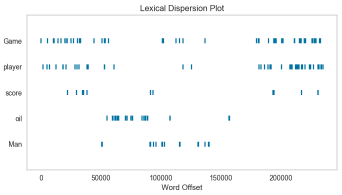


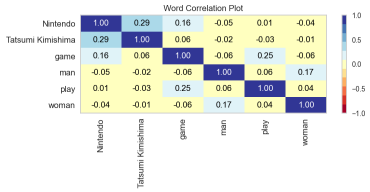


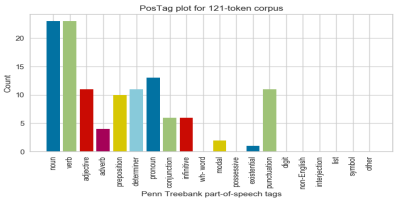
8.8.6 Text Modeling Visualizers



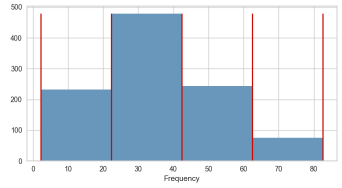


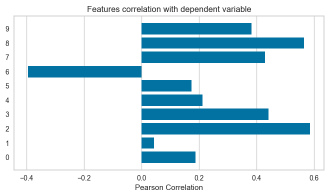






8.8.7 Target Visualizers





8.9 About

Image by QuatroCinco, used with permission, Flickr Creative Commons.

Yellowbrick is an open source, pure Python project that extends the scikit-learn API with visual analysis and diagnostic tools. The Yellowbrick API also wraps matplotlib to create publication-ready figures and interactive data explorations while still allowing developers fine-grain control of figures. For users, Yellowbrick can help evaluate the performance, stability, and predictive value of machine learning models and assist in diagnosing problems throughout the machine learning workflow.

Recently, much of this workflow has been automated through grid search methods, standardized APIs, and GUI-based applications. In practice, however, human intuition and guidance can more effectively hone in on quality models than exhaustive search. By visualizing the model selection process, data scientists can steer towards final, explainable models and avoid pitfalls and traps.

The Yellowbrick library is a diagnostic visualization platform for machine learning that allows data scientists to steer the model selection process. It extends the scikit-learn API with a new core object: the Visualizer. Visualizers allow visual

models to be fit and transformed as part of the scikit-learn pipeline process, providing visual diagnostics throughout the transformation of high-dimensional data.

8.9.1 Model Selection

Discussions of machine learning are frequently characterized by a singular focus on model selection. Be it logistic regression, random forests, Bayesian methods, or artificial neural networks, machine learning practitioners are often quick to express their preference. The reason for this is mostly historical. Though modern third-party machine learning libraries have made the deployment of multiple models appear nearly trivial, traditionally the application and tuning of even one of these algorithms required many years of study. As a result, machine learning practitioners tended to have strong preferences for particular (and likely more familiar) models over others.

However, model selection is a bit more nuanced than simply picking the "right" or "wrong" algorithm. In practice, the workflow includes:

1. selecting and/or engineering the smallest and most predictive feature set
2. choosing a set of algorithms from a model family
3. tuning the algorithm hyperparameters to optimize performance

The **model selection triple** was first described in a 2015 SIGMOD paper by Kumar et al. In their paper, which concerns the development of next-generation database systems built to anticipate predictive modeling, the authors cogently express that such systems are badly needed due to the highly experimental nature of machine learning in practice. "Model selection," they explain, "is iterative and exploratory because the space of [model selection triples] is usually infinite, and it is generally impossible for analysts to know a priori which [combination] will yield satisfactory accuracy and/or insights."

8.9.2 Who is Yellowbrick for?

Yellowbrick `Visualizers` have multiple use cases:

- For data scientists, they can help evaluate the stability and predictive value of machine learning models and improve the speed of the experimental workflow.
- For data engineers, Yellowbrick provides visual tools for monitoring model performance in real world applications.
- For users of models, Yellowbrick provides visual interpretation of the behavior of the model in high dimensional feature space.
- For teachers and students, Yellowbrick is a framework for teaching and understanding a large variety of algorithms and methods.

8.9.3 Name Origin

The Yellowbrick package gets its name from the fictional element in the 1900 children's novel **The Wonderful Wizard of Oz** by American author L. Frank Baum. In the book, the yellow brick road is the path that the protagonist, Dorothy Gale, must travel in order to reach her destination in the Emerald City.

From Wikipedia:

> "The road is first introduced in the third chapter of The Wonderful Wizard of Oz. The road begins in the heart of the eastern quadrant called Munchkin Country in the Land of Oz. It functions as a guideline that leads all who follow it, to the road's ultimate destination—the imperial capital of Oz called Emerald City that is located in the exact center of the entire continent. In the book, the novel's main protagonist, Dorothy, is forced to search for the road before she can begin her quest to seek the Wizard. This is because the cyclone from Kansas did not

> release her farmhouse closely near it as it did in the various film adaptations. After the council with the native Munchkins and their dear friend the Good Witch of the North, Dorothy begins looking for it and sees many pathways and roads nearby, (all of which lead in various directions). Thankfully it doesn't take her too long to spot the one paved with bright yellow bricks."

8.9.4 Team

Yellowbrick is developed by volunteer data scientists who believe in open source and the project enjoys contributions from Python developers all over the world. The project was started by @rebeccabilbro and @bbengfort as an attempt to better explain machine learning concepts to their students at Georgetown University where they teach a data science certificate program. They quickly realized, however, that the potential for visual steering could have a large impact on practical data science and developed it into a production-ready Python library.

Yellowbrick was then incubated by District Data Labs (DDL) in partnership with Georgetown University. District Data Labs is an organization that is dedicated to open source development and data science education and provided resources to help Yellowbrick grow. Yellowbrick was first introduced to the Python Community at PyCon 2016 in both talks and during the development sprints. The project was then carried on through DDL Research Labs – semester-long sprints where members of the DDL community contribute to various data-related projects.

Since then, Yellowbrick has enjoyed the participation of a large number of contributors from around the world and growing support in the PyData community. Yellowbrick has been featured in talks at PyData, Scipy, NumFOCUS, and PSF organized events as well as blog posts and Kaggle competitions. We are so thrilled to have such a dedicated community involved in active contributions both large and small.

For a full list of current maintainers and core contributors, please see MAINTAINERS.md in the root of our GitHub repository. Thank you so much to everyone who has contributed to Yellowbrick!

8.9.5 Affiliations

Yellowbrick is proud to be affiliated with several organizations that provide institutional support to the project. Such support is sometimes financial, often material, and always in the spirit of free and open source software. We can't thank them enough for their role in making Yellowbrick what it is today.

District Data Labs: District Data Labs incubated Yellowbrick and sponsors research labs by purchasing food and organizing events. Research labs are semester long sprints that allow Yellowbrick contributors to meet in person, share a meal, and hack on the project. DDL also sponsors travel to PyCon and PyData conferences for Yellowbrick maintainers and helps us buy promotional material such as stickers and t-shirts.

NumFOCUS: Yellowbrick is a NumFOCUS affiliated project (not a fiscally sponsored project). Our relationship with NumFOCUS has given us a lot of data science cred in the community by being listed on their website. We are also eligible to apply for small development grants and infrastructure support. We often participate in the project developers mailing list and other activities such as Google Summer of Code.

Georgetown University: Georgetown primarily provides space for Yellowbrick events including the research labs. Additionally, Georgetown Data Science Certificate students are introduced to Yellowbrick at the beginning of their machine learning education and we often perform user testing of new features on them!

How to Support Yellowbrick

Yellowbrick is developed by volunteers who work on the project in their spare time and not as part of their regular full-time work. If Yellowbrick has become critical to the success of your organization, please consider giving back to Yellowbrick.

> "... open source thrives on human rather than financial resources. There are many ways to grow human resources, such as distributing the workload among more contributors or encouraging companies to make open source part of their employees' work. An effective support strategy must include multiple ways to generate time and resources besides directly financing development. It must start from the principle that the open source approach is not inherently flawed, but rather under-resourced."
>
> —Roads and Bridges: The Unseen Labor Behind our Digital Infrastructure

The main thing that the Yellowbrick maintainers need is *time*. There are many ways to provide that time through non-financial mechanisms such as:

- Create a written policy in your company handbook that dedicates time for your employees to contribute to open source projects like Yellowbrick.
- Interact with our community giving encouragement and advice, particularly for long term planning and non-code related activities like design and documentation.
- Advocate and evangelize your use of Yellowbrick and other open source software through blog posts and social media.
- Consider long term support strategies rather than ad hoc or one-off actions.
- Teach your students Machine Learning with Yellowbrick.

More concrete and financial support is also welcome, particularly if it's directed through a specific effort. If you are interested in this kind of support consider:

- Making a donation to NumFOCUS on behalf of Yellowbrick.
- Engaging District Data Labs for coporate training on visual machine learning with Yellowbrick (which will directly support Yellowbrick maintainers).
- Supporting your employee's continuing professional education in the Georgetown Data Science Certificate.
- Providing long term support for fixed costs such as hosting.

Yellowbrick's mission is to enhance the machine learning workflow through open source visual steering and diagnostics. If you're interested in a more formal affiliate relationship to support this mission, please get in contact with us directly.

8.9.6 License

Yellowbrick is an open source project and its license is an implementation of the FOSS Apache 2.0 license by the Apache Software Foundation. In plain English this means that you can use Yellowbrick for commercial purposes, modify and distribute the source code, and even sublicense it. We want you to use Yellowbrick, profit from it, and contribute back if you do cool things with it.

There are, however, a couple of requirements that we ask from you. First, when you copy or distribute Yellowbrick source code, please include our copyright and license found in the LICENSE.txt at the root of our software repository. In addition, if we create a file called "NOTICE" in our project you must also include that in your source distribution. The "NOTICE" file will include attribution and thanks to those who have worked so hard on the project! Note that you may not use our names, trademarks, or logos to promote your work or in any other way than to reference Yellowbrick. Finally, we provide Yellowbrick with no warranty and you can't hold any Yellowbrick contributor or affiliate liable for your use of our software.

We think that's a pretty fair deal, and we're big believers in open source. If you make any changes to our software, use it commercially or academically, or have any other interest, we'd love to hear about it.

8.9.7 Presentations

Yellowbrick has enjoyed the spotlight in several presentations at recent conferences. We hope that these notebooks, talks, and slides will help you understand Yellowbrick a bit better.

Papers:

- Yellowbrick: Visualizing the Scikit-Learn Model Selection Process

Conference Presentations (videos):

- Visual Diagnostics for More Informed Machine Learning: Within and Beyond Scikit-Learn (PyCon 2016)
- Yellowbrick: Steering Machine Learning with Visual Transformers (PyData London 2017)

Jupyter Notebooks:

- Data Science Delivered: ML Regression Predications

Slides:

- Machine Learning Libraries You'd Wish You'd Known About (PyData Budapest 2017)
- Visualizing the Model Selection Process
- Visualizing Model Selection with Scikit-Yellowbrick
- Visual Pipelines for Text Analysis (Data Intelligence 2017)

8.9.8 Citing Yellowbrick

We hope that Yellowbrick facilitates machine learning of all kinds and we're particularly fond of academic work and research. If you're writing a scientific publication that uses Yellowbrick you can cite *Bengfort et al. (2018)* with the following BibTex:

```
@software{bengfort_yellowbrick_2018,
    title = {Yellowbrick},
    rights = {Apache License 2.0},
    url = {http://www.scikit-yb.org/en/latest/},
    abstract = {Yellowbrick is an open source, pure Python project that
        extends the Scikit-Learn {API} with visual analysis and
        diagnostic tools. The Yellowbrick {API} also wraps Matplotlib to
        create publication-ready figures and interactive data
        explorations while still allowing developers fine-grain control
        of figures. For users, Yellowbrick can help evaluate the
        performance, stability, and predictive value of machine learning
        models, and assist in diagnosing problems throughout the machine
        learning workflow.},
    version = {0.9.1},
    author = {Bengfort, Benjamin and Bilbro, Rebecca and Danielsen, Nathan and
        Gray, Larry and {McIntyre}, Kristen and Roman, Prema and Poh, Zijie and
        others},
```

(continues on next page)

(continued from previous page)

```
    date = {2018-11-14},
    year = {2018},
    doi = {10.5281/zenodo.1206264}
}
```

You can also find DOI (digital object identifiers) for every version of Yellowbrick on zenodo.org; use the BibTeX on this site to reference specific versions or changes made to the software.

We've also published a paper in the Journal of Open Source Software (JOSS) that discusses how Yellowbrick is designed to influence the model selection workflow. You may cite this paper if you are discussing Yellowbrick more generally in your research (instead of a specific version) or are interested in discussing visual analytics or visualization for machine learning. Please cite *Bengfort and Bilbro (2019)* with the following BibTex:

```
@article{bengfort_yellowbrick_2019,
    title = {Yellowbrick: {{Visualizing}} the {{Scikit}}-{{Learn Model Selection Process}
↪}},
    journaltitle = {The Journal of Open Source Software},
    volume = {4},
    number = {35},
    series = {1075},
    date = {2019-03-24},
    year = {2019},
    author = {Bengfort, Benjamin and Bilbro, Rebecca},
    url = {http://joss.theoj.org/papers/10.21105/joss.01075},
    doi = {10.21105/joss.01075}
}
```

8.9.9 Contacting Us

The best way to contact the Yellowbrick team is to send us a note on one of the following platforms:

- Send an email via our mailing list.
- Direct message us on Twitter.
- Ask a question on Stack Overflow.
- Report an issue on our GitHub Repo.

8.10 Frequently Asked Questions

Welcome to our frequently asked questions page. We're glad that you're using Yellowbrick! If your question is not captured here, please submit it to our Google Groups Listserv. This is an email list/forum that you, as a Yellowbrick user, can join and interact with other users to address and troubleshoot Yellowbrick issues. The Google Groups Listserv is where you should be able to receive the quickest response. We would welcome and encourage you to join the group so that you can respond to others' questions! You can also ask questions on Stack Overflow and tag them with "yellowbrick". Finally, you can add issues on GitHub and you can tweet or direct message us on Twitter @scikit_yb.

8.10.1 How can I change the size of a Yellowbrick plot?

You can change the `size` of a plot by passing in the desired dimensions in pixels on instantiation of the visualizer:

```
# Import the visualizer
from yellowbrick.features import RadViz

# Instantiate the visualizer using the ``size`` param
visualizer = RadViz(
    classes=classes, features=features, size=(1080, 720)
)

...
```

Note: we are considering adding support for passing in `size` in inches in a future Yellowbrick release. For a convenient inch-to-pixel converter, check out www.unitconversion.org.

8.10.2 How can I change the title of a Yellowbrick plot?

You can change the `title` of a plot by passing in the desired title as a string on instantiation:

```
from yellowbrick.classifier import ROCAUC
from sklearn.linear_model import RidgeClassifier

# Create your custom title
my_title = "ROCAUC Curves for Multiclass RidgeClassifier"

# Instantiate the visualizer passing the custom title
visualizer = ROCAUC(
    RidgeClassifier(), classes=classes, title=my_title
)

...
```

8.10.3 How can I change the color of a Yellowbrick plot?

Yellowbrick uses colors to make visualzers as interpretable as possible for intuitive machine learning diagnostics. Generally, color is specified by the target variable, `y` that you might pass to an estimator's fit method. Therefore Yellowbrick considers color based on the datatype of the target:

- **Discrete**: when the target is represented by discrete classes, Yellowbrick uses categorical colors that are easy to discriminate from each other.
- **Continuous**: when the target is represented by continous values, Yellowbrick uses a sequential colormap to show the range of data.

Most visualizers therefore accept the `colors` and `colormap` arguments when they are initialized. Generally speaking, if the target is discrete, specify *colors* as a list of valid matplotlib colors; otherwise if your target is continuous, specify a matplotlib colormap or colormap name. Most Yellowbrick visualizers are smart enough to figure out the colors for each of your data points based on what you pass in; for example if you pass in a colormap for a discrete target, the visualizer will create a list of discrete colors from the sequential colors.

Note: Although most visualizers support these arguments, please be sure to check the visualizer as it may have specific color requirements. E.g. the *ResidualsPlot* accepts the `train_color`, `test_color`, and `line_color` to modify its visualization. To see a visualizer's arguments you can use `help(Visualizer)` or `visualizer.get_params()`.

For example, the *Manifold* can visualize both discrete and sequential targets. In the discrete case, pass a list of valid color values as follows:

```
from yellowbrick.features.manifold import Manifold

visualizer = Manifold(
    manifold="tsne", target="discrete", colors=["teal", "orchid"]
)

...
```

... whereas for `continuous` targets, it is better to specify a matplotlib colormap:

```
from yellowbrick.features.manifold import Manifold

visualizer = Manifold(
    manifold="isomap", target="continuous", colormap="YlOrRd"
)

...
```

Finally please note that you can manipulate the default colors that Yellowbrick uses by modifying the matplotlib styles, particularly the default color cycle. Yellowbrick also has some tools for style management, please see *Colors and Style* for more information.

8.10.4 How can I save a Yellowbrick plot?

Save your Yellowbrick plot by passing an `outpath` into `show()`:

```
from sklearn.cluster import MiniBatchKMeans
from yellowbrick.cluster import KElbowVisualizer

visualizer = KElbowVisualizer(MiniBatchKMeans(), k=(4,12))

visualizer.fit(X)
visualizer.show(outpath="kelbow_minibatchkmeans.png")

...
```

Most backends support png, pdf, ps, eps and svg to save your work!

8.10.5 How can I make overlapping points show up better?

You can use the `alpha` param to change the opacity of plotted points (where `alpha=1` is complete opacity, and `alpha=0` is complete transparency):

```
from yellowbrick.contrib.scatter import ScatterVisualizer

visualizer = ScatterVisualizer(
    x="light", y="C02", classes=classes, alpha=0.5
)
```

8.10.6 How can I access the sample datasets used in the examples?

Visit the *Example Datasets* page.

8.10.7 Can I use Yellowbrick with libraries other than scikit-learn?

Potentially! Yellowbrick visualizers rely on the internal model implementing the scikit-learn API (e.g. having a `fit()` and `predict()` method), and often expect to be able to retrieve learned attributes from the model (e.g. `coef_`). Some third-party estimators fully implement the scikit-learn API, but not all do.

When using third-party libraries with Yellowbrick, we encourage you to `wrap` the model using the `yellowbrick.contrib.wrapper` module. Visit the *Using Third-Party Estimators* page for all the details!

8.11 User Testing Instructions

We are looking for people to help us Alpha test the Yellowbrick project! Helping is simple: simply create a notebook that applies the concepts in this *Getting Started* guide to a small-to-medium size dataset of your choice. Run through the examples with the dataset, and try to change options and customize as much as possible. After you've exercised the code with your examples, respond to our alpha testing survey!

8.11.1 Step One: Questionnaire

Please open the questionnaire, in order to familiarize yourself with the type of feedback we are looking to receive. We are very interested in identifying any bugs in Yellowbrick. Please include any cells in your Jupyter notebook that produce errors so that we may reproduce the problem.

8.11.2 Step Two: Dataset

Select a multivariate dataset of your own. The greater the variety of datasets that we can run through Yellowbrick, the more likely we'll discover edge cases and exceptions! Please note that your dataset must be well-suited to modeling with scikit-learn. In particular, we recommend choosing a dataset whose target is suited to one of the following supervised learning tasks:

- Regression (target is a continuous variable)
- Classification (target is a discrete variable)

There are datasets that are well suited to both types of analysis; either way, you can use the testing methodology from this notebook for either type of task (or both). In order to find a dataset, we recommend you try the following places:

- UCI Machine Learning Repository
- MLData.org
- Awesome Public Datasets

You're more than welcome to choose a dataset of your own, but we do ask that you make at least the notebook containing your testing results publicly available for us to review. If the data is also public (or you're willing to share it with the primary contributors) that will help us figure out bugs and required features much more easily!

8.11.3 Step Three: Notebook

Create a notebook in a GitHub repository. We suggest the following:

1. Fork the Yellowbrick repository
2. Under the `examples` directory, create a directory named with your GitHub username
3. Create a notebook named `testing`, i.e. examples/USERNAME/testing.ipynb

Alternatively, you could just send us a notebook via Gist or your own repository. However, if you fork Yellowbrick, you can initiate a pull request to have your example added to our gallery!

8.11.4 Step Four: Model with Yellowbrick and Scikit-Learn

Add the following to the notebook:

- A title in markdown
- A description of the dataset and where it was obtained
- A section that loads the data into a Pandas dataframe or NumPy matrix

Then conduct the following modeling activities:

- Feature analysis using scikit-learn and Yellowbrick
- Estimator fitting using scikit-learn and Yellowbrick

You can follow along with our `examples` directory (check out examples.ipynb) or even create your own custom visualizers! The goal is that you create an end-to-end model from data loading to estimator(s) with visualizers along the way.

IMPORTANT: please make sure you record all errors that you get and any tracebacks you receive for step three!

8.11.5 Step Five: Feedback

Finally, submit feedback via the Google Form we have created:

https://goo.gl/forms/naoPUMFa1xNcafY83

This form is allowing us to aggregate multiple submissions and bugs so that we can coordinate the creation and management of issues. If you are the first to report a bug or feature request, we will make sure you're notified (we'll tag you using your Github username) about the created issue!

8.11.6 Step Six: Thanks!

Thank you for helping us make Yellowbrick better! We'd love to see pull requests for features you think should be added to the library. We'll also be doing a user study that we would love for you to participate in. Stay tuned for more great things from Yellowbrick!

8.12 Code of Conduct

The Yellowbrick project is an open source, Python affiliated project. As a result, all interactions that occur with Yellowbrick must meet the guidelines described by the Python Software Foundation Code of Conduct. This includes interactions on all websites, tools, and resources used by Yellowbrick members including (but not limited to) mailing lists, issue trackers, GitHub, StackOverflow, etc.

In general this means everyone is expected to be open, considerate, and respectful of others no matter what their position is within the project.

Beyond this code of conduct, Yellowbrick is striving to set a very particular tone for contributors to the project. We show gratitiude for any contribution, no matter how small. We don't only point out constructive criticism, we always identify positive feedback. When we communicate via text, we write as though we are speaking to each other and our mothers are in the room with us. Our goal is to make Yellowbrick the best possible place to do your first open source contribution, no matter who you are.

8.13 Changelog

8.13.1 Version 1.5

- Tag: v1.5
- Deployed Sunday, August 21, 2022
- Current Contributors: Stefanie Molin, Prema Roman, Sangam Swadik, David Gilbertson, Larry Gray, Benjamin Bengfort, @admo1, @charlesincharge, Uri Nussbaum, Patrick Deziel, Rebecca Bilbro

Major

- Added `WordCorrelationPlot` Visualizer
- Built tests for using sklearn pipeline with visualizers
- Allowed Marker Style to be specified in Validation Curve Visualizer
- Fixed `get_params` for estimator wrapper to prevent `AttributeError`
- Updated missing values visualizer to handle multiple data types and work on both numpy arrays and pandas data frames.
- Added pairwise distance metrics to scoring metrics in KElbowVisualizer

Minor

- Pegged Numba to v0.55.2
- Updated Umap to v0.5.3
- Fixed Missing labels in classification report visualizer
- Updated Numpy to v1.22.0

Documentation

- The Spanish language Yellowbrick docs are now live: https://www.scikit-yb.org/es/latest/
- Added Dropping curve documentation
- Added new example Notebook for Regression Visualizers
- Fixed Typo in PR section of getting started docs
- Fixed Typo in rank docs
- Updated docstring in kneed.py utility file
- Clarified how to run 'make html' in PR template

Infrastructure

- Added ability to run linting Actions on PRs
- Implemented black code formatting as pre-commit hook

8.13.2 Version 1.4

- Tag: v1.4
- Deployed Saturday, February 19, 2022
- Current Contributors: Benjamin Bengfort, Larry Gray, Rebecca Bilbro, @pkaf, Antonio Carlos Falcão Petri, Aarni Koskela, Prema Roman, Nathan Danielsen, Eleni Markou, Patrick Deziel, Adam Morris, Hung-Tien Huang, @charlesincharge

Major

- Upgrade dependencies to support sklearn v1.0, Numpy 1.20+, Scipy 1.6, nltk 3.6.7, and Matplotlib 3.4.1
- Implement new `set_params` and `get_params` on ModelVisualizers to ensure wrapped estimator is being correctly accessed via the new `Estimator` methods.
- Fix the test dependencies to prevent variability in CI (must periodically review dependencies to ensure we're testing what our users are experiencing).
- Change `model` param to `estimator` param to ensure that Visualizer arguments match their property names so that inspect works with get and set params and other scikit-learn utility functions.

Minor

- Improved argmax handling in `DiscriminationThreshold` Visualizer
- Improved error handling in `FeatureImportances` Visualizer
- Gave option to remove colorer from `ClassificationReport` Visualizer
- Allowed for more flexible `KElbow` colors that use default palette by default
- Import scikit-learn private API _safe_indexing without error.
- Remove any calls to `set_params` in Visualizer `__init__` methods.
- Modify test fixtures and baseline images to accommodate new sklearn implementation
- Temporarily set the numpy dependency to be less than 1.20 because this is causing Pickle issues with joblib and umap
- Add `shuffle=True` argument to any CV class that uses a random seed.
- Set our CI matrix to Python and Miniconda 3.7 and 3.8

Bugs

- Fixed score label display in `PredictionError` Visualizer
- Fixed axes limit in `PredictionError` Visualizer
- Fixed `KElbowVisualizer` to handle null cluster encounters
- Fixed broken url to pytest fixtures
- Fixed `random_state` to be in sync with `PCA` transformer
- Fixed the inability to place `FeatureCorrelations` into subplots
- Fixed hanging printing impacting model visualizers
- Fixed error handling when decision function models encounter binary data
- Fixed missing code in README.md

Infrastructure/Housekeeping/documentation

- Updated status badges for build result and code coverage
- Removed deprecated pytest-runner from testing
- Replaced Travis with Github Actions
- Changed our master branch to the main branch
- Created a release issue template
- Updated our CI to test Python 3.8 and 3.9
- Managed test warnings
- Adds .gitattributes to fix handle white space changes
- Updated to use `add_css_file` for documentation because of deprecation of `add_stylesheet`
- Added a Sphinx build to GitHub Actions for ensuring that the docs build correctly
- Switched to a YB-specific data lake for datasets storage

8.13.3 Version 1.3.post1

- Tag: v1.3.post1
- Deployed Saturday, February 13, 2021
- Current Contributors: Rebecca Bilbro, Benjamin Bengfort, (EJ) Vivek Pandey

Fixes hanging print impacting ModelVisualizers.

8.13.4 Version 1.3

- Tag: v1.3
- Deployed Tuesday, February 9, 2021
- Current Contributors: Benjamin Bengfort, Rebecca Bilbro, Paul Johnson, Philippe Billet, Prema Roman, Patrick Deziel

This version primarily repairs the dependency issues we faced with scipy 1.6, scikit-learn 0.24 and Python 3.6 (or earlier). As part of the rapidly changing Python library landscape, we've been forced to react quickly to dependency changes, even where those libraries have been responsibly issuing future and deprecation warnings.

Major Changes:

- Implement new `set_params` and `get_params` on ModelVisualizers to ensure wrapped estimator is being correctly accessed via the new Estimator methods.
- Freeze the test dependencies to prevent variability in CI (must periodically review dependencies to ensure we're testing what our users are experiencing).
- Change `model` param to `estimator` param to ensure that Visualizer arguments match their property names so that inspect works with get and set params and other scikit-learn utility functions.

Minor Changes:

- Import scikit-learn private API `_safe_indexing` without error.
- Remove any calls to `set_params` in Visualizer `__init__` methods.
- Modify test fixtures and baseline images to accommodate new sklearn implementation
- Set the numpy dependency to be less than 1.20 because this is causing Pickle issues with joblib and umap
- Add `shuffle=True` argument to any CV class that uses a random seed.
- Set our CI matrix to Python and Miniconda 3.7 and 3.8
- Correction in README regarding ModelVisualizer API.

8.13.5 Hotfix 1.2.1

- Tag: v1.2.1
- Deployed Friday, January 15, 2020
- Contributors: Rebecca Bilbro, Benjamin Bengfort, Paul Johnson, Matt Harrison

On December 22, 2020, scikit-learn released version 0.24 which deprecated the external use of scikit-learn's internal utilities such as `safe_indexing`. Unfortunately, Yellowbrick depends on a few of these utilities and must refactor our internal code base to port this functionality or work around it. To ensure that Yellowbrick continues to work when installed via `pip`, we have temporarily changed our scikit-learn dependency to be less than 0.24. We will update our dependencies on the v1.3 release when we have made the associated fixes.

8.13.6 Version 1.2

- Tag: v1.2
- Deployed Friday, October 9, 2020
- Current Contributors: Rebecca Bilbro, Larry Gray, Vladislav Skripniuk, David Landsman, Prema Roman, @aldermartinez, Tan Tran, Benjamin Bengfort, Kellen Donohue, Kristen McIntyre, Tony Ojeda, Edwin Schmierer, Adam Morris, Nathan Danielsen

Major Changes:

- Added Q-Q plot as side-by-side option to the `ResidualsPlot` visualizer.
- More robust handling of binary classification in `ROCAUC` visualization, standardizing the way that classifiers with `predict_proba` and `decision_function` methods are handling. A `binary` hyperparameter was added to the visualizer to ensure correct interpretation of binary ROCAUC plots.
- Fixes to `ManualAlphaSelection` to move it from prototype to prime time including documentation, tests, and quick method. This method allows users to perform alpha selection visualization on non-CV estimators.

- Removal of AppVeyor from the CI matrix after too many out-of-core (non-Yellowbrick) failures with setup and installation on the VisualStudio images. Yellowbrick CI currently omits Windows and Miniconda from the test matrix and we are actively looking for new solutions.
- Third party estimator wrapper in contrib to provide enhanced support for non-scikit-learn estimators such as those in Keras, CatBoost, and cuML.

Minor Changes:

- Allow users to specify colors for the `PrecisionRecallCurve`.
- Update `ClassificationScoreVisualizer` base class to have a `class_colors_` learned attribute instead of a `colors` property; additional polishing of multi-class colors in `PrecisionRecallCurve`, `ROCAUC`, and `ClassPredictionError`.
- Update `KElbowVisualizer` fit method and quick method to allow passing `sample_weight` parameter through the visualizer.
- Enhancements to classification documentation to better discuss precision and recall and to diagnose with `PrecisionRecallCurve` and `ClassificationReport` visualizers.
- Improvements to `CooksDistance` visualizer documentation.
- Corrected `KElbowVisualizer` label and legend formatting.
- Typo fixes to `ROCAUC` documentation, labels, and legend. Typo fix to `Manifold` documentation.
- Use of `tight_layout` accessing the Visualizer figure property to finalize images and resolve discrepancies in plot directive images in documentation.
- Add `get_param_names` helper function to identify keyword-only parameters that belong to a specific method.
- Splits package namespace for `yellowbrick.regressor.residuals` to move `PredictionError` to its own module, `yellowbrick.regressor.prediction_error`.
- Update tests to use `SVC` instead of `LinearSVC` and correct `KMeans` scores based on updates to scikit-learn v0.23.
- Continued maintenance and management of baseline images following dependency updates; removal of mpl.cbook dependency.
- Explicitly include license file in source distribution via `MANIFEST.in`.
- Fixes to some deprecation warnings from `sklearn.metrics`.
- Testing requirements depends on Pandas v1.0.4 or later.
- Reintegrates pytest-spec and verbose test logging, updates pytest dependency to v0.5.0 or later.
- Added Pandas v0.20 or later to documentation dependencies.

8.13.7 Version 1.1

- Tag: v1.1
- Deployed Wednesday, February 12, 2020
- Contributors: Benjamin Bengfort, Rebecca Bilbro, Kristen McIntyre, Larry Gray, Prema Roman, Adam Morris, Shivendra Sharma, Michael Chestnut, Michael Garod, Naresh Bachwani, Piyush Gautam, Daniel Navarrete, Molly Morrison, Emma Kwiecinska, Sarthak Jain, Tony Ojeda, Edwin Schmierer, Nathan Danielsen

Major Changes:

- Quick methods (aka Oneliners), which return a fully fitted finalized visualizer object in only a single line, are now implemented for all Yellowbrick Visualizers. Test coverage has been added for all quick methods. The documentation has been updated to document and demonstrate the usage of the quick methods.
- Added Part of Speech tagging for raw text using spaCy and NLTK to POSTagVisualizer.

Minor Changes:

- Adds Board of Directors minutes for Spring meeting.
- Miscellaneous documentation corrections and fixes.
- Miscellaneous CI and testing corrections and fixes.

8.13.8 Hotfix 1.0.1

- Tag: v1.0.1
- Deployed Sunday, October 6, 2019
- Contributors: Benjamin Bengfort, Rebecca Bilbro, Kristen McIntyre

> **Warning:** **Major API change**: the `poof()` method is now deprecated, please use `show()` instead. After a significant discussion with community members we have deprecated our original "make the magic happen" method due to concerns about the usage of the word. We've renamed the original method to and created a stub method with the original name that issues a deprecation warning and calls `show()`.

Changes:

- Changes `poof()` to `show()`.
- Updated clustering and regression example notebooks.
- Fixes a syntax error in Python 3.5 and earlier.
- Updated Manifold documentation to fix example bug.
- Added advisors names to the release changelog.
- Adds advisory board minutes for Fall 2019.
- Updates our Travis-CI semi-secure token for Slack integration.

8.13.9 Version 1.0

- Tag: v1.0
- Deployed Wednesday, August 28, 2019
- Contributors: Benjamin Bengfort, Rebecca Bilbro, Nathan Danielsen, Kristen McIntyre, Larry Gray, Prema Roman, Adam Morris, Tony Ojeda, Edwin Schmierer, Carl Dawson, Daniel Navarrete, Francois Dion, Halee Mason, Jeff Hale, Jiayi Zhang, Jimmy Shah, John Healy, Justin Ormont, Kevin Arvai, Michael Garod, Mike Curry, Nabanita Dash, Naresh Bachwani, Nicholas A. Brown, Piyush Gautam, Pradeep Singh, Rohit Ganapathy, Ry Whittington, Sangarshanan, Sourav Singh, Thomas J Fan, Zijie (ZJ) Poh, Zonghan, Xie

> **Warning:** **Python 2 Deprecation**: Please note that this release deprecates Yellowbrick's support for Python 2.7. After careful consideration and following the lead of our primary dependencies (NumPy, scikit-learn, and Matplolib), we have chosen to move forward with the community and support Python 3.4 and later.

Major Changes:

- New `JointPlot` visualizer that is specifically designed for machine learning. The new visualizer can compare a feature to a target, features to features, and even feature to feature to target using color. The visualizer gives correlation information at a glance and is designed to work on ML datasets.
- New `PosTagVisualizer` is specifically designed for diagnostics around natural language processing and grammar-based feature extraction for machine learning. This new visualizer shows counts of different parts-of-speech throughout a tagged corpus.
- New datasets module that provide greater support for interacting with Yellowbrick example datasets including support for Pandas, npz, and text corpora.
- Management repository for Yellowbrick example data, `yellowbrick-datasets`.
- Add support for matplotlib 3.0.1 or greater.
- `UMAPVisualizer` as an alternative manifold to TSNE for corpus visualization that is fast enough to not require preprocessing PCA or SVD decomposition and preserves higher order similarities and distances.
- Added `..plot::` directives to the documentation to automatically build the images along with the docs and keep them as up to date as possible. The directives also include the source code making it much simpler to recreate examples.
- Added `target_color_type` functionality to determine continuous or discrete color representations based on the type of the target variable.
- Added alpha param for both test and train residual points in `ResidualsPlot`.
- Added `frameon` param to `Manifold`.
- Added frequency sort feature to `PosTagVisualizer`.
- Added elbow detection using the "kneedle" method to the `KElbowVisualizer`.
- Added governance document outlining new Yellowbrick structure.
- Added `CooksDistance` regression visualizer.
- Updated `DataVisualizer` to handle target type identification.
- Extended `DataVisualizer` and updated its subclasses.
- Added `ProjectionVisualizer` base class.
- Restructured `yellowbrick.target`, `yellowbrick.features`, and `yellowbrick.model_selection` API.
- Restructured regressor and classifier API.

Minor Changes:

- Updated `Rank2D` to include Kendall-Tau metric.
- Added user specification of ISO F1 values to `PrecisionRecallCurve` and updated the quick method to accept train and test splits.
- Added code review checklist and conventions to the documentation and expanded the contributing docs to include other tricks and tips.
- Added polish to missing value visualizers code, tests, and documentation.
- Improved `RankD` tests for better coverage.
- Added quick method test for `DispersionPlot` visualizer.

- BugFix: fixed resolve colors bug in TSNE and UMAP text visualizers and added regression tests to prevent future errors.
- BugFix: Added support for Yellowbrick palettes to return `colormap`.
- BugFix: fixed `PrecisionRecallCurve` visual display problem with multi-class labels.
- BugFix: fixed the `RFECV` step display bug.
- BugFix: fixed error in distortion score calculation.
- Extended `FeatureImportances` documentation and tests for stacked importances and added a warning when stack should be true.
- Improved the documentation readability and structure.
- Refreshed the `README.md` and added testing and documentation READMEs.
- Updated the gallery to generate thumbnail-quality images.
- Updated the example notebooks and created a quickstart notebook.
- Fixed broken links in the documentation.
- Enhanced the `SilhouetteVisualizer` with `legend` and `color` parameter, while also move labels to the y-axis.
- Extended `FeatureImportances` docs/tests for stacked importances.
- Documented the `yellowbrick.download` script.
- Added JOSS citation for "Yellowbrick: Visualizing the Scikit-Learn Model Selection Process".
- Added new pull request (PR) template.
- Added `alpha` param to PCA Decomposition Visualizer.
- Updated documentation with affiliations.
- Added a `windows_tol` for the visual unittest suite.
- Added stacked barchart to `PosTagVisualizer`.
- Let users set colors for `FreqDistVisualizer` and other `ax_bar` visualizers.
- Updated `Manifold` to extend `ProjectionVisualizer`.
- Check if an estimator is already fitted before calling `fit` method.
- Ensure `poof` returns `ax`.

Compatibility Notes:

- This version provides support for matplotlib 3.0.1 or greater and drops support for matplotlib versions less than 2.0.
- This version drops support for Python 2

8.13.10 Hotfix 0.9.1

This hotfix adds matplotlib3 support by requiring any version of matplotlib except for 3.0.0 which had a backend bug that affected Yellowbrick.

- Tag: v0.9.1
- Deployed: Tuesday, February 5, 2019
- Contributors: Benjamin Bengfort, Rebecca Bilbro, Ian Ozsvald, Francois Dion

8.13.11 Version 0.9

- Tag: v0.9
- Deployed: Wednesday, November 14, 2018
- Contributors: Rebecca Bilbro, Benjamin Bengfort, Zijie (ZJ) Poh, Kristen McIntyre, Nathan Danielsen, David Waterman, Larry Gray, Prema Roman, Juan Kehoe, Alyssa Batula, Peter Espinosa, Joanne Lin, @rlshuhart, @archaeocharlie, @dschoenleber, Tim Black, @iguk1987, Mohammed Fadhil, Jonathan Lacanlale, Andrew Godbehere, Sivasurya Santhanam, Gopal Krishna

Major Changes:

- Target module added for visualizing dependent variable in supervised models.
- Prototype missing values visualizer in contrib module.
- `BalancedBinningReference` visualizer for thresholding unbalanced data (undocumented).
- `CVScores` visualizer to instrument cross-validation.
- `FeatureCorrelation` visualizer to compare relationship between a single independent variable and the target.
- `ICDM` visualizer, intercluster distance mapping using projections similar to those used in pyLDAVis.
- `PrecisionRecallCurve` visualizer showing the relationship of precision and recall in a threshold-based classifier.
- Enhanced `FeatureImportance` for multi-target and multi-coefficient models (e.g probabilistic models) and allows stacked bar chart.
- Adds option to plot PDF to `ResidualsPlot` histogram.
- Adds document boundaries option to `DispersionPlot` and uses colored markers to depict class.
- Added alpha parameter for opacity to the scatter plot visualizer.
- Modify `KElbowVisualizer` to accept a list of k values.
- `ROCAUC` bugfix to allow binary classifiers that only have a decision function.
- `TSNE` bugfix so that title and size params are respected.
- `ConfusionMatrix` bugfix to correct percentage displays adding to 100.
- `ResidualsPlot` bugfix to ensure specified colors are both in histogram and scatterplot.
- Fixed unicode decode error on Py2 compatible Windows using Hobbies corpus.
- Require matplotlib 1.5.1 or matplotlib 2.0 (matplotlib 3.0 not supported yet).
- Deprecated percent and sample_weight arguments to `ConfusionMatrix` fit method.
- Yellowbrick now depends on SciPy 1.0 and scikit-learn 0.20.

Minor Changes:

- Removed hardcoding of `SilhouetteVisualizer` axes dimensions.
- Audit classifiers to ensure they conform to score API.
- Fix for `Manifold` `fit_transform` bug.
- Fixed `Manifold` import bug.
- Started reworking datasets API for easier loading of examples.
- Added `Timer` utility for keeping track of fit times.
- Added slides to documentation for teachers teaching ML/Yellowbrick.
- Added an FAQ to the documentation.
- Manual legend drawing utility.
- New examples notebooks for regression and clustering.
- Example of interactive classification visualization using ipywidgets.
- Example of using Yellowbrick with PyTorch.
- Repairs to `ROCAUC` tests and binary/multiclass `ROCAUC` construction.
- Rename tests/random.py to tests/rand.py to prevent NumPy errors.
- Improves `ROCAUC`, `KElbowVisualizer`, and `SilhouetteVisualizer` documentation.
- Fixed visual display bug in `JointPlotVisualizer`.
- Fixed image in `JointPlotVisualizer` documentation.
- Clear figure option to poof.
- Fix color plotting error in residuals plot quick method.
- Fixed bugs in `KElbowVisualizer`, `FeatureImportance`, Index, and Datasets documentation.
- Use LGTM for code quality analysis (replacing Landscape).
- Updated contributing docs for better PR workflow.
- Submitted JOSS paper.

8.13.12 Version 0.8

- Tag: v0.8
- Deployed: Thursday, July 12, 2018
- Contributors: Rebecca Bilbro, Benjamin Bengfort, Nathan Danielsen, Larry Gray, Prema Roman, Adam Morris, Kristen McIntyre, Raul Peralta, Sayali Sonawane, Alyssa Riley, Petr Mitev, Chris Stehlik, @thekylesaurus, Luis Carlos Mejia Garcia, Raul Samayoa, Carlo Mazzaferro

Major Changes:

- Added Support to `ClassificationReport` - @ariley1472
- We have an updated Image Gallery - @ralle123
- Improved performance of `ParallelCoordinates` Visualizer @ thekylesaurus
- Added Alpha Transparency to `RadViz` Visualizer @lumega
- `CVScores` Visualizer - @pdamodaran

- Added fast and alpha parameters to `ParallelCoordinates` visualizer @bbengfort
- Make support an optional parameter for `ClassificationReport` @lwgray
- Bug Fix for Usage of multidimensional arrays in `FeatureImportance` visualizer @rebeccabilbro
- Deprecate `ScatterVisualizer` to contrib @bbengfort
- Implements histogram alongside `ResidualsPlot` @bbengfort
- Adds biplot to the `PCADecomposition` visualizer @RaulPL
- Adds Datasaurus Dataset to show importance of visualizing data @lwgray
- Add `DispersionPlot` Plot @lwgray

Minor Changes:

- Fix grammar in tutorial.rst - @chrisfs
- Added Note to tutorial indicating subtle differences when working in Jupyter notebook - @chrisfs
- Update Issue template @bbengfort
- Added Test to check for NLTK postag data availability - @Sayali
- Clarify quick start documentation @mitevpi
- Deprecated `DecisionBoundary`
- Threshold Visualization aliases deprecated

8.13.13 Version 0.7

- Tag: v0.7
- Deployed: Thursday, May 17, 2018
- Contributors: Benjamin Bengfort, Nathan Danielsen, Rebecca Bilbro, Larry Gray, Ian Ozsvald, Jeremy Tuloup, Abhishek Bharani, Raúl Peralta Lozada, Tabishsada, Kristen McIntyre, Neal Humphrey

Changes:

- *New Feature!* Manifold visualizers implement high-dimensional visualization for non-linear structural feature analysis.
- *New Feature!* There is now a `model_selection` module with `LearningCurve` and `ValidationCurve` visualizers.
- *New Feature!* The `RFECV` (recursive feature elimination) visualizer with cross-validation visualizes how removing the least performing features improves the overall model.
- *New Feature!* The `VisualizerGrid` is an implementation of the `MultipleVisualizer` that creates axes for each visualizer using `plt.subplots`, laying the visualizers out as a grid.
- *New Feature!* Added `yellowbrick.datasets` to load example datasets.
- New Experimental Feature! An experimental `StatsModelsWrapper` was added to `yellowbrick.contrib.statsmodels` that will allow user to use StatsModels estimators with visualizers.
- *Enhancement!* `ClassificationReport` documentation to include more details about how to interpret each of the metrics and compare the reports against each other.
- *Enhancement!* Modifies scoring mechanism for regressor visualizers to include the R2 value in the plot itself with the legend.

- *Enhancement!* Updated and renamed the `ThreshViz` to be defined as `DiscriminationThreshold`, implements a few more discrimination features such as F1 score, maximizing arguments and annotations.
- *Enhancement!* Update clustering visualizers and corresponding `distortion_score` to handle sparse matrices.
- Added code of conduct to meet the GitHub community guidelines as part of our contributing documentation.
- Added `is_probabilistic` type checker and converted the type checking tests to pytest.
- Added a `contrib` module and `DecisionBoundaries` visualizer has been moved to it until further work is completed.
- Numerous fixes and improvements to documentation and tests. Add academic citation example and Zenodo DOI to the Readme.

Bug Fixes:

- Adds `RandomVisualizer` for testing and add it to the `VisualizerGrid` test cases.
- Fix / update tests in `tests.test_classifier.test_class_prediction_error.py` to remove hard-coded data.

Deprecation Warnings:

- `ScatterPlotVisualizer` is being moved to contrib in 0.8
- `DecisionBoundaryVisualizer` is being moved to contrib in 0.8
- `ThreshViz` is renamed to `DiscriminationThreshold`.

NOTE: These deprecation warnings originally mentioned deprecation in 0.7, but their life was extended by an additional version.

8.13.14 Version 0.6

- Tag: v0.6
- Deployed: Saturday, March 17, 2018
- Contributors: Benjamin Bengfort, Nathan Danielsen, Rebecca Bilbro, Larry Gray, Kristen McIntyre, George Richardson, Taylor Miller, Gary Mayfield, Phillip Schafer, Jason Keung

Changes:

- *New Feature!* The `FeatureImportances` Visualizer enables the user to visualize the most informative (relative and absolute) features in their model, plotting a bar graph of `feature_importances_` or `coef_` attributes.
- *New Feature!* The `ExplainedVariance` Visualizer produces a plot of the explained variance resulting from a dimensionality reduction to help identify the best tradeoff between number of dimensions and amount of information retained from the data.
- *New Feature!* The `GridSearchVisualizer` creates a color plot showing the best grid search scores across two parameters.
- *New Feature!* The `ClassPredictionError` Visualizer is a heatmap implementation of the class balance visualizer, which provides a way to quickly understand how successfully your classifier is predicting the correct classes.
- *New Feature!* The `ThresholdVisualizer` allows the user to visualize the bounds of precision, recall and queue rate at different thresholds for binary targets after a given number of trials.
- New `MultiFeatureVisualizer` helper class to provide base functionality for getting the names of features for use in plot annotation.

- Adds font size param to the confusion matrix to adjust its visibility.
- Add quick method for the confusion matrix
- Tests: In this version, we've switched from using nose to pytest. Image comparison tests have been added and the visual tests are updated to matplotlib 2.2.0. Test coverage has also been improved for a number of visualizers, including `JointPlot`, `AlphaPlot`, `FreqDist`, `RadViz`, `ElbowPlot`, `SilhouettePlot`, `ConfusionMatrix`, `Rank1D`, and `Rank2D`.
- Documentation updates, including discussion of Image Comparison Tests for contributors.

Bug Fixes:

- Fixes the `resolve_colors` function. You can now pass in a number of colors and a colormap and get back the correct number of colors.
- Fixes `TSNEVisualizer` Value Error when no classes are specified.
- Adds the circle back to `RadViz`! This visualizer has also been updated to ensure there's a visualization even when there are missing values
- Updated `RocAuc` to correctly check the number of classes
- Switch from converting structured arrays to ndarrays using `np.copy` instead of `np.tolist` to avoid NumPy deprecation warning.
- `DataVisualizer` updated to remove `np.nan` values and warn the user that nans are not plotted.
- `ClassificationReport` no longer has lines that run through the numbers, is more grid-like

Deprecation Warnings:

- `ScatterPlotVisualizer` is being moved to contrib in 0.7
- `DecisionBoundaryVisualizer` is being moved to contrib in 0.7

8.13.15 Version 0.5

- Tag: v0.5
- Deployed: Wednesday, August 9, 2017
- Contributors: Benjamin Bengfort, Rebecca Bilbro, Nathan Danielsen, Carlo Morales, Jim Stearns, Phillip Schafer, Jason Keung

Changes:

- Added `VisualTestCase`.
- New `PCADecomposition` Visualizer, which decomposes high-dimensional data into two or three dimensions so that each instance can be plotted in a scatter plot.
- New and improved `ROCAUC` Visualizer, which now supports multiclass classification.
- Prototype `DecisionBoundary` Visualizer, which is a bivariate data visualization algorithm that plots the decision boundaries of each class.
- Added `Rank1D` Visualizer, which is a one-dimensional ranking of features that utilizes the Shapiro-Wilks ranking by taking into account only a single feature at a time (e.g. histogram analysis).
- Improved `PredictionErrorPlot` with identity line, shared limits, and R-squared.
- Updated `FreqDist` Visualizer to make word features a hyperparameter.
- Added normalization and scaling to `ParallelCoordinates`.

- Added Learning Curve Visualizer, which displays a learning curve based on the number of samples versus the training and cross validation scores to show how a model learns and improves with experience.
- Added data downloader module to the Yellowbrick library.
- Complete overhaul of the Yellowbrick documentation; categories of methods are located in separate pages to make it easier to read and contribute to the documentation.
- Added a new color palette inspired by ANN-generated colors

Bug Fixes:

- Repairs to `PCA`, `RadViz`, `FreqDist` unit tests
- Repair to matplotlib version check in `JointPlotVisualizer`

8.13.16 Hotfix 0.4.2

Update to the deployment docs and package on both Anaconda and PyPI.

- Tag: v0.4.2
- Deployed: Monday, May 22, 2017
- Contributors: Benjamin Bengfort, Jason Keung

8.13.17 Version 0.4.1

This release is an intermediate version bump in anticipation of the PyCon 2017 sprints.

The primary goals of this version were to (1) update the Yellowbrick dependencies (2) enhance the Yellowbrick documentation to help orient new users and contributors, and (3) make several small additions and upgrades (e.g. pulling the Yellowbrick utils into a standalone module).

We have updated the scikit-learn and SciPy dependencies from version 0.17.1 or later to 0.18 or later. This primarily entails moving from `from sklearn.cross_validation import train_test_split` to `from sklearn.model_selection import train_test_split`.

The updates to the documentation include new Quickstart and Installation guides, as well as updates to the Contributors documentation, which is modeled on the scikit-learn contributing documentation.

This version also included upgrades to the KMeans visualizer, which now supports not only `silhouette_score` but also `distortion_score` and `calinski_harabaz_score`. The `distortion_score` computes the mean distortion of all samples as the sum of the squared distances between each observation and its closest centroid. This is the metric that KMeans attempts to minimize as it is fitting the model. The `calinski_harabaz_score` is defined as ratio between the within-cluster dispersion and the between-cluster dispersion.

Finally, this release includes a prototype of the `VisualPipeline`, which extends scikit-learn's `Pipeline` class, allowing multiple Visualizers to be chained or sequenced together.

- Tag: v0.4.1
- Deployed: Monday, May 22, 2017
- Contributors: Benjamin Bengfort, Rebecca Bilbro, Nathan Danielsen

Changes:

- Score and model visualizers now wrap estimators as proxies so that all methods on the estimator can be directly accessed from the visualizer
- Updated scikit-learn dependency from >=0.17.1 to >=0.18

- Replaced `sklearn.cross_validation` with `model_selection`
- Updated SciPy dependency from >=0.17.1 to >=0.18
- ScoreVisualizer now subclasses ModelVisualizer; towards allowing both fitted and unfitted models passed to Visualizers
- Added CI tests for Python 3.6 compatibility
- Added new quickstart guide and install instructions
- Updates to the contributors documentation
- Added `distortion_score` and `calinski_harabaz_score` computations and visualizations to KMeans visualizer.
- Replaced the `self.ax` property on all of the individual `draw` methods with a new property on the `Visualizer` class that ensures all visualizers automatically have axes.
- Refactored the utils module into a package
- Continuing to update the docstrings to conform to Sphinx
- Added a prototype visual pipeline class that extends the scikit-learn pipeline class to ensure that visualizers get called correctly.

Bug Fixes:

- Fixed title bug in Rank2D FeatureVisualizer

8.13.18 Version 0.4

This release is the culmination of the Spring 2017 DDL Research Labs that focused on developing Yellowbrick as a community effort guided by a sprint/agile workflow. We added several more visualizers, did a lot of user testing and bug fixes, updated the documentation, and generally discovered how best to make Yellowbrick a friendly project to contribute to.

Notable in this release is the inclusion of two new feature visualizers that use few, simple dimensions to visualize features against the target. The `JointPlotVisualizer` graphs a scatter plot of two dimensions in the data set and plots a best fit line across it. The `ScatterVisualizer` also uses two features, but also colors the graph by the target variable, adding a third dimension to the visualization.

This release also adds support for clustering visualizations, namely the elbow method for selecting K, `KElbowVisualizer` and a visualization of cluster size and density using the `SilhouetteVisualizer`. The release also adds support for regularization analysis using the `AlphaSelection` visualizer. Both the text and classification modules were also improved with the inclusion of the `PosTagVisualizer` and the `ConfusionMatrix` visualizer respectively.

This release also added an Anaconda repository and distribution so that users can `conda install` yellowbrick. Even more notable, we got Yellowbrick stickers! We've also updated the documentation to make it more friendly and a bit more visual; fixing the API rendering errors. All-in-all, this was a big release with a lot of contributions and we thank everyone that participated in the lab!

- Tag: v0.4
- Deployed: Thursday, May 4, 2017
- Contributors: Benjamin Bengfort, Rebecca Bilbro, Nathan Danielsen, Matt Andersen, Prema Roman, Neal Humphrey, Jason Keung, Bala Venkatesan, Paul Witt, Morgan Mendis, Tuuli Morril

Changes:

- Part of speech tags visualizer – `PosTagVisualizer`.

- Alpha selection visualizer for regularized regression – `AlphaSelection`
- Confusion Matrix Visualizer – `ConfusionMatrix`
- Elbow method for selecting K vis – `KElbowVisualizer`
- Silhouette score cluster visualization – `SilhouetteVisualizer`
- Joint plot visualizer with best fit – `JointPlotVisualizer`
- Scatter visualization of features – `ScatterVisualizer`
- Added three more example datasets: mushroom, game, and bike share
- Contributor's documentation and style guide
- Maintainers listing and contacts
- Light/Dark background color selection utility
- Structured array detection utility
- Updated classification report to use colormesh
- Added anacondas packaging and distribution
- Refactoring of the regression, cluster, and classification modules
- Image based testing methodology
- Docstrings updated to a uniform style and rendering
- Submission of several more user studies

8.13.19 Version 0.3.3

Intermediate sprint to demonstrate prototype implementations of text visualizers for NLP models. Primary contributions were the `FreqDistVisualizer` and the `TSNEVisualizer`.

The `TSNEVisualizer` displays a projection of a vectorized corpus in two dimensions using TSNE, a nonlinear dimensionality reduction method that is particularly well suited to embedding in two or three dimensions for visualization as a scatter plot. TSNE is widely used in text analysis to show clusters or groups of documents or utterances and their relative proximities.

The `FreqDistVisualizer` implements frequency distribution plot that tells us the frequency of each vocabulary item in the text. In general, it could count any kind of observable event. It is a distribution because it tells us how the total number of word tokens in the text are distributed across the vocabulary items.

- Tag: v0.3.3
- Deployed: Wednesday, February 22, 2017
- Contributors: Rebecca Bilbro, Benjamin Bengfort

Changes:

- `TSNEVisualizer` for 2D projections of vectorized documents
- `FreqDistVisualizer` for token frequency of text in a corpus
- Added the user testing evaluation to the documentation
- Created scikit-yb.org and host documentation there with RFD
- Created a sample corpus and text examples notebook
- Created a base class for text, `TextVisualizer`

- Model selection tutorial using Mushroom Dataset
- Created a text examples notebook but have not added to documentation.

8.13.20 Version 0.3.2

Hardened the Yellowbrick API to elevate the idea of a Visualizer to a first principle. This included reconciling shifts in the development of the preliminary versions to the new API, formalizing Visualizer methods like `draw()` and `finalize()`, and adding utilities that revolve around scikit-learn. To that end we also performed administrative tasks like refreshing the documentation and preparing the repository for more and varied open source contributions.

- Tag: v0.3.2
- Deployed: Friday, January 20, 2017
- Contributors: Benjamin Bengfort, Rebecca Bilbro

Changes:

- Converted Mkdocs documentation to Sphinx documentation
- Updated docstrings for all Visualizers and functions
- Created a DataVisualizer base class for dataset visualization
- Single call functions for simple visualizer interaction
- Added yellowbrick specific color sequences and palettes and env handling
- More robust examples with downloader from DDL host
- Better axes handling in visualizer, matplotlib/sklearn integration
- Added a finalize method to complete drawing before render
- Improved testing on real data sets from examples
- Bugfix: score visualizer renders in notebook but not in Python scripts.
- Bugfix: tests updated to support new API

8.13.21 Hotfix 0.3.1

Hotfix to solve pip install issues with Yellowbrick.

- Tag: v0.3.1
- Deployed: Monday, October 10, 2016
- Contributors: Benjamin Bengfort

  Changes:

  - Modified packaging and wheel for Python 2.7 and 3.5 compatibility
  - Modified deployment to PyPI and pip install ability
  - Fixed Travis-CI tests with the backend failures.

8.13.22 Version 0.3

This release marks a major change from the previous MVP releases as Yellowbrick moves towards direct integration with scikit-learn for visual diagnostics and steering of machine learning and could therefore be considered the first alpha release of the library. To that end we have created a Visualizer model which extends `sklearn.base.BaseEstimator` and can be used directly in the ML Pipeline. There are a number of visualizers that can be used throughout the model selection process, including for feature analysis, model selection, and hyperparameter tuning.

In this release specifically, we focused on visualizers in the data space for feature analysis and visualizers in the model space for scoring and evaluating models. Future releases will extend these base classes and add more functionality.

- Tag: v0.3
- Deployed: Sunday, October 9, 2016
- Contributors: Benjamin Bengfort, Rebecca Bilbro, Marius van Niekerk

  Enhancements:

  - Created an API for visualization with machine learning: Visualizers that are `BaseEstimators`.
  - Created a class hierarchy for Visualizers throughout the ML process particularly feature analysis and model evaluation
  - Visualizer interface is draw method which can be called multiple times on data or model spaces and a poof method to finalize the figure and display or save to disk.
  - `ScoreVisualizers` wrap scikit-learn estimators and implement `fit()` and `predict()` (pass-throughs to the estimator) and also score which calls draw in order to visually score the estimator. If the estimator isn't appropriate for the scoring method an exception is raised.
  - `ROCAUC` is a `ScoreVisualizer` that plots the receiver operating characteristic curve and displays the area under the curve score.
  - `ClassificationReport` is a `ScoreVisualizer` that renders the confusion matrix of a classifier as a heatmap.
  - `PredictionError` is a `ScoreVisualizer` that plots the actual vs. predicted values and the 45 degree accuracy line for regressors.
  - `ResidualPlot` is a `ScoreVisualizer` that plots the residuals (y - yhat) across the actual values (y) with the zero accuracy line for both train and test sets.
  - `ClassBalance` is a `ScoreVisualizer` that displays the support for each class as a bar plot.
  - `FeatureVisualizers` are scikit-learn Transformers that implement `fit()` and `transform()` and operate on the data space, calling draw to display instances.
  - `ParallelCoordinates` plots instances with class across each feature dimension as line segments across a horizontal space.
  - `RadViz` plots instances with class in a circular space where each feature dimension is an arc around the circumference and points are plotted relative to the weight of the feature.
  - `Rank2D` plots pairwise scores of features as a heatmap in the space [-1, 1] to show relative importance of features. Currently implemented ranking functions are Pearson correlation and covariance.
  - Coordinated and added palettes in the bgrmyck space and implemented a version of the Seaborn set_palette and set_color_codes functions as well as the `ColorPalette` object and other matplotlib.rc modifications.
  - Inherited Seaborn's notebook context and whitegrid axes style but make them the default, don't allow user to modify (if they'd like to, they'll have to import Seaborn). This gives Yellowbrick a consistent look and feel without giving too much work to the user and prepares us for matplotlib 2.0.

- Jupyter Notebook with Examples of all Visualizers and usage.

Bug Fixes:

- Fixed Travis-CI test failures with matplotlib.use('Agg').
- Fixed broken link to Quickstart on README
- Refactor of the original API to the scikit-learn Visualizer API

8.13.23 Version 0.2

Intermediate steps towards a complete API for visualization. Preparatory stages for scikit-learn visual pipelines.

- Tag: v0.2
- Deployed: Sunday, September 4, 2016
- Contributors: Benjamin Bengfort, Rebecca Bilbro, Patrick O'Melveny, Ellen Lowy, Laura Lorenz

Changes:

- Continued attempts to fix the Travis-CI Scipy install failure (broken tests)
- Utility function: get the name of the model
- Specified a class based API and the basic interface (render, draw, fit, predict, score)
- Added more documentation, converted to Sphinx, autodoc, docstrings for viz methods, and a quickstart
- How to contribute documentation, repo images etc.
- Prediction error plot for regressors (mvp)
- Residuals plot for regressors (mvp)
- Basic style settings a la seaborn
- ROC/AUC plot for classifiers (mvp)
- Best fit functions for "select best", linear, quadratic
- Several Jupyter notebooks for examples and demonstrations

8.13.24 Version 0.1

Created the yellowbrick library MVP with two primary operations: a classification report heat map and a ROC/AUC curve model analysis for classifiers. This is the base package deployment for continuing yellowbrick development.

- Tag: v0.1
- Deployed: Wednesday, May 18, 2016
- Contributors: Benjamin Bengfort, Rebecca Bilbro

Changes:

- Created the Anscombe quartet visualization example
- Added DDL specific color maps and a stub for more style handling
- Created crplot which visualizes the confusion matrix of a classifier
- Created rocplot_compare which compares two classifiers using ROC/AUC metrics
- Stub tests/stub documentation

8.14 Governance

Version 1.1 on May 15, 2019

8.14.1 Purpose

Yellowbrick has grown from a project with a handful of people hacking on it in their spare time into one with thousands of users, dozens of contributors, and several organizational affiliations. The burden of the effort to maintain the project has so far been borne by a few individuals dedicated to its success. However, as the project continues to grow, we believe it is important to establish a system that clearly defines how the project will be administered, organized, and maintained in the coming years. We hope that this clarity will lighten the administrative load of the maintainers and core-developers, streamline decision making, and allow new people to participate meaningfully. Most importantly, we hope that the product of these effects is to ensure the members of the Yellowbrick project are able to maintain a healthy and productive relationship with the project, allowing them to avoid burnout and fatigue.

The purpose of this document is to establish a system of governance that will define the interactions of Yellowbrick members and facilitate decision-making processes. This governance document serves as guidance to Yellowbrick members, and all Yellowbrick members must agree to adhere to its guidance in order to participate in the project. Although not legally binding, this document may also serve as bylaws for the organization of Yellowbrick contributors. We believe this document should be treated as a living document that is amended regularly in order to support the future objectives of the project.

8.14.2 Organization

The Yellowbrick project, also referred to as scikit-yellowbrick or scikit-yb, is an open source Python library for machine learning visual analysis and diagnostics. The library is licensed by the Apache 2.0 open source license, the source code is version controlled on GitHub, and the package is distributed via PyPI. The project is maintained by volunteer contributors who refer to themselves corporately as "the scikit-yb developers".

Founders and Leadership

Yellowbrick was founded by Benjamin Bengfort and Rebecca Bilbro, who were solely responsible for the initial prototypes and development of the library. In the tradition of Python, they are sometimes referred to as the "benevolent dictators for life", or BDFLs, of the project. For the purpose of this document and to emphasize the role of maintainers and core-contributors, they are simply referred to as "the founders". From a governance perspective, the founders have a special role in that they provide vision, thought leadership, and high-level direction for the project's members and contributors.

The Yellowbrick project is bigger than two people, however, therefore the primary mechanism of governance will be a collaboration of contributors and advisors in the tradition of the Apache Software Foundation. This collaboration ensures that the most active project contributors and users (the primary stakeholders of the project) have a voice in decision-making responsibilities. Note that Yellowbrick defines who active project contributors are (e.g. advisors, maintainers, core-contributors, and contributors) in a very specific and time-limited fashion. Depending on the type of required action, a subset of the active project contributors will deliberate and make a decision through majority voting consensus, subject to a final say of the founders. To ensure this happens correctly, contributors must be individuals who represent themselves and not companies.

Contributors and Roles

There are many ways to participate as a contributor to the Yellowbrick community, but first and foremost, all Yellowbrick contributors are users. Only by using Yellowbrick do contributors gain the requisite understanding and context to meaningfully contribute. Whether you use Yellowbrick as a student to learn machine learning (or as a teacher to teach it) or you are a professional data scientist or analyst who uses it for visual analytics or diagnostics, you cannot be a contributor without some use-case driven motivation. This definition also specifically excludes other motivations, such as financial motivations or attention-seeking. The reason for this is simple, Yellowbrick is a technical library that is intended for a technical audience.

You become a contributor when you give something back to the Yellowbrick community. A contribution can be anything large or small—related to code or something unique to your talents. In the end, the rest of the community of contributors will decide what constitutes a meaningful contribution, but some of the most common contributions include:

- Successfully merging a pull request after a code review.
- Contributing to the scikit-yb documentation.
- Translating the scikit-yb documentation into other languages.
- Submitting detailed, reproducible bug reports.
- Submitting detailed, prototyped visualizer or feature requests.
- Writing blog posts or giving talks that demonstrate Yellowbrick's use.
- Answering questions on Stack Overflow or on our GitHub issues.
- Organizing a Yellowbrick users meetup or other events.

Once you have contributed to Yellowbrick, you *will always be a contributor*. We believe this is a badge of honor and we hold all Yellowbrick contributors in special esteem and respect.

If you are routinely or frequently contributing to Yellowbrick, you have the opportunity to assume one of the roles of an active project contributor. These roles allow you to fundamentally guide the course of the Yellowbrick project but are also time-limited to help us avoid burnout and fatigue. We would like to plan our efforts as fixed-length sprints rather than as an open-ended run.

Note that the roles described below are not mutually exclusive. A Yellowbrick contributor can simultaneously be a core-contributor, a maintainer, and an advisor if they would like to take on all of those responsibilities at the same time. None of these roles "outranks" another, they are simply a delineation of different responsibilities. In fact, they are designed to be fluid so that members can pick and choose how they participate at any given time. A detailed description of the roles follows.

Core Contributor

A core-contributor commits to actively participate in a 4-month *semester* of Yellowbrick development, which culminates in the next release (we will discuss semesters in greater detail later in this document). At the beginning of the semester, the core-contributor will outline their participation and goals for the release, usually by accepting the assignment of 1-5 issues that should be completed before the semester is over. Core-contributors work with the maintainers over the course of the semester to ensure the library is at the highest quality. The responsibilities of core-contributors are as follows:

- Set specific and reasonable goals for contributions over the semester.
- Work with maintainers to complete issues and goals for the next release.
- Respond to maintainer requests and be generally available for discussion.
- Participating actively on the #yellowbrick Slack channel.

- Join any co-working or pair-programming sessions required to achieve goals.

Core-contributors can reasonably set as high or as low a challenge for themselves as they feel comfortable. We expect core-contributors to work on Yellowbrick for a total of about 5 hours per week. If core-contributors do not meet their goals by the end of the semester, they will likely not be invited to participate at that level in the next semester, though they may still volunteer subject to the approval of the maintainers for that semester.

Maintainer

Maintainers commit to actively manage Yellowbrick during a 4-month semester of Yellowbrick development and are primarily responsible for building and deploying the next release. Maintainers may also act as core-contributors and use the dedicated semester group to ensure the release objectives set by the advisors are met. This primarily manifests itself in the maintenance of GitHub issues and code reviews of Pull Requests. The responsibilities of maintainers are as follows:

- Being active and responsive on the #yellowbrick and #oz-maintainers Slack channels.
- Being active and responsive on the team-oz/maintainers GitHub group.
- Respond to user messages on GitHub and the ListServ (Stack Overflow, etc).
- Code review pull requests and safely merge them into the project.
- Maintain the project's high code quality and well-defined API.
- Release the next version of Yellowbrick.
- Find and share things of interest to the Yellowbrick community.

The maintainer role is an exhausting and time-consuming role. We expect maintainers to work on Yellowbrick 10-20 hours per week. Maintainers should have first been core-contributors so they understand what the role entails (and to allow a current maintainer to mentor them to assume the role). Moreover, we recommend that maintainers periodically/regularly take semesters off to ensure they don't get burnt out.

Coordinator

If a semester has a large enough group of maintainers and core-contributors, we may optionally appoint a contributor as a coordinator. Coordinators specialize in the project management aspects of a version release and also direct traffic to the actively participating group. The coordinator may or may not be a maintainer. If the coordinator is nominated from the maintainers, the coordination role is an auxiliary responsibility. Otherwise, the coordinator may be a dedicated contributor that focuses only on communication and progress. The coordinator's primary responsibilities are as follows:

- Ensure that the features, bugs, and issues assigned to the version release are - progressing.
- Be the first to respond welcomingly to new contributors.
- Assign incoming pull requests, issues, and other responses to maintainers.
- Communicate progress to the advisors and to the Yellowbrick community.
- Encourage and motivate the active contributors group.
- Coach core-contributors and maintainers.
- Organize meetups, video calls, and other social and engagement activities.

The coordinator's role is mostly about communication and the amount of dedicated effort can vary depending on external factors. Based on our experience it could be as little as a couple of hours per week to as much work as being a maintainer. The coordinator's responsibilities are organizational and are ideal for someone who wants to be a part of the community but has less of a software background. In fact, we see this role as being similar to a tech

project management role, which is often entry level and serves as experience to propel coordinators into more active development. Alternatively, the coordinator can be a more experienced maintainer who needs a break from review but is willing to focus on coaching.

Mentor-Contributor

A mentor-contributor assumes all the responsibilities of a core contributor but commits 25-50% of their time toward mentoring a new contributor who has little-to-no experience. The purpose of the mentor-contributor role is to assist in recruiting and developing a pipeline of new contributors for Yellowbrick.

A mentor-contributor would guide a new contributor in understanding the community, roles, ethos, expectations, systems, and processes in participating and contributing to an open source project such as Yellowbrick. A new contributor is someone who is interested in open source software projects but has little-to-no experience working on an open source project or participating in the community.

The mentor-contributor would work in tandem with the new contributor during the semester to complete assigned issues and prepare the new contributor to be a core contributor in a future semester. A mentor-contributor would mostly likely work on fewer issues during the semester in order to allocate time for mentoring. Mentor-contributors would be matched with new contributors prior to the start of each semester or recruit their own contributor (eg. colleague or friend). The responsibilities of mentor-contributors are as follows:

- When possible, recruit new contributors to Yellowbrick.
- Schedule weekly mentoring sessions and assign discrete yet increasingly complex tasks to the new contributor.
- Work with the new contributor to complete assigned issues during the semester.
- Set specific and reasonable goals for contributions over the semester.
- Work with maintainers to complete issues and goals for the next release.
- Respond to maintainer requests and be generally available for discussion.
- Participating actively on the #yellowbrick Slack channel.
- Join any co-working or pair-programming sessions required to achieve goals.
- Make a determination at the end of the semester on the readiness of the new contributor to be a core-contributor.

The mentor-contributor role may also appeal to students who use Yellowbrick in a machine learning course and seek the experience of contributing to an open source project.

Advisor

Advisors are the primary decision-making body for Yellowbrick. They serve over the course of 3 semesters (1 calendar year) and are broadly responsible for defining the roadmap for Yellowbrick development for the next three releases. Advisors meet regularly, at least at the beginning of every semester, to deliberate and discuss next steps for Yellowbrick and to give guidance for core-contributors and maintainers for the semester. The detailed responsibilities of the advisors are as follows:

- Contribute dues based on the number of advisors in a year to meet fixed running costs.
- Make decisions that affect the entire group through votes (ensure a quorum).
- Meet at least three times a year at the beginning of each semester (or more if required).
- Elect officers to conduct the business of Yellowbrick.
- Ensure Yellowbrick's financial responsibilities are met.
- Create a roadmap for Yellowbrick development.

- Approve the release of the next Yellowbrick version.
- Approve core-contributor and maintainer volunteers.
- Ensure Yellowbrick is growing by getting more users.
- Ensure Yellowbrick is a good citizen of the PyData and Python communities.
- Recruit more contributors to participate in the Yellowbrick community.

The advisor role is primarily intended to allow members to guide the course of Yellowbrick and actively participate in decisions without making them feel that they need to participate as maintainers (seriously, take a break - don't burn out!). As such, the role of advisors is limited to preparing for and participating in the semester meetings or any other meetings that are called. Therefore, we assume that the time commitment of an advisor is roughly 30-40 hours per year (less than 1 hour per week).

The board of advisors is open to every contributor, and in fact, is open to all users because joining the board of advisors requires a financial contribution. Yellowbrick has limited fixed running costs every year, for example, $10 a month to Read The Docs and $14.99 for our domain name. When you volunteer to be an advisor for a year, you commit to paying an equal portion of those fixed running costs, based on the number of advisor volunteers. This also ensures that we have a fixed quorum to ensure votes run smoothly.

Affiliations

Yellowbrick may be affiliated with one or more organizations that promote Yellowbrick and provide financial assistance or other support. Yellowbrick affiliations are determined by the advisors who should work to ensure that affiliations are in both the organization's and Yellowbrick's best interests. Yellowbrick is currently affiliated with:

- District Data Labs
- NumFOCUS
- Georgetown University

The advisors should update this section to reflect all current and past affiliations.

8.14.3 Operations

The primary activity of Yellowbrick contributors is to develop the Yellowbrick library into the best tool for visual model diagnostics and machine learning analytics. A secondary activity is to support Yellowbrick users and to provide a high level of customer service. Tertiary activities include being good citizens of the Python and PyData communities and to the open source community at large. These activities are directed to the primary mission of Yellowbrick: to greatly enhance the machine learning workflow with visual steering and diagnostics.

Although Yellowbrick is not a commercial enterprise, it does not preclude its members from pursuing commercial activities using Yellowbrick subject to our license agreement.

In this section, we break down the maintenance and development operations of the project. These operations are a separate activity from administration and decision making, which we will discuss in the following section.

Semesters and Service Terms

In order to ensure that maintainers are able to contribute meaningfully and effectively without getting burned out, we divided the maintenance and development life cycle into three semesters a year as follows:

- Spring semester: January - April
- Summer semester: May - August
- Fall semester: September - December

Every maintainer and core-contributor serves in these terms and is only expected to participate at the commitment level described by the roles section for the term they have joined. At the end of the term, there is no expectation that the maintainer must continue to the next term. If they wish to do so, they must volunteer to participate in the next term. We hope that this allows maintainers to be more strategic in their participation, e.g. participating every other semester, or taking off a semester where they know they will be busy with other obligations.

An advisor's service term is 1 year, and they must accept the advisory role by the end of January in that calendar year by paying the dues computed by the number of participating advisors. Advisors can remain advisors so long as they wish to participate by paying the dues, though if advising meetings are unable to achieve quorum; absent advisors may be asked to step down.

The goal of the advising service term is to allow maintainers who wish to take a semester off to still have a meaningful say in the development of the project. We hope that this will also facilitate maintainers feeling that they can take a break without being cut out of the loop, and allowing them to rejoin the project as maintainers or core-contributors in a meaningful way when they are ready again.

Release Guidelines

The focus of the semester is to release the next version of Yellowbrick. The advisors set a roadmap based on the current issues and features they wish to address. Core-contributors volunteer to take on specific issues and maintainers are responsible for organizing themselves and the core-contributors to complete the work in the version, releasing it at the end of the semester.

Maintainers may also include in the semester release any other contributions or pull requests made by members of the community at their discretion. Maintainers should address all pull-requests and opened issues (as well as emails on the listserv and questions on Stack Overflow, Twitter, etc.) - facilitating their inclusion in the release, or their closure if they become stale or are out of band.

During the middle of a semester, maintainers may be required to issue a hotfix release for time-critical or security-related fixes. Hotfix releases should be small, incremental improvements on the previous release. We hope that the expected release schedule of April, August, and December also assists the user community in giving feedback and guidance about the direction of Yellowbrick.

8.14.4 Advisors

Yellowbrick advisors are contributors who take on the responsibility of setting the vision and direction for the development of the project. They may, for example, make decisions about which features are focused on during a semester or to apply for a small development grant and use the funds to improve Yellowbrick. They may also ask to affiliate with other projects and programs that are tangential to but crucial to the success of Yellowbrick - e.g. organizing workshops or talks, creating t-shirts or stickers, etc.

Advisors influence Yellowbrick primarily through advisor meetings. This section describes advisor interactions, meetings, and decision-making structure.

Officers

During the first advisor meeting in January, three officers should be elected to serve in special capacities over the course of the year. The officer positions are as follows:

Chair: the chair's responsibility is to organize and lead the advisor meetings, to ensure good conduct and order, and to adjudicate any disputes. The chair may call additional advisor meetings and introduce proposals for the advisors to make decisions upon. The chair calls votes to be held and may make a tie-breaking vote if a tie exists. At the beginning of every meeting, the chair should report the status of Yellowbrick and the state of the current release.

Secretary: the secretary's responsibility is to create an agenda and record the minutes of advisor meetings, including all decisions undertaken and their results. The secretary may take the place of the chair if the chair is not available during a meeting.

Treasurer: the treasurer assumes responsibility for tracking cash flow—both dues payments from the advisors as well as any outgoing payments for expenses. The treasurer may also be responsible to handle monies received from outside sources, such as development grants. The treasurer should report on the financial status of the project at the beginning of every advisor meeting.

Officers may either self-nominate or be nominated by other advisors. Nominations should be submitted before the first advisor meeting in January, and the first order of business for that meeting should be to vote the officers into their positions. Officer votes follow the normal rules of voting as described below; in the case of a tie, the founders cast the tie-breaking decision.

Meetings

Advisors must schedule at least three meetings per year in the first month of each semester. At the advisor's discretion, they may schedule additional meetings as necessary. No vote may take place without a meeting and verbal deliberation (e.g. no voting by email or by poll). Meetings must be held with a virtual component. E.g. even if the meeting is in-person with available advisors, there must be some teleconference or video capability to allow remote advisors to participate meaningfully in the meetings.

Scheduling

To schedule a meeting, the chair must prepare a poll to find the availability of all advisors with at least 6 options over the next 10 days. The chair must ensure that every reasonable step is taken to include as many of the advisors as possible. No meeting may be scheduled without a quorum attending.

Meetings must be scheduled in January, May, and September; as close to the start of the semester as possible. It is advisable to send the scheduling poll for those meetings in December, April, and August. Advisors may hold any number of additional meetings.

Voting

Voting rules are simple—a proposal for a vote must be made by one of the advisors, submitted to the chair who determines if a vote may be held. The proposal may be seconded to require the chair to hold a vote on the issue. Votes may only be held if a quorum of all advisors (half of the advisors plus one) is present or accounted for in some way (e.g. through a delegation). Prior to the vote, a period of discussion of no less than 5 minutes must be allowed so that members may speak for or against the proposal.

Votes may take place in two ways, the mechanism of which must be decided by the chair as part of the vote proposal. The vote can either be performed by secret ballot as needed or, more generally, by counting the individual votes of advisors. Advisors can either submit a "yea", "nay", or "abstain" vote. A proposal is passed if a majority (half of the advisors plus one) submit a "yea" vote. A proposal is rejected if a majority (half of the advisors plus one) submit a

"nay" vote. If neither a majority "yea" or "nay" votes are obtained, the vote can be conducted again at the next advisors meeting.

The proposal is not ratified until it is agreed upon by the founders, who have the final say in all decision making. This can be interpreted as either a veto (the founders reject a passed proposal) or as a request for clarification (the founders require another vote for a rejected proposal). There is no way to overturn a veto by the founders, though they recognize that by "taking their ball and going home", they are not fostering a sense of community and expect that these disagreements will be extraordinarily rare.

Agenda and Minutes

The secretary is responsible for preparing a meeting agenda and sending it to all advisors at least 24 hours before the advisors meeting. Minutes that describe in detail the discussion, any proposals, and the outcome of all votes should be taken as well. Minutes should be made public to the rest of the Yellowbrick community.

8.14.5 Amendments

Amendments or changes to this governance document can be made by a vote of the advisors. Proposed changes must be submitted to the advisors for review at least one week prior to the vote taking place. Prior to the vote, the advisors should allow for a period of discussion where the changes or amendments can be reviewed by the group. Amendments are ratified by the normal voting procedure described above.

Amendments should update the version and date at the top of this document.

8.14.6 Board of Advisors Minutes

May 15, 2019

Yellowbrick Advisory Board Meeting Held on May 15, 2019 from 2030-2230 EST via Video Conference Call. Minutes taken by Benjamin Bengfort.

Attendees: Edwin Schmierer, Kristen McIntyre, Larry Gray, Nathan Danielsen, Adam Morris, Prema Roman, Rebecca Bilbro, Tony Ojeda, Benjamin Bengfort.

Agenda

A broad overview of the topics for discussion in the order they were presented:

1. Welcome and introduction (Benjamin Bengfort)
2. Officer nominations
3. Advisor dues and Yellowbrick budget
4. PyCon Sprints debrief (Larry Gray)
5. Summer 2019 contributors and roles
6. Google Summer of Code (Adam Morris)
7. Yellowbrick v1.0 milestone planning
8. Project roadmap through 2020
9. Proposed amendment to governance: mentor role (Edwin Schmierer)
10. Other business

Votes and Resolutions

Officer Elections

During the first advisory board meeting of the year, officers are nominated and elected to manage Yellowbrick for the year. The following nominations were proposed:

Nominations for Chair:

- Rebecca Bilbro by Benjamin Bengfort

Nominations for Secretary:

- Benjamin Bengfort by Rebecca Bilbro

Nominations for Treasurer:

- Edwin Schmierer by Rebecca Bilbro

Motion: a vote to elect the nominated officers: Rebecca Bilbro as Chair, Benjamin Bengfort as Secretary, and Edwin Schmierer as Treasurer. Moved by Nathan Danielsen, seconded by Adam Morris.

The motion was adopted unanimously.

Operating Budget

The following operating budget is proposed for 2019.

Last year's budget consisted of only two line items, which we believe will also primarily contribute to this year's budget. The total budget is **$266.49** we have 9 advisors this year, therefore the dues are $29.61 per advisor.

Budget breakdown:

Description	Frequency	Total
Name.com domain registration (scikit-yb.org)	annually	$12.99
Read the Docs Gold Membership	$10 monthly	$120.00
Stickers (StickerMule.com)	annually	$133.50
Datasets hosting on S3	monthly (DDL)	$10-15
Cheatsheets	annually (DDL)	$70

Based on this budget, it is proposed that the dues are rounded to **$30.00** per advisor. Further, dues should be submitted to the treasurer via Venmo or PayPal no later than May 30, 2019. The treasurer will reimburse the payees of our expenses directly.

A special thank you to District Data Labs for covering travel costs to PyCon, the printing of cheatsheets, and hosting our datasets on S3.

Motion: confirm the 2019 budget and advisor dues. Moved by Rebecca Bilbro, seconded by Nathan Danielsen.

The motion was adopted unanimously.

Governance Amendment

A proposed amendment to the governance document adding a "mentor-contributor" role as follows:

```
Mentor-Contributor
^^^^^^^^^^^^^^^^^^

A mentor-contributor assumes all the responsibilities of a core contributor but
commits 25-50% of their time toward mentoring a new contributor who has
little-to-no experience. The purpose of the mentor-contributor role is to assist in
recruiting and developing a pipeline of new contributors for Yellowbrick.

A mentor-contributor would guide a new contributor in understanding the community,
roles, ethos, expectations, systems, and processes in participating and
contributing to an open source project such as Yellowbrick. A new contributor is
someone who is interested in open source software projects but has little-to-no
experience working on an open source project or participating in the community.

The mentor-contributor would work in tandem with the new contributor during the
semester to complete assigned issues and prepare the new contributor to be a core
contributor in a future semester. A mentor-contributor would mostly likely work on
fewer issues during the semester in order to allocate time for mentoring.
Mentor-contributors would be matched with new contributors prior to the start of
each semester or recruit their own contributor (eg. colleague or friend). The
responsibilities of mentor-contributors are as follows:

- When possible, recruit new contributors to Yellowbrick.
- Schedule weekly mentoring sessions and assign discrete yet increasingly complex tasks␣
↪to the new contributor.
- Work with the new contributor to complete assigned issues during the semester.
- Set specific and reasonable goals for contributions over the semester.
- Work with maintainers to complete issues and goals for the next release.
- Respond to maintainer requests and be generally available for discussion.
- Participating actively on the #yellowbrick Slack channel.
- Join any co-working or pair-programming sessions required to achieve goals.
- Make a determination at the end of the semester on the readiness of the new␣
↪contributor to be a core-contributor.

The mentor-contributor role may also appeal to students who use Yellowbrick in a
machine learning course and seek the experience of contributing to an open source
project.
```

Motion: accept the amendment adding a mentor contributor role to the governance document. Moved by Edwin Schmierer, seconded by Rebecca Bilbro.

The motion was adopted unanimously without modifications.

Semester and Roadmap

The summer semester will be dedicated to completing **Yellowbrick Version 1.0**. The issues associated with this release can be found in the v1.0 Milestone on GitHub. This release has a number of deep issues: including figuring out the axes/figure API and compatibility. It is a year behind and it'll be good to move forward on it!

We have a critical need for maintainers, we only have 2-3 people regularly reviewing PRs and far more PRs than we can handle. So far there have been no volunteers for summer maintainer status. We will address this in two ways:

1. Maintainer mentorship
2. Redirect non v1.0 PRs to the fall

Rebecca will take the lead in mentoring new maintainers so that they feel more confident in conducting code reviews. This will include peer-reviews and a deep discussion of what we're looking for during code reviews. These code reviews will only be conducted on PRs by the summer contributors.

Otherwise PRs will be handled as follows this semester:

- Triage new PRs/Issues. If it is critical to v1.0 a bugfix or a docfix, assign a maintainer.
- Otherwise we will request the contributor join the fall semester.
- If a PR is older than 30 days it is "gone stale" and will be closed.
- Recommend contributors push to `yellowbrick.contrib` or write blog post or notebook.

We likely will need at least three maintainers per semester moving forward.

Summer 2019 Contributors

Name	Role
Lawrence Gray	Coordinator
Prashi Doval	Core Contributor
Benjamin Bengfort	Core Contributor
Saurabh Daalia	Core Contributor
Bashar Jaan Khan	Core Contributor
Piyush Gautam	Core Contributor
Naresh Bachwani	GSoC Student
Xinyu You	Core Contributor
Sandeep Banerjee	Core Contributor
Carl Dawson	Core Contributor
Mike Curry	Core Contributor
Nathan Danielsen	Maintainer
Rebecca Bilbro	Mentor
Prema Roman	Maintainer
Kristen McIntyre	Maintainer

Project Roadmap

The v1.0 release has a number of significant changes that may not be backward compatible with previous versions (though for the most part it will be). Because of this, and because many of the issues are *contagious* (e.g. affect many files in Yellowbrick), we are reluctant to plan too much into the future for Yellowbrick. Instead we have created a v1.1 milestone in GitHub to start tracking issues there.

Broadly some roadmap items are:

- *Make quick methods primetime*. Our primary API is the visualizer, which allows for the most configuration and customization of visualizations. The quick methods, however, are a simple workflow that is in demand. The quick methods will do all the work of the most basic visualizer functions in one line of code and return the visualizer for further customization.
- *Add a neural package for ANN specific modeling*. We already have a text package for natural language processing, as deep learning is becoming more important, Yellowbrick should help with the interpretability of these models as well.

Other roadmap ideas and planning discussed included:

- A yb-altair prototype potentially leading to an Altair backend side-by-side with matplotlib.
- Devops/engineering focused content for Yellowbrick (e.g. model management and maintenance).
- Fundraising to pay for more ambitious YB development.
- Having Yellowbrick attendees at more conferences.
- Determining who is really using Yellowbrick to better understand the community we're supporting.

Minutes

Comments on the Inaugural Advisors Meeting

Remarks delivered by Benjamin Bengfort

Welcome to the inaugural meeting of the Yellowbrick Board of Advisors. Due to the number of agenda items and limited time, this meeting will have a more formal tone than normal if simply to provide structure so that we can get to everything. Before we start, however, I wanted to say a few words to mark this occasion and to remark on the path that led us here.

When Rebecca and I started Yellowbrick 3 years ago this month, I don't think that we intended to create a top tier Python library, nor did we envision how fast the project would grow. At the time, we wanted to make a statement about the role of visual analytics - the iterative interaction between human and computer - in the machine learning workflow. Though Rebecca and I communicated to different audiences, we were lucky enough to say this at a time when our message resonated with both students and professionals. Because of this, Yellowbrick has enjoyed a lot of success across a variety of different metrics: downloads, stars, blog posts. But the most important metric to me personally is the number of contributors we have had.

Yellowbrick has become more than a software package, it has also become a community. And it is in no small part due to the efforts of the people here tonight. I think it is not inappropriate and perhaps can serve as an introduction for me to individually recognize your contributions to the rest of the group.

Starting at the beginning, *Tony Ojeda* has uniquely supported Yellowbrick in a way no one else could. He incubated the project personally through his company, District Data Labs, even though there was no direct market value to him. He put his money where his heart was and to me, that will always make him the model of the ideal entrepreneur.

Larry Gray joined Yellowbrick by way of the research labs. In a way, he was exactly the type of person we hoped would join the labs - a professional data scientist who wanted to contribute to an open source data project. Since then he has

become so much more, working hard to organize and coordinate the project and investing time and emotional energy which we have come to rely on.

Nathan Danielsen represents a counterpoint to the data science contributor and instead brings a level of software engineering professionalism that is sorely needed in every open source project. Although we often mention his work on image similarity tests - what that effort really did was to open the door to many new contributors, allowing us to review and code with confidence know the tests had our backs.

Prema Roman has brought a thoughtful, measured approach to the way she tackles issues and is personally responsible for the prototyping and development of several visualizers including JointPlot and CrossValidation. We have relied on her to deliver new, high-impact features to the library.

Kristen McIntyre is the voice of our users, an essential role we could not do without. Without her, we would be envious of scikit-learn's documentation. Instead, we are blessed to have clear, consistent, and extensive descriptions of the visualizers which unlock their use to new and veteran users of our software.

Adam Morris has become the voice of our project, particularly on Twitter. I'm joyously surprised every time I read one of his tweets. It is not the voice I would have used and that's what makes it so sweet. He has taken to heart our code of conduct's admonishments to over the top positivity and encouragement, and I'm always delighted by it.

Edwin Schmierer many of you may not have met yet - but without him, this project would never have happened. He is responsible for creating the Data Science program at Georgetown and he's been a good friend and advisor ever since. During our lunches and breakfasts together, we have often discussed the context of Yellowbrick in the bigger picture. I'm very excited to have him be able to give us advice more directly.

Every time I use Yellowbrick I can see or sense your individual impact and signature on the project. Thank you all so much for joining Rebecca and I in building Yellowbrick into what it is today. It is not an understatement that Yellowbrick has been successful beyond what we expected or hoped.

With success comes responsibility. I know that we have all felt that responsibility keenly, particularly recently as the volume of direct contacts with the community has been increasing. We have reached a stage where that responsibility cannot be borne lightly by a few individuals, because the decisions we make reach and affect a much larger group of people then we may intend. To address this, we as a group, have ratified a new governance document whose primary purpose is to help us organize and act strategically as a cohesive group. I believe that the structures we have put into place will allow us to move Yellowbrick even further toward being a professional grade software library and will help us minimize risks to the project - particularly the risk of maintainer burnout.

We are now making another statement at another pivotal moment because we are not alone in formalizing or changing our mode of governance. Perhaps the most public example of this is Python itself, but the message of maintainer-exhaustion and burnout has been discussed for the past couple of years in a variety of settings - OSCON, PyCON, video and podcast interviews, surveys and reports. It has caused growing concern and an essential question: "how do we support open source projects that are critical to research and infrastructure when they are not supported by traditional commercial and economic mechanisms"?

Many of you have heard me say that "I believe in open source the way I believe in democracy". But like democracy, open source is evolving as new contributors, new technology, and new cultures start to participate more. Yellowbrick is representative of what open source can achieve and what it does both for and to the people who maintain it. As we move forward from here, the choices we make may have an impact beyond our project and perhaps even beyond Python and I hope that you, like me, find that both exciting and terrifying.

Today we are here to discuss how we will conduct the project toward a version 1.0 release, a significant milestone, while also mentoring a Google Summer of Code Student and managing the many open pull requests and issues currently outstanding. The details matter, but they must also be considered within a context. Therefore I would like to personally commission you all toward two goals that set that context:

1. **To be a shining example of what an open source project should be.**
2. **To think strategically of how we can support a community of both users and maintainers.**

I'm very much looking forward to the future, and am so excited to be doing it together with you all.

PyCon 2019 Debrief

Yellowbrick had an open source booth at PyCon 2019 for the second time, manned by Larry, Nathan, Prema, and Kristen. There were lots of visitors to the booth who were genuinely excited about using Yellowbrick. Next year it would be great to have a large banner and vertical display of visualizers to help people learn more. Guido von Rossum even had lunch at the booth, which sparked a lot of attention! Special thank you to Tony for his financial support including shirts, stickers, transportation, and supplies.

We also hosted two days of development sprints with 13 sprinters and 9 PRs. The majority of sprinters were there for both days. The sprints included a wide variety of people including a high schooler and even a second year sprinter. Guido von Rossum also stopped by during the sprints. Prema and Kristen announced the sprints on stage and there is a great picture of them that we tweeted. Special thank you to Daniel for creating the cheatsheets, which cut down a lot of explaining, and to Nathan for the appetizers. Beets are delicious!

Hot takes from PyCon:

- Cheatsheets were a huge success and very important
- It would be good to have a "Contributing to Yellowbrick" cheatsheet for next year
- Have to capitalize on attention before the conference starts
- More visual stand out: e.g. metal grommets and banner with visualizers
- NumFOCUS has an in-house designer who may be able to help

Important request: we know that there will be a large number of PRs after PyCon or during Hacktoberfest, we need to *plan* for this and ensure we know who/how is triaging them and handling them ahead of the event. The maintainers who did not attend PyCon ended up slammed with work even though they couldn't enjoy participating in the event itself.

Google Summer of Code

We applied to GSoC under the NumFOCUS umbrella and have been given a student, Naresh, to work with us over the summer. Adam has already had initial communication with him and he's eager to get started. So far we've already interacted with him on Github via some of his early PRs and he's going to be a great resource.

In terms of the GSoC actions, the coding period starts May 27 and ends August 26 when deliverables are due. Every 4 weeks we do a 360 review where he evaluates us and we evaluate him (pass/fail). NumFOCUS requires a blog post and will work with Naresh directly on the posts.

We need to determine what he should work on. His proposal was about extending PCA, but Visual Pipelines are also extremely important. Proposal, break his work into three phases. First, get the first introductory PR across the finish line, then scope the ideas in his proposal and involve him in the visualizer audit. After working on blog posts and using YB on his own dataset, then get started on actual deliverables.

Action Items:

- Take note that Naresh is in India and is 9.5 hours ahead (time zone).
- Schedule introduction to team and maintainers.
- Coding period begins May 27.
- How do we want to manage communications (Slack)
- 360 evaluations due every 4 weeks
- Determine final deliverables due on Aug 19-26

Other Topics

Yellowbrick as a startup. Perhaps we can think big and pay for a full-time developer? There are many potential grant sources. The problem is that more money means more responsibility and we can't keep things together as it is now.

It would be good to send updates to NumFOCUS and District Data Labs to keep them apprised of what we're doing. For example:

- JOSS paper releases
- Sprints and conference attendance
- Version releases
- Talks or presentations

Hopefully this will allow them to also spread the word about what we're doing.

Action Items

- Send dues to Edwin Schmierer via Venmo/PayPal (all)
- Prepare more budget options for December board meeting (Treasurer)
- Create PR with Governance Amendment (Secretary)
- Modify documentation to change language about "open a PR as early as possible" (Larry)
- Add Naresh to Slack (Adam)
- Let Adam know what talks are coming up so he can tweet them (all)
- Send updates big and small to NumFOCUS (all)

September 9, 2019

Yellowbrick Advisory Board Meeting Held on September 9, 2019 from 2030-2230 EST via Video Conference Call. Rebecca Bilbro presiding as Chair, Benjamin Bengfort serving as Secretary and Edwin Schmierer as Treasurer. Minutes taken by Benjamin Bengfort.

Attendees: Benjamin Bengfort, Larry Gray, Rebecca Bilbro, Tony Ojeda, Kristen McIntyre, Prema Roman, Edwin Schmierer, Adam Morris, Eunice Chendjou

Agenda

A broad overview of the topics for discussion in the order they were presented:

1. Welcome (Rebecca Bilbro)
2. Summer 2019 Retrospective (Rebecca Bilbro)
3. GSoC Report (Adam Morris)
4. YB API Audit Results (Benjamin Bengfort)
5. OpenTeams Participation (Eunice Chendjou)
6. Fall 2019 Contributors and Roles
7. Yellowbrick v1.1 Milestone Planning

8. Project Roadmap through 2020
9. Other business

Votes and Resolutions

There were no votes or resolutions during this board meeting.

Summer 2019 Retrospective

In traditional agile development, sprints are concluded with a retrospective to discuss what went well, what didn't go well, what were unexpected challenges, and how we can adapt to those challenges in the future. Generally these meetings exclude the major stakeholders so that the contributors to the sprint can speak freely. Because the Board of Advisors are the major stakeholders of the Yellowbrick project, there was an intermediate retrospective with just the maintainers of the Summer 2019 semester so they could communicate anonymously and frankly with the Board. Their feedback as well as additions by the advisors follow.

Accomplishments

tl;dr we had a very productive summer!

- Larry stepped into the new coordinator role, setting the tone for future coordinators
- We had two new maintainers: Prema and Kristen
- Approximately 12 new contributors
- 65 Pull Requests were merged
- 35 tweets were tweeted
- Only 95 issues remain open (down from over 125, not including new issues)
- 3 Yellowbrick talks in DC, Texas, and Spain
- We completed our first Google Summer of Code successfully

Perhaps most importantly, Version 1.0 was released. This release included some big things from new visualizers to important bug/API fixes, a new datasets module, plot directives and more. We are very proud of the result!

Shoutouts

- To Larry for being the first coordinator!
- To Prema for shepherding in a new contributor from the PyCon sprints who enhanced our SilhouetteScores visualizer!
- To Kristen for working with a new contributor to introduce a brand new NFL receivers dataset!
- To Nathan for working with a new contributor on the cross-operating system tests!
- To Benjamin for shoring up the classifier API and the audit!
- To Adam and Prema for spearheading the GSoC application review and to Adam for serving as guide and mentor to Naresh, our GSoC student!
- To Rebecca for mentoring the maintainers!

Challenges

- It was difficult to adjust to new contributors/GSoC
- "Contagious issues" made it tough to parallelize some work
- Maintainer vacation/work schedules caused communication interruptions
- Balancing between external contributor PRs and internal milestone goals
- This milestone may have been a bit over-ambitious

Moving Forward

These are some of the things that worked well for us and that we should keep doing:

- Make sure the "definition of done" is well defined/understood in issues
- Balancing PR assignments so that no one gets too many
- Using the "assignee" feature in GitHub to assign PRs so that it's easier to see who is working on what tasks
- Use the maintainer's Slack channel to unify communications
- Communication in general – make sure people know what's expected of them and what to expect of us
- Getting together to celebrate releases!
- Pair reviews of PRs (especially for larger PRs)

Semester and Roadmap

The fall semester will be dedicated to completing **Yellowbrick Version 1.1**. The issues associated with this release can be found in the v1.1 Milestone on GitHub.

The primary milestone objectives are as follows:

1. Make quick methods prime time (and extend the oneliners page)
2. Add support for sklearn Pipelines and FeatureUnions

The secondary objectives are at the discretion of the core contributors but should be along one of the following themes:

1. A neural-network specific package for deep learning visualization
2. Adding support for visual pipelines and other multi-image reports
3. Creating interactive or animated visualizers

The maintainers will create a Slack channel and discuss with the Fall contributors what direction they would like to go in, to be decided no later than September 20, 2019.

Fall 2019 Contributors

Name	Role
Adam Morris	Coordinator
Prema Roman	Maintainer
Kristen McIntyre	Maintainer
Benjamin Bengfort	Maintainer
Nathan Danielsen	Maintainer
Lawrence Gray	Core Contributor
Michael Chestnut	Core Contributor
Prashi Doval	Core Contributor
Saurabh Daalia	Core Contributor
Bashar Jaan Khan	Core Contributor
Rohan Panda	Core Contributor
Pradeep Singh	Core Contributor
Mahkah Wu	Core Contributor
Thom Lappas	Core Contributor
Stephanie R Miller	Core Contributor
Coleen W Chen	Core Contributor
Franco Bueno Mattera	Core Contributor
Shawna Carey	Core Contributor
George Krug	Core Contributor
Aaron Margolis	Core Contributor
Molly Morrison	Core Contributor

Project Roadmap

With the release of v1.0, Yellowbrick has become a stable project that we would like to see increased usage of. The only urgent remaining task is that of the quick methods - which will happen in v1.1. Beyond v1.1 we have concluded that it would be wise to understand who is really using the software and to get feature ideas from them. We do have a few themes we are considering.

- *Add a neural package for ANN specific modeling*. We already have a text package for natural language processing, as deep learning is becoming more important, Yellowbrick should help with the interpretability of these models as well.
- *Reporting and data engineering focused content*. We could consider a text output format (like .ipynb) that allows easy saving of multiple visualizers to disk in a compact format that can be committed to GitHub, stored in a database, and redrawn on demand. This theme would also include model management and maintenance tasks including detecting changes in models and tracking performance over time.
- *Visual optimization*. This tasks employs optimization and learning to enhance the quality of the visualizers, for example by maximizing white space in `RadViz` or `ParallelCoordinates`, detecting inflection points as with the `kneed` port in `KElbow`, or adding layout algorithms for better clustering visualization in `ICDM` or the inclusion of word maps or trees.
- *Interactive and Animated visualizers*. Adding racing bar charts or animated TSNE to provide better interpetibility to visualizations or adding an Altair backend to create interactive Javascript plots or other model visualization tools like pyldaviz.
- *Publication and conferences*. We would like to continue to participate in PyCon and other conferences. We might also submit proposals to O'Reilly to do Yellowbrick/Machine Learning related books or videos.

These goals are all very high level but we also want to ensure that the package makes progress. Lower level goals such as adding 16 new visualizers in 2020 should be discussed at the January board meeting. To that end, advisors should look at how they're using Yellowbrick in their own work to consider more detailed roadmap goals.

Minutes

In her welcome, Rebecca described the goal of our conversation for the second governance meeting was first to talk about how things went over the summer, to celebrate our successes with the v1.0 launch and to highlight specific activities such as GSoC and the audit. The second half of the meeting is to be used to discuss our plans for the fall, which should be more than half the conversation. In so doing she set a technical tone for the mid-year meetings that will hopefully serve as a good guideline for future advisory meetings.

Google Summer of Code

Adam reports that Naresh successfully completed the GSoC period and that he wrote a positive review for him and shared the feedback we discussed during the v1.0 launch. You can read more about his summer at his blog, which documents his journey.

Naresh completed the following pull requests/tasks:

1. Added train alpha and test alpha to residuals
2. Added an alpha parameter to `PCA`
3. Added a stacked barchart helper and stacking to `PoSVisualizer`
4. Updated several visualizers to use the stacked barchart helper function
5. Updated the `DataVisualizer` to handle target type identification
6. Added a `ProjectionVisualizer` base class.
7. Updated Manifold and PCA to extend the `ProjectionVisualizer`
8. Added final tweaks to unify the functionality of `PCA`, `Manifold`, and other projections.

We will work on sending Naresh a Yellowbrick T-shirt to thank him and have already encouraged him to continue to contribute to Yellowbrick (he is receptive to it). We will also follow up with him on his work on effect plots.

If we decide to participate in GSoC again, we should reuse the idea list for the application, but potentially it's easiest to collaborate with matplotlib for GSoC 2020.

API Audit Results

We conducted a full audit of all visualizers and their bases in Yellowbrick and categorized each as red (needs serious work), yellow (has accumulated technical debt), and green (production-ready). A summary of these categorizations is as follows:

- There are 14 base classes, 1 red, 3 yellow, and 10 green
- There are 36 visualizers (7 aliases), 4 red, 7 yellow, 25 green
- There are 3 other visualizer utilities, 2 red, 1 green
- There are 35 quick methods, 1 for each visualizer (except manual alpha selection)

Through the audit process, we clarified our API and ensured that the visualizers conformed to it:

- `fit()` returns `self`, `transform()` returns `Xp`, `score()` returns [0,1], `draw()` returns `ax`, and `finalize()` returns `None` (we also updated `poof()` to return `ax`).
- No _ suffixed properties should be set in `__init__()`
- Calls to `plt` should be minimized (and we added `fig` to the visualizer)
- Quick methods should return the fitted/scored visualizer

Additionally, we took into account the number/quality of tests for each visualizer, the documentation, and the robustness of the visualization implementation to rank the visualizers.

Along the way, a lot of technical debt was cleaned up; including unifying formatting with black and flake8 style checkers, updating headers, unifying scattered functionality into base classes, and more.

In the end, the audit should give us confidence that v1.0 is a production-ready implementation and that it is a stable foundation to grow the project on.

OpenTeams

Eunice Chendjou, COO of OpenTeams, joined the meeting to observe Yellowbrick as a model for successful open source community governance, and to let the Advisory Board know about OpenTeams. OpenTeams is designed to highlight the contributions and work of open source developers and to help support them by assisting them in winning contracts and finding funding. Although currently it is in its initial stages, they have a lot of big plans for helping open source teams grow.

Please add your contributions to Yellowbrick by joining OpenTeams. Invite others to join as well!

Action Items

- Add your contributions to the Yellowbrick OpenTeams projection
- Send invitations to those interested in joining the 2020 board (all)
- Begin considering who to nominate for January election of board members (all)
- Send Naresh a Yellowbrick T-shirt or thank you (Adam)
- Create the Fall 2019 contributors Slack channel (Benjamin)
- Start thinking about how to guide the 2020 roadmap (all)
- Publications task group for O'Reilly content (Kristen, Larry)

January 7, 2020

Yellowbrick Advisory Board Meeting Held on January 7, 2020 from 2030-2230 EST via Video Conference Call. Rebecca Bilbro presiding as Chair, Benjamin Bengfort serving as Secretary and Edwin Schmierer as Treasurer. Minutes taken by Benjamin Bengfort.

Attendees: Rebecca Bilbro, Benjamin Bengfort, Kristen McIntyre, Tony Ojeda, Larry Gray, Prema Roman, Edwin Schmierer, Nathan Danielsen, and Adam Morris.

Agenda

A broad overview of the topics for discussion in the order they were presented:

1. Welcome (Rebecca Bilbro)
2. Fall 2019 retrospective (Rebecca Bilbro)
3. Yellowbrick v1.2 milestone planning (Benjamin Bengfort)
4. Advisor dues and Yellowbrick Budget (Edwin Schmierer)
5. Officer nominations and election
6. Spring 2020 contributors and roles

Votes and Resolutions

Operating Budget

The following operating budget is proposed for 2020.

The board discussed a minor increase in operating costs from the previous year and acknowledged a 2020 budget of **$271.48** and **$30.17** advisor member dues per person. Because of the minimal change, the board is leaving the final budget open to any additional proposed expenses by January 14.

Budget breakdown:

Description	Frequency	Total
Name.com domain registration (scikit-yb.org)	annually	$17.98
Read the Docs Gold Membership	$10 monthly	$120.00
Stickers (StickerMule.com)	annually	$133.50
Datasets hosting on S3	monthly (DDL)	$10-15
Cheatsheets	annually (DDL)	$70

Advisors to pay dues to Edwin via Venmo. Edwin will send his Venmo handle and QR code via YB Slack channel. Dues are payable by Feb 14, 2020. If any advisors would like to make changes or add to the budget, please contact Edwin by January 14 via email or Slack. Edwin will take suggestions to the board for discussion and vote by the end of January.

A special thank you to District Data Labs for covering travel costs to PyCon, the printing of cheatsheets, and hosting our datasets on S3 for another year!

Motion: any opposed to the current budget and leaving the budget open until the end of January. Moved by Edwin Shmierer, seconded by Benjamin Bengfort.

No opposition to the motion.

Officer Elections

During the first advisory board meeting of the year, officers are nominated and elected to manage Yellowbrick for the year. Current officers gave an overview of the role and responsibilities for Chair, Secretary, and Treasurer, respectively. The following nominations were proposed:

Nominations for Chair:

- Kristen McIntyre by Benjamin Bengfort

Motion: a vote to elect Kristen McIntyre as 2020 Chair. Moved by Rebecca Bilbro, seconded by Larry Gray.

The motion was adopted unanimously.

Nominations for Secretary:

- Larry Gray by Rebecca Bilbro

Motion: a vote to elect Larry Gray as 2020 Secretary. Moved by Rebecca Bilbro, seconded by Prema Roman.

The motion was adopted unanimously.

Nominations for Treasurer:

- Edwin Schmierer by Rebecca Bilbro

Motion: a vote to elect Edwin Shmierer as 2020 Treasurer. Moved by Rebecca Bilbro, seconded by Kristen McIntyre.

The motion was adopted unanimously.

Kristen McIntyre and Larry Gray addressed the board expressing gratitude for the opportunity to serve and voicing their commitment to fulfilling the responsibilities to the best of their abilities.

Fall 2019 Retrospective

In traditional agile development, sprints are concluded with a retrospective to discuss what went well, what didn't go well, what were unexpected challenges, and how we can adapt to those challenges in the future.

Accomplishments

- Semester has been a lot lighter than previous semesters, focused on quick methods.
- Fixing a bug with PCA visualizer (regression made with projection argument).
- Decision to move from `poof()` to `show()` – we made a decision and executed on it (even in the face of a difficult situation).
- Test coverage has been catching things well in advance of problems (and before users experience them).
- Dependency set is small, which limits the number of problems we can have from them.
- Had a lot of contributors in the fall group (at least on the roster).
- 6 new contributors who opened PRs.
- 12 participants in the "burrito poll" made the API decision of the quick method signature.

Shoutouts

- Matt Harrison featured Yellowbrick in his machine learning book.
- Molly Morrison, Michael Chestnut, Michael Garod, Rohan Panda, George Krug, and Stephanie Miller all contributed to the quick methods development.
- Prema Roman has done a wonderful job this semester, stepped up and responding on GitHub, helping with new contributors, responding to questions.
- Guidance document made it easier and clearer about how to take a break for a semester.

Challenges

Potentially the biggest challenge Yellowbrick had this semester was a note from a user that the name of one of our methods was potentially a homophobic slur. Although the board considered the name to be one of the unique and charming features of Yellowbrick, we quickly decided to make a change to the method name, particularly as we were in the process of incrementing a major version number, which meant support for backward compatibility was not required.

- Updated dependencies breaking CI (numba, umap-learn, sklearn, scipy) – happened 2x
- Inconsistent behavior between miniconda and vanilla Python
- Jobs and other priorities have been swamping our ability to contribute to YB as much as we'd like to; the roles are still a bit new and we wish we could do more than we are
- There wasn't a lot of activity from all of the contributors; perhaps we didn't engage enough? Or maybe we want to have a range of issues available for different skill/experience levels?
- 7 quick methods are completed, 3 or 4 in review, 35 were assigned - most people who were assigned quick methods didn't show up after the start of the semester.
- 10 new contributors who signed up but didn't open a PR or participate much

Semester and Roadmap

v1.1 Status

The v1.1 release has not yet been finalized because the goal of updating all of the quick methods has not been completed. Because we've done the hard work of figuring out the API signature and documentation mechanism, the board feels that we should finish all the quick methods before releasing v1.1 *before the end of February*.

In order to complete v1.1 the board will hold one or two hackathons (either in person or virtual) to complete the quick methods together and with food and drink.

Spring 2020: v1.2

The Spring 2020 goal is to have everyone implement and review one new visualizer. Implementing visualizers from scratch is much more fun than trying to improve current visualizers or fix bugs - so the focus of this semester is fun! We'll be looking at Probability Curves, Missing Values, Multimodal Visualizers and more, trying to have a flashy semester!

Visualizer implementation and review assignments will be handed out at the beginning of the semester depending on preference.

Additionally, the board will prepare for PyCon sprints by implementing cheatsheets and preparing blog posts and other materials for people to quickly get started with Yellowbrick, scikit-learn, and matplotlib.

Summer 2020: v1.3

It is likely that the Summer 2020 goal will be to implement visual pipelines. This is a research oriented task that doesn't have a clear path forward, so some planning before the semester is required. This is a major new feature to the libary.

Spring 2020 Contributors

Note: those wishing to sign up as core contributors must have previously successfully contributed to Yellowbrick by closing a PR. Yellowbrick is open to unassigned, non-core submissions from anyone at any time.

Name	Role	Visualizer	Reviewer
Rebecca Bilbro	Maintainer	yes	yes
Benjamin Bengfort	Core Contributor	yes	yes
Nathan Danielsen	Maintainer	yes	yes
Kristen McIntyre	Core Contributor	yes	yes
Larry Gray	Core Contributor	yes	yes
Adam Morris	Coordinator	pair	yes
Prema Roman	Maintainer	yes	yes

Minutes

This meeting was primarily focused on electing the new board members and doing some initial planning for the year ahead. No additional topics were discussed.

Action Items

- Sign up for contributor roles and specify implement or review a visualizer (all)
- Review the 2020 budget and add any additional budget requests (all)
- Coordinate quick methods hackathon to finalize v1.1 (Kristen McIntyre)
- Assign visualizers and reviewers (Benjamin Bengfort)
- Update v1.1, v1.2, and v1.3 milestones on GitHub (Benjamin Bengfort & Larry Gray)
- Coordinate PyCon sprints with maintainers (Larry Gray and Prema Roman)

May 13, 2020

Yellowbrick Advisory Board Meeting Held on May 13, 2020 from 2030-2200 EST via Video Conference Call. Kristen McIntyre presided as Chair, Larry Gray serving as Secretary and Edwin Schmierer as Treasurer. Minutes taken by Larry Gray.

Attendees: Rebecca Bilbro, Benjamin Bengfort, Kristen McIntyre, Larry Gray, Prema Roman, Edwin Schmierer and Adam Morris.

Agenda

A broad overview of the topics for discussion in the order they were presented:

1. Welcome (by Kristen McIntyre)
2. Annual Budget Update
3. Community Code of Conduct
4. Spring 2020 Semester Retrospective
5. Yellowbrick v1.2 Milestone planning
6. Summer 2020 contributors and roles

Annual Budget Updated

In January, we approved our budget for the year, which totaled $271.48. Since we have 9 advisors for this year, the dues totalled $30.17 per advisor. Thank you to everyone for paying your dues on time!

- In our January meeting, it was noted that if someone had something they'd like to add to the budget, we could put it to a vote the next semester.
- We will likely have a little extra since a large portion of the stickers cost was intended for PyCon stickers.

2020 Annual Budget

Description	Frequency	Total
Name.com domain registration (scikit-yb.org)	annually	$17.98
Read the Docs Gold Membership	$10 monthly	$120.00
Stickers	annually	$133.50
		$271.48

Community Code of Conduct

Topic Background

- GitHub includes a *Community Profile* for each public repository, which shows how the project compares to GitHub's recommended community standards. This is done via a checklist to see if a project includes recommended community health files, such as README, CODE_OF_CONDUCT, LICENSE, or CONTRIBUTING.
- Yellowbrick does have a page in our docs; however, GitHub only checks specifically for code of conduct files it recognizes in a supported location (.github/), so not just the existence of a CODE_OF_CONDUCT.md file. We should add ours so that our Community Profile is complete. We should should discuss how someone report a violation of our policy?

Discussion/Resolution

- We currently use the PSF code of conduct.
- We discussed the need to set up an emergency contact procedure. Benjamin will set up a generic email at scikit-yb.org that can be forwarded. Rebecca Bilbro, Larry Gray, and Kristen McIntyre all agreed to accept these emails.
- The group has decided to convene later on and decide how to go about handling complaints then publish the agreed upon procedure. Benjamin Bengfort suggested that we create a designated role for handling complaints and Prema countered that it should be 2 individuals for this role.
- The invention of a new role for handling complaints was not voted upon.

Spring 2020 Semester Retrospective

In traditional agile development, sprints are concluded with a retrospective to discuss what went well, what didn't go well, what were unexpected challenges, and how we can adapt to those challenges in the future.

The primary focus of the Sprint 2020 Semester was on visualizers.

Accomplishments

What were some of the key achievements and successes this past semester? Which Visualizers have been added and what has changed?

- Hackathon on February 6th to finish remaining quick methods before v1.1 release. Over the five days, we merged in 23 PRs to finally tackle it!
- At the soldout "Python 2 End of Life Celebration" conference in early February, four Yellowbrick board members (Larry, Prema, Adam, Kristen) were invited to speak at the day's final panel, "Python for the People: Open Source Development."
- A full house for Rebecca Bilbro's excellent "Visual Diagnostics for Machine Learning with Python" presentation at the February 25th DC Python Meetup event.
- Prema and Kristen presented at Statistical Seminars DC's April 30th meetup, "Machine Learning with Visualization" that was attended by over 30 people.
- Advisory Board hosted a virtual Happy Hour on April 15th
- Since our last meeting, we had 3 new contributors have their first PRs merged in, compared to 5 in the Fall when we were actively recruiting new contributors.
  - One of our new contributors, VladSkripniuk, had three PRs merged!
    - Added Q-Q Plot to the Residuals Plot. Larry did a great job helping a new contributor bring this one over the goalline!
    - Updated the ROCAUC Visualizer to be more robust.
    - Allows users to specify colors for PrecisionRecallCurve.
  - Another new contributor this semester, Express50, not only found a bug but also opened PR to fix it, by adding the sample_weight parameter to the KElbowPlotVisualizer.
  - Our third new contributor to have their PR merged in, ekwska, had their first PR merged in back in January. It introduces a new feature that allows the user to optionally parse their raw text documents directly using the PosTagVisualiser

Challenges/Issues

What are some of the challenges and issues faced by the maintainer and contributor team this past semester (technical or otherwise)?

- Larry's gave advice for dealing with these Challenges/Issues
  - Don't underestimate the time required to produce a high quality PR.
  - It is ok to disagree with a contributor as long as it is done in a respectful and kind-hearted manner.
  - Bugs happen when you least expect it but you have to roll with the punches, lean on other maintainers, and see it as an opportunity to better understand the internals of YB.
  - Don't forget that you are not alone and asking for help is ok.
  - Do move quickly to finish the PRs, stay engaged with the contributor but don't beat yourself up when life gets in the way.
- The quick methods for certain visualizers presented unexpected issues, such as the Pandas 1.0 release and the ResidualsPlot needed additional tests to take into account edge cases.
- Evergreen issue: Matplotlib updates & Travis Errors
- CI Pipeline : After the hackathon there was a surge in the number of PRs to merge and the process was slow because of the CI. Rebecca had to manually direct PR traffic. We wondered at the time if we could increase the number of concurrent jobs then the issue would be more tolerable. Larry investigated our current setup. We currently operate on the opensource/free versions of Travis/appveyor. We found out that Travis can run 5 concurrent jobs and It isn't obvious how much additional concurrent jobs will cost. However, we know that the private repos Appveyor can only run 1 concurrent job and it cost $99 per month to increase to 2 concurrent jobs

Shoutouts

Who deserves special recognition for their contributions to Yellowbrick this past semester?

- Rebecca for jumping into the deep end and saving Larry's PR
- Adam for continuing to do such an excellent job monitoring the Yellowbrick Twitter account!
- Rebecca for hosting our Hackathon at ICX Media and arranging for food for everyone when we lost our previously scheduled space at the last minute.
- Larry for all of his hard work helping one of our newest contributors on their first PR, who has already had two more PRs merged in since then!
- Kristen is very grateful to Prema for taking the lead on the Statistical Society DC meetup last month!
- Evergreen shoutout: to Ben for doing "Ben-type" things
- Adam to Kristen, "Thanks for your positivity!"

Minutes

In Kristen's first meeting as chair she opened the meeting by acknowledging the challenges we all have faced because of COVID-19 Pandemic and its effect on Yellowbrick development. She delivered a message about solidarity and perseverance. She outlined that we would talk about the successes and challenges of the Spring Semester, discuss our code of conduct then wrap things up with v1.2 status updates and milestone planning.

Semester and Roadmap

Yellowbrick v1.2 Status Updates

The issues that are part of this milestone can be found here: https://github.com/DistrictDataLabs/yellowbrick/milestone/15

- The primary goal of the Spring semester was to focus on completing the issues included in the v1.2 milestone, which included the creation of seven new Visualizers that we would like to have added to Yellowbrick.

NewVisualizer	Issue #	Assigned To:	Reviewed By:
AnimatedFeatures	507	Nathan	Prema
DetectionError Trade-off(DET)	453	Ben and Adam	Naresh
Effect Plot	604	Naresh	Rebecca
MostInformativeFeatures	657	Larry	Ben
ProbabilityCalibra-tionCurve	365	Kristen	Naresh
Topic Saliency	570	Prema	Nathan
Tree-DepthPlot	305	Rebecca	Kristen

- Another focus of the Spring semester had been to update the Yellowbrick cheatsheet and prepare for the PyCon sprints; however, this didn't occur due to the conference unfortunately having to be canceled this year.

Milestone Planning: Yellowbrick v1.2

The following questions were asked of the technical board: What (if any) additional work should we commit to before doing the 1.2 version release? Which issues should be moved to v1.3 or to the backlog? What would we like the focus of development to be for the Summer semester? Should we Implement Black code formatting as pre-commit? Should the summer's focus be finishing our assigned Visualizers? Should we accept new contributors for the Summer semester, or delay until the Fall?

Decisions for Semester

- Focus on Visualizers for Semester
- Do not take on any new contributors this Semester

Action Items

- Create a generic email for emergency contact (Benjamin Bengfort)

October 6, 2020

Yellowbrick Advisory Board Meeting Held on October 6, 2020 from 2030-2200 EST via Video Conference Call. Kristen McIntyre presided as Chair, Larry Gray serving as Secretary and Edwin Schmierer as Treasurer. Minutes taken by Larry Gray.

Attendees: Rebecca Bilbro, Benjamin Bengfort, Kristen McIntyre, Larry Gray, Prema Roman, Edwin Schmierer, Tony Ojeda and Adam Morris.

Agenda

A broad overview of the topics for discussion in the order they were presented:

1. Welcome (by Kristen McIntyre)
2. Treasurer Update
3. Summer 2020 Semester Retrospective
4. Yellowbrick v1.2 Milestone planning
5. Member Topics
    a. Departure from AppVeyor Usage
    b. Managing Dependency upgrades
    c. New Handling of Third-Party Contribs
6. Additional focus for the Fall

Treasurer Update

There are no major issues to report. As a reminder, in January, we approved our budget for the year, which totaled $271.48. Since we have 9 advisors for this year, the dues totalled $30.17 per advisor. All dues have been paid and all members have been reimbursed for expenses incurred on behalf of YB.

2020 Annual Budget

Description	Frequency	Total	Paid By	Reimbursed
Name.comdomainregistration(scikit-yb.org)	annually	$17.98	Ben	Yes
Read theDocs Gold-Membership	$10/ month	$120.00	Ben	Yes
Stickers	annually	$133.50	Rebecca	Yes
		$271.48		

Question: What, if any, budget changes do we anticipate in 2021? No Budget changes anticipated in 2021.

Summer 2020 Semester Retrospective

Here were some of the key achievements and successes this past semester?

Yellowbrick Statistics May - October:

Issues closed: 20

Issues from this time period that remain open: 6

PRs from this time period that remain open: 3

Approved PRs (Contributors): 19

Rebecca - 14 Big Props!, Ben - 1, @Tktran - 2, @AlderMartinez - 1, @Melonhead901 - 1

A Summary of Approved PR Topics - Complete changelog Since v1.1 can be found here. (https://github.com/DistrictDataLabs/yellowbrick/pull/1110)

Major events:

1. Removed Appveyor from CI Matrix
2. Split Prediction Error Plot from Residuals Plot

Minor events:

1. Third-Party Estimator Wrappers/Checks
2. Improved Cook's Distance and Manifold Documentation
3. Improved RankD Tests and Upgrade
4. Reintegrated spec and verbose as pytest addopts
5. Small fixes to Manual Alpha Selection and ROCAUC

Bugs:

1. Matplotlib 3.3/3.2.2 Test/Image Failure
2. Quick-Start Plot Directive Fix
3. Fixed Errors/Warnings due to upgrading dependencies

Challenges/Issues

What are some of the challenges and issues faced by the maintainer and contributor team this past semester (technical or otherwise)?

- Covid-19: No one expected the global pandemic…

Shout-outs

Who deserves special recognition for their contributions to Yellowbrick this past semester?

- Adam for always doing such a great job with our Twitter account
- Evergreen shoutout: to Ben for doing "Ben-type" things
- Rebecca for being a PR Master Blaster

Votes/Resolutions

Ben proposed that we immediately release v1.2. The release would contain non-trivial changes. Votes were cast on whether we should issue a release and all in attendance voted that it be done immediately.

We also resolved that we would lock all dependency versions in place then proactively upgrade Yellowbrick in accordance to dependency changes (see Member Topics Below).

Member Topics

We discussed:

1. Managing Dependency Upgrades (Discussion Leader: Ben)
    a. Overarching Question: How do we manage this technical debt?
        i. Different approaches proposed:
            1. The most active approach would be to deprecate old quickly and refactor our code to meet new upgrades. This would involve rewriting tests and Visualizers
            2. A less active approach would be to only handle depreciation warnings
            3. The weakest form of a passive approach would be to limit dependencies
            4. The strongest form of a passive approach would be to freeze the repo

        ii. Rebecca noted that determining which dependency changes cause the errors is the hardest problem we face and it takes hours of work to find them.
            1. She requests that we don't add any new dependencies and that she will deny and PR that introduces them

        iii. Prema suggested that we convene on a case-by-case basis and discuss whether we should take an active/passive approach

        iv. Rebecca noted that PIP is going to be changing soon and we need to put it as a Topic for the next Meeting

        v. Ben proposed that we fix test dependency with == and maintain knowledge of upgrades. He notes that it is a risky approach because we have to stay on top of changes in dependencies.

    b. Decided ACTION
        i. We will freeze dependency before 1.2 release and be proactive about upgrading dependencies
2. Departure from Appveyor usage. Due to trouble tracking down Errors. Rebecca deceived to abandon usage.
3. Rebecca recommended a reading by Nadia Eghbal called Working in Public
4. New handling of Third-Party Contribs (Discussion Leader Ben)

a. The work was pursued because we are highly dependent on properties and attributes of Scikit-learn API and the way we handle inputs prevents third- party integration beyond scikit-learn. In response, we provided a new framework to handle third-party Estimators and move them to the contrib. We now provide a nice response when non-sklearn estimators don't work.

5. Focus for the Fall
    a. Ben suggested a blog on Skorch and Yellowbrick.
        i. It should represent the compatibility of Skorch and Yellowbrick
        ii. Rebecca recommends using her mini-lab on LSTM as a template
    b. Rebecca will be giving a talk that discusses some Yellowbrick at PyData Global Talk

Actions Items

1. Release version 1.2
2. Freeze dependency before 1.2 release

January 13, 2021

Yellowbrick Advisory Board Meeting Held on January 13, 2021 from 2030-2200 EST via Video Conference Call. Kristen McIntyre presided as Chair, Larry Gray serving as Secretary and Edwin Schmierer as Treasurer. Minutes taken by Adam Morris.

Attendees: Rebecca Bilbro, Benjamin Bengfort, Kristen McIntyre, Larry Gray, Prema Roman, Edwin Schmierer and Adam Morris.

Agenda

A broad overview of the topics for discussion in the order they were presented:

1. Welcome (by Kristen McIntyre)
2. Fall 2020 Semester Retrospective
3. 2021 Advisory Board Elections
4. 2021 Budget
5. Yellowbrick v1.3 Milestone planning
6. Member Topics
7. Additional Focus for the Spring Semester
8. Spring 2021 contributors and roles

Fall 2021 Semester Retrospective

- Rebecca gave a talk "Thrifty Machine Learning" and was highlighted by PyLadies Berlin (and they highlighted Yellowbrick!) on the 19th day of their 2020 [Advent Calendar Tweet Series](https://twitter.com/PyLadiesBer/status/1340321653839040513?s=20)
- The entire team showed resiliency in the face of COVID-19 and being unable to meet in person by moving the project forward.
- We closed 19 issues and had 13 open issues. We had 3 open PRs.
- Approved PRs (Contributors): Rebecca and Ben approved 3, Larry approved 1, Michael Garod and @arkvei approved 1 each.
- Summary of Fall PR Topics- Complete changelog since v1.2 can be found [here:](https://github.com/DistrictDataLabs/yellowbrick/pull/1110)

Main PR Topic Areas included: Yellowbrick1.2 release, Dependence Management issue [PR 1111](https://github.com/DistrictDataLabs/yellowbrick/pull/1111), update to Dispersion plot color and title, update to kneed algorithm, added FAQ on wrapper, third party estimator wrapper, adjustment to top_n param for feature importances. We also addressed a public/private API bug in [PR 1124](https://github.com/DistrictDataLabs/yellowbrick/pull/1124)

Board Shout-outs

- Ben for his constant contributions to the project
- Rebecca for her rapid response to issues all Semester long.
- Adam for meeting the 1000 mark for Social Media Followers (@scikit-yb)
- Strong user interactions with the library: 2300 downloads per day and 60,000 per month!
- Robust package evidenced by low number of issues being opened over the Semester.
- Kristen, Larry, Edwin for "surviving the 2nd worst year"

2021 Advisory Board

Nominations for officer positions for the 2021 Yellowbrick Advisory Board were revealed and voted upon. Before the reveal, Kristen McIntyre, Larry Gray and Edwin Schmierer provided an overview of the roles.

Nominations for Chair: Rebecca and Ben nominate Prema Tony nominated Ben **Result: Prema was selected with all Yay Votes**

Nominations for Secretary: Rebecca and Ben nominated Prema Kristen nominated Adam **Result: Adam was selected unanimously**

Nomnations for Treasurer: Rebecca, Ben and Larry nominated Adam Prema nominated herself Tony nominated Edwin **Result: Kristen was selected unanimously**

2021 Annual Budget Update

In January, we approved our budget for the year, which totaled **$271.48** . Since we have 9 advisors for this year, the dues totaled $30.17 per advisor although we are waiting to see if new members join the board which could lower the per person due amount. Thank you to everyone for paying your dues on time!

- In our January meeting, it was noted that if someone had something they'd like to add to the budget, we could put it to a vote the next semester.
- We will likely have a little extra since a large portion of the stickers cost was intended for PyCon stickers.
- The Treasurer, Edwin, provided an update that we needed to approve the budget.
- The group discussed and decided removing stickers from the 2021 budget due to COVID-19 and the lack of in person events (previously cost $133.50 and paid for by Rebecca. Thank you Rebecca!)
- The board decided to reallocate this sticker money towards buying small thank you gifts for developers who make significant contributions to Yellowbrick.
- *Ben suggested that we add an additional item to the budget* Add cost for gifts to Reviews and Contributors* - such as coffee and a YB branded T-Shirt. This is to show the YB spirit of Gratitude. A budget of $750. We have two potential sponsors (detailed below)
- Proposal 2 lines of budget: Board gifts 8 of us - Create budget $320 External Funding, External Gifts paid for partly by board dues/external funding:
  1. We voted to split these 2 lines items into separate voted - Unanimously Support
  2. Vote only external funding for Board gifts - Unanimously supported
  3. Vote to remove Sticker budget and put back into budget for external gifts - Unanimously supported
  4. Vote to remove Nathan from Board Roster

2021 Annual Budget

Description	Frequency	Total	Paid for By
Name.com domain registration (scikit-yb.org)	annually	$17.98	Ben
Read the Docs Gold Membership	$10 monthly	$120.00	Ben
Yellowbrick Conbtributor gifts	annually	$133.50	Kristen
		$271.48	

- Advisors pay their share (dues) to Kristen via Venmo. Kristen sent her Venmo handle and QR Code via the group's slack channel. Dues are payable by March 1, 2021.
- Since we are reaching out to potential new board members, we will delay the collection due date since we do not the final total number of board members and everyone's retrospective dues.
- Budget for appreciation gifts to to contributors. We will have 2 buckets of money, one for gifts for contributors and one that is board-funded to get a gift for board members.

Description	Donation	POC
Kansas Labs	$375	Ben
District Data Labs	$375	Tony
	$750	

Yellowbrick v1.3 Status Updates/Milestone Planning

Status Update: The issues that are part of this milestone can be found [here:] (https://github.com/DistrictDataLabs/yellowbrick/milestone/16)

Milestone planning:

- We need to ensure 0.24 Scikit-Learn Compatibility (Scipy 1.6 issues) * to help deal with this * pip sklearn dependency 0.23 (current)
- We need to research PEP517 and how to implement "pip install -e ." See how python is now dealing with python packaging.

Ideas for next Administrative Projects:

1.) Release a User Survey on Twitter

2.) Content Marketing through Twitter

3.) Prema to review backlog

Member Topics

- Kristen suggested exploring incorporating pip dependency resolver:In its January release (21.0), pip will use the new dependency resolver by default. The documentation gives a good overview of the new changes and guidance on how to respond to the new ResolutionImpossible error message.
- Kristen recommended replacing the iris dataset with other datasets in ours documentation.
- Changes to sklearn.utils for Sklearn Private/Public addressed in API [PR 1138] (https://github.com/DistrictDataLabs/yellowbrick/pull/1138)
- We decided to pass on participating in Google Summer of Code (GSOC)
- There was discussion of adding new board members such as Michael Garod, Molly and Matt Harrison.
- Semester focus on *Marketing & Outstanding Issues & PyDistrict*
- Try to get users to tell us how they're using YB such as COVID research and viz
- User Survey
- Board unanimously support to add new members. Adam and Prema to reach out to Molly, Matt and Michael to ask them to join the board.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

y

INDEX

F

G

H

I

J

K

L

M

N

P

Q

R

S

T