yellowbrick Documentation

Sürüm 0.5

District Data Labs

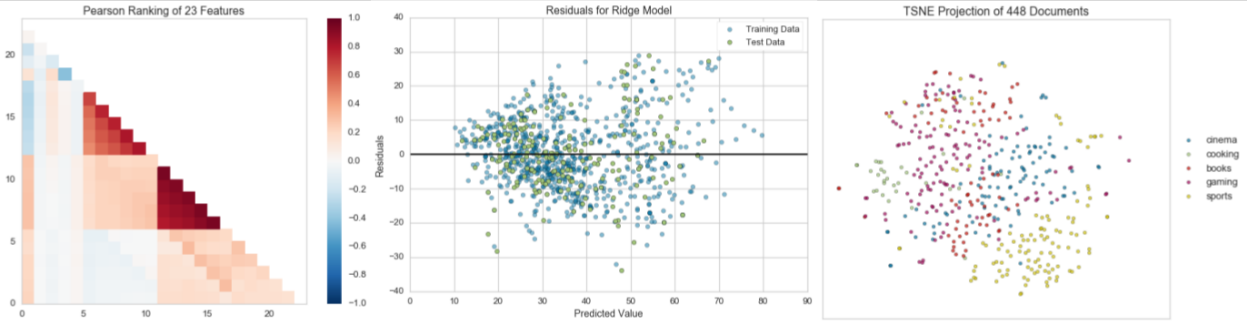
07 May 2018

İçindekiler

Yellowbrick'e hoşgeldiniz.

Şuanda Yellowbrick Türkçe dökümantasyonu üzerinde çalışmaktayız. Lütfen sayfamıza tekrar uğrayınız.

Ayrıca yardım tekliflerinize açığız. Türkçe tercüme için katkıda bulunmak isterseniz yellowbrick-docs-tr adresine pull request sorgusu gönderebilirsiniz. Eğer Yellowbrick için katkıda bulunmak isterseniz codebase adresine pull request sorgusu gönderebilirsiniz.



Yellowbrick, insanların model seçim sürecini yönlendirmeye izin veren "Görselleştirici" adı verilen Scikit-Learn API'sini genişleten görsel tanı araçları paketidir. Kısacası Yellowbrick, Scikit-Learn'i Matplotlib ile Scikit-Learn dökümantasyonuna gore birleştirerek modelinize göre görselleştirme üretmektedir. Yellowbrick ile ilgili daha fazlası için, lütfen bakınız: *Hakkında*.

Eğer Yellowbrick'te yeniyseniz, *Hızlı Başlangıç* ya da bu kısmı geçerek *Model Seçim Eğitseli* kısmını ziyaret edebilirsiniz. Yellowbrick, düzenli olarak eklenen birçok görselleştiricisi ile zengin bir kütüphanedir. Spesifik Görselleştiriciler ve genişletilmiş kullanımları ile ilgili detayları *Görselleştiriciler ve API* kısmından bulabilirsiniz. Yellowbrick'e katkıda bulunmak ister misiniz? contributing guide sayfasına göz atabilirsiniz. Şayet kullanıcı testi için başvurduysanız, *Kullanıcı Testi Talimatları* sayfasına bakınız.(ve Teşekkür ederiz)

BÖLÜM 1

Görselleştiriciler

Görselleştiriciler, öncelikli amacı model seçim işlemine içgörü sağlamaya izin veren görselleştirmeleri oluşturan tahmin edicilerdir(veriden öğrenilen nesneler). Scikit-Learn'e göre, Görselleştiriciler; veri alanını görselleştirirken, dönüstürücülere benzeyebilir ya da "ModelCV" (e.g. RidgeCV, LassoCV) metodların calışmasında olduğu gibi, bir model tahmin edicisini sarabilmektedir. Yellowbrick'in öncelikli amacı, Scikit-Learn benzeri bir mantıksal API oluşturmaktır. En populer görselleştiricilerimizden bazıları şunlardır:

1.1 Özellik Görselleştirme

- *Rank Features*: ilişki saptaması için özelliklerin ikili olarak sıralanması
- *Parallel Coordinates*: özelliklerin yatay görselleştirilmesi
- *Radial Visualization*: örneklerin bir dairesel alanda ayrılması
- *PCA Projection*: ana bileşenlere göre örneklerin gösterimi
- *Feature Importances*: spesifik bir model için önem veya doğrusal katsayısına göre özellik sıralaması
- *Scatter and Joint Plots*: özellik seçimi ile doğrudan veri görselleştirimi

1.2 Klasifikasyon Görselleştirme

- *Class Balance*: sınıfların dagılımının modeli nasıl etkilediğinin gösterimi
- *Classification Report*: precision, recall, ve F1 görsel temsili
- *ROC/AUC Curves*: işlem karakteristik eğrisi ve eğri altında kalan alan
- *Confusion Matrices*: sınıf karar veriminin görsel açıklaması

1.3 Regresyon Görselleştirme

- *Prediction Error Plot*: hedef alanı boyunca oluşan model hatalarının bulunması
- *Residuals Plot*: eğitim ve test verisi rezidüellerindeki farkın gösterimi
- *Alpha Selection*: alfa değer seçiminin regülasyonu nasıl etkilediğinin gösterimi

1.4 Kümesel Görselleştirme

- *K-Elbow Plot*: elbow metodu ve çeşitli metriklerin kullanımı ile k seçimi
- *Silhouette Plot*: siluet katsayısı değerlerinin görselleştirimi ile k seçimi

1.5 Metin Görselleştirmesi

- *Term Frequency*: metin gövdesi içinde bulunan terimlerin dağılım sıklığının görselleştirimi
- *t-SNE Corpus Visualization*: proje dökümanına stokastik yakınsal yerleştirimin kullanılması

... ve daha fazlası! Yeni görselleştiriciler sürekli olarak eklenmekte; örnekleri kontrol ettiğinizden emin olun (ya da hatta develop branch) ve yeni görselleştiricilerle ilgili fikirlerinizle katkıda bulunmaktan lütfen çekinmeyin.

BÖLÜM 2

Yardım İçin

Yellowbrick, Matplotlib ve Scikit-Learn geleneğinde olduğu gibi kapsamlı ve herkesi davet eden bir projedir. Bu projelere benzer olarak Python Software Foundation Code of Conduct ölçütlerini takip etmeye çalışıyoruz. Lütfen yardıma ihtiyacınız olduğunda ya da herhangi bir katkıda bulunmak isterseniz veyahut bug bulursanız bizlere çekinmeden ulaşabilirsiniz.

Yellowbrick yardımı için ilk yol bu isteğinizi gönderi olarak Google Groups Listserv kısmında paylaşmanız. Topluluk üyelerinin katılım gösterebildiği ve üyelerin birbirlerine cevap verebildigi email liste/forumu olup, burada en hızlı şekilde yanıt alabilirsiniz. Lütfen gruba katılmayı düşünün, böylelikle siz de soruları cevaplayabilirsiniz. Ayrıca Stack Overflow da soru sorabilir ve sorularınızı "yellowbrick" olarak etiketleyebilirsiniz. Ya da Github üzerinde Issues kısmına ekleme yapabilirsiniz. Veya Twitter hesabımıza @DistrictDataLab tweet ya da direk mesaj atabilirsiniz.

BÖLÜM 3

Açık Kaynak

Yellowbrick lisansı açık kaynaklı bir Apache 2.0 lisansıdır. Yellowbrick çok aktif geliştirici topluluğuna sahip olup; lütfen sizde katılmayı ve katkıda bulunmayı düşünün!

Yellowbrick GitHub üzerinde bulunmaktadır. issues ve pull requests detaylarını bu linklerden takip edebilirsiniz.

BÖLÜM 4

İçindekiler Tablosu

Kütüphanenin bu versiyonu için olan Yellowbrick dökümantasyonunun tüm listesini aşağıda bulabilirsiniz:

4.1 Hızlı Başlangıç

Yellowbrick'te yeniyseniz, bu kılavuz başlamanıza ve makine öğrenimi iş akışınıza görselleştiricileri dahil etmenize yardımcı olacaktır. Fakat başlamadan önce, geliştirme ortamlarıyla ilgili dikkate almanız gereken birkaç not bulunmakta.

Yellowbrick'in iki temel bağlılığı bulunmaktadır: Scikit-Learn ve Matplotlib. Şayet bu Python paketleriniz yoksa, Yellowbrick'le birlikte kurulacaktır. Yellowbrick, Scikit-Learn 0.18 versiyonu veya üstü ve Matplotlib 2.0 versiyonu ve üstü ile en iyi çalıştığını dikkate alınız. Her iki paketin de derlenmesi için Windows gibi sistemler üzerinde derlenmesi zor olan bazı C kodlarına gereksinim duymaktadır. Şayet problem yaşıyorsanız Anaconda gibi bu paketleri de içeren bir Python dağıtımını deneyebilirsiniz.

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.

Not: Jupyter, Pandas, and other ancillary libraries like NLTK for text visualizers are not installed with Yellowbrick and must be installed separately.

4.1.1 Kurulum

Yellowbrick, Python 2.7 ve üst sürümleri ile uyumludur fakat Yellowbrick'in tüm işlevlerinden yararlanmak için Python 3.5 ve üst sürümlerinin kullanımı tercih edilmektedir. Yellowbrick'i kurmanın en kolay yolu, PyPI 'den Python'un tercih edilen paket kurulumcusu pip kullanımıdır.

```
$ 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 Windows or Anaconda, you can take advantage of the conda utility to install the Anaconda Yellowbrick package:

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

Uyarı: There is a known bug installing matplotlib on Linux with Anaconda. If you're having trouble 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 without jumping through some hoops.

4.1.2 Yellowbrick Kullanımı

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 `poof()` method, which does the magic!

For example, there are several visualizers that act as transformers, used to perform feature analysis prior to fitting a model. Here is an example to visualize a high dimensional data set with parallel coordinates:

```
from yellowbrick.features import ParallelCoordinates

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

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 `poof()` 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. 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.poof(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.

Not: Data input to Yellowbrick is identical to that of Scikit-Learn: a dataset, `X`, which is a two-dimensional matrix of shape `(n,m)` where `n` is the number of instances (rows) and `m` is the number of features (columns). The dataset `X` can be a Pandas DataFrame, a Numpy array, or even a Python list of lists. Optionally, a vector `y`, which represents the target variable (in supervised learning), can also be supplied as input. The target `y` must have length `n` – the same number of elements as rows in `X` and can be a Pandas Series, Numpy array, or 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()
visualizer = ClassificationReport(model)

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

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 `poof()` 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.

4.1.3 Açıklamalar

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.

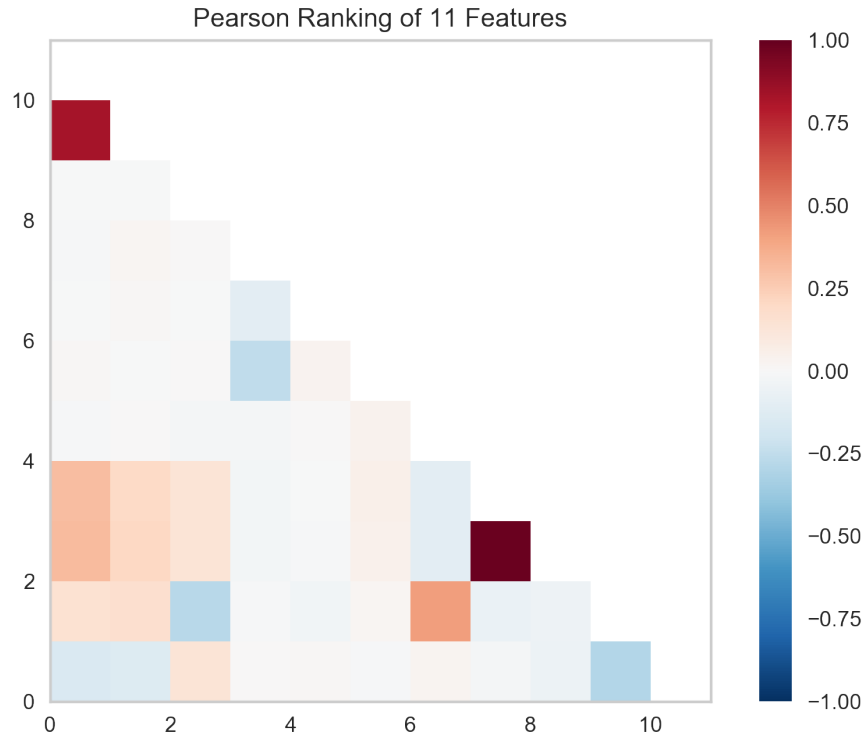
Not: We have updated the dataset from the UCI ML repository to make it a bit easier to load into Pandas; make sure you download the Yellowbrick version of the dataset.

After downloading the dataset and unzipping it in your current working directory, we can load our data as follows:

```
import pandas as pd

data = pd.read_csv('bikeshare.csv')
X = data[[
    "season", "month", "hour", "holiday", "weekday", "workingday",
    "weather", "temp", "feelslike", "humidity", "windspeed"
]]
y = data["riders"]
```

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 Rank2D 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.poof()
```

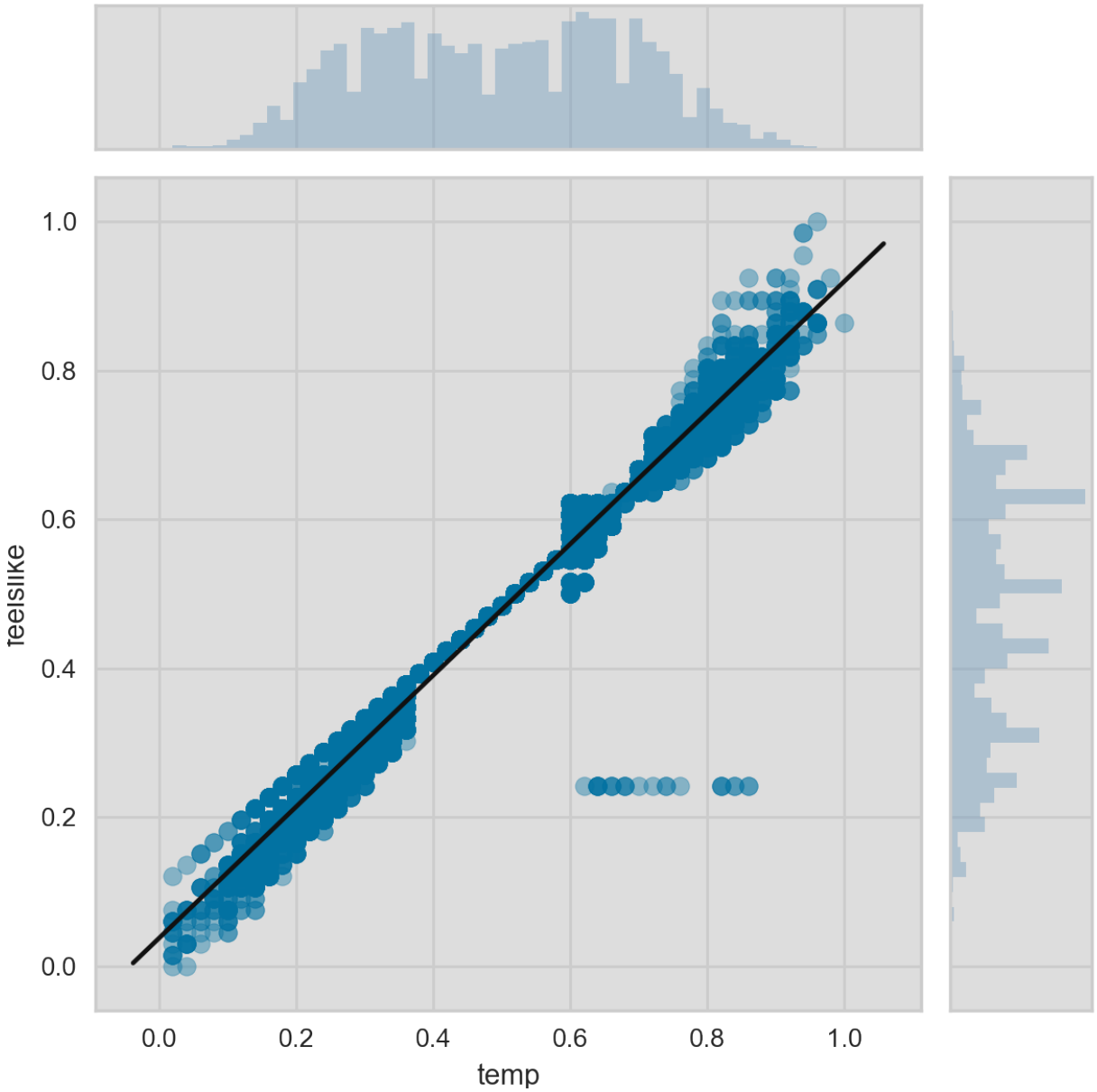


This figure shows us the Pearson correlation between pairs of features such that each cell in the grid represents two features identified in order on the x and y axes and whose color displays the magnitude of the correlation. A Pearson correlation of 1.0 means that there is a strong positive, linear 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 features 7 (temperature) and feature 9 (feelslike) have a strong correlation and also that feature

0 (season) has a strong correlation with feature 1 (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 JointPlotVisualizer to inspect those relationships.

```
from yellowbrick.features import JointPlotVisualizer

visualizer = JointPlotVisualizer(feature='temp', target='feelslike')
visualizer.fit(X['temp'], X['feelslike'])
visualizer.poof()
```



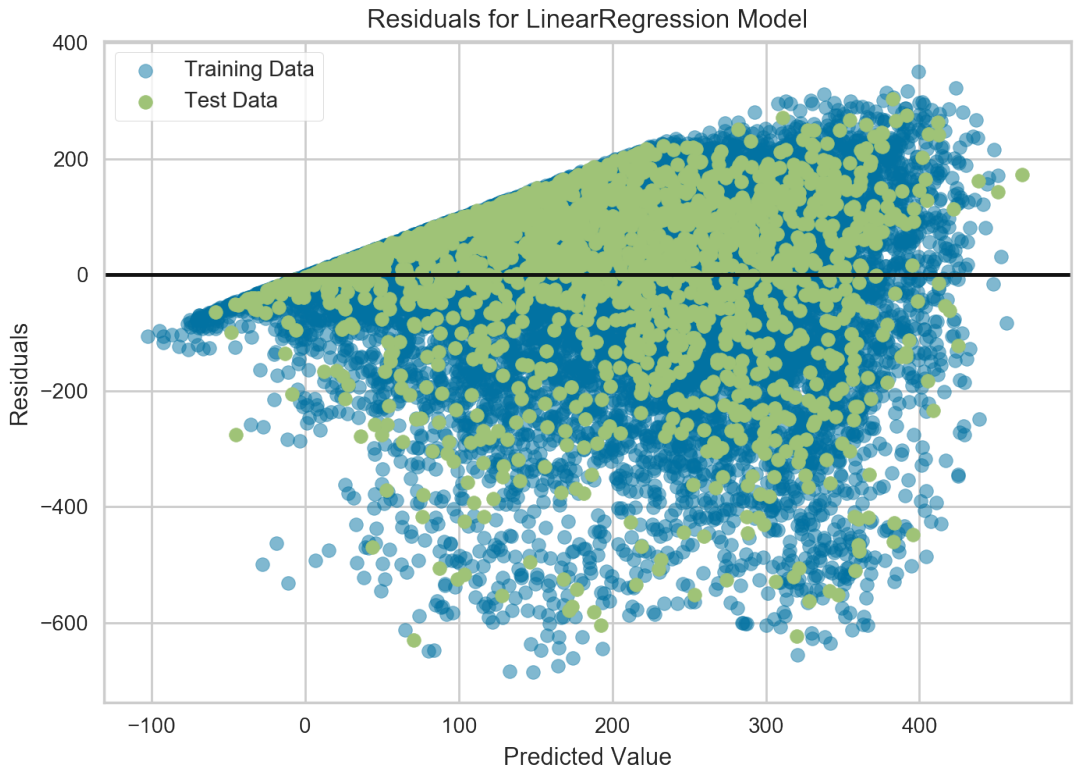
This visualizer plots a scatter diagram of the apparent temperature on the y axis and the actual measured temperature on the x axis and draws a line of best fit using a simple linear regression. Additionally, univariate distributions are shown as histograms above the x axis for temp and next to the y 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; first there appear to be some outliers in the dataset, where feelslike is approximately equal to 0.25. These instances may need to be manually removed in order to improve the quality of the final model because they could represent data input errors. Secondly, we can 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 appears to be. Moderate temperatures feel like they do. This gives us the intuition that feelslike may be a better feature than temp, and if it is causing problems in our regression analysis, we should probably remove the temp variable in favor of feels like.

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.poof()
```



The residuals plot shows the error against the predicted value, and allows us to look for heteroskedasticity 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 effected by the components of our model (namely the features).

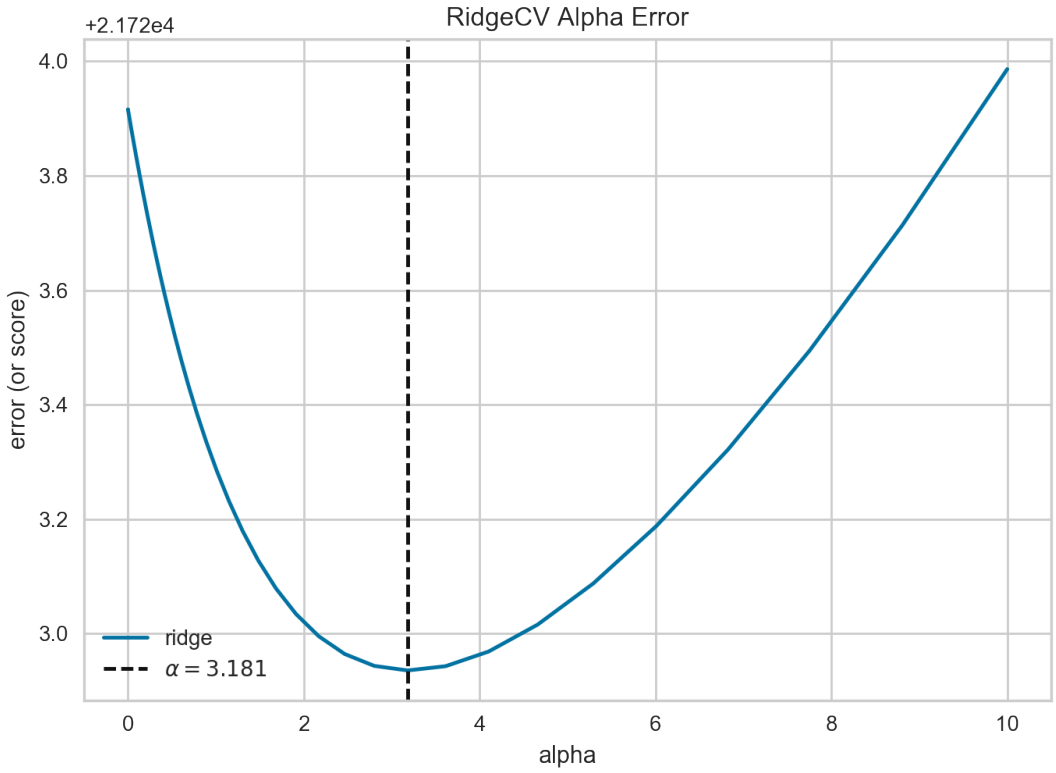
In this case, we can see that the lower the predicted value (the lower the number of riders), the lower the error, but the higher the number of predicted riders, the higher the error. This indicates that our model has more noise in certain regions of the target or that two variables are colinear, meaning 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 affect 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.

Because our coefficient of determination for this model is 0.328, 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.poof()
```



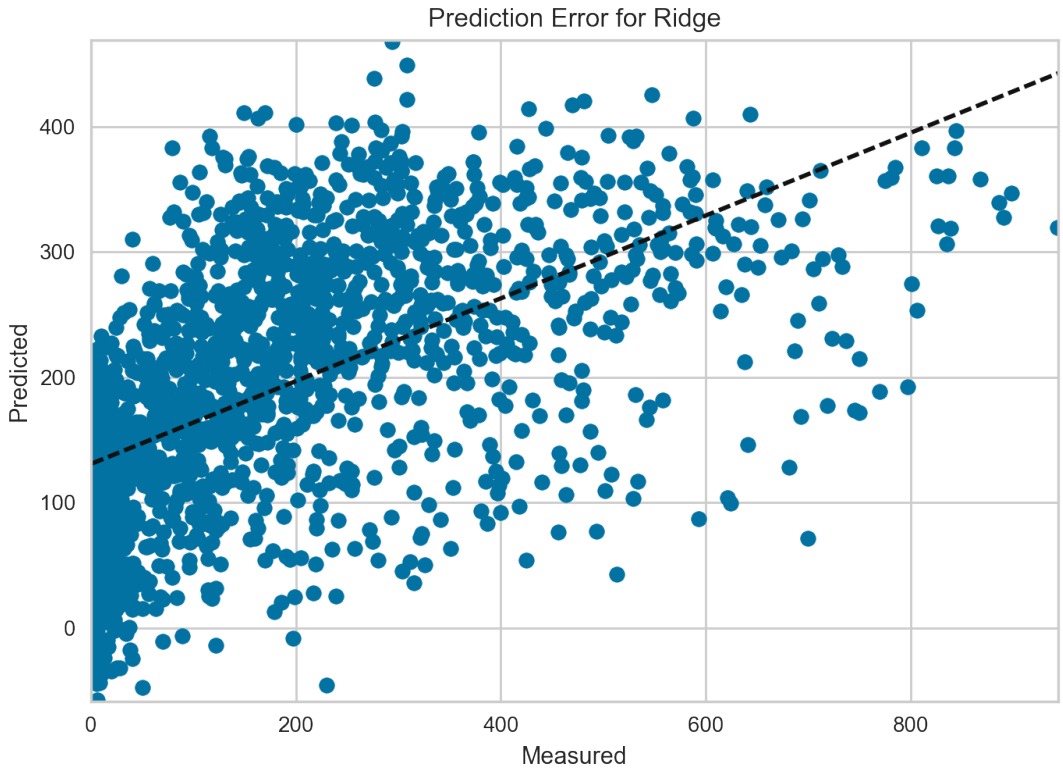
When exploring model families, the primary thing to consider is how the model becomes more *complex*. As the model increases in complexity, the error due to variance increases because the model is becoming more overfit and cannot generalize to unseen data. However, the simpler the model is the more error 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.poof()
```



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 Seçim Eğitseli*. After that you can get up to speed on specific visualizers detailed in the *Görselleştiriciler ve API*.

4.2 Model Seçim Eğitseli

Bu eğitselde, çeşitli Scikit-Learn modellerinin skorlarına bakacağız ve bunları Yellowbrick görsel tanı araçlarını kullanarak sırayla verilerimize göre en iyi modelin seçimi için karşılaştıracağız.

4.2.1 Model Seçim Üçlüsü

Makine öğrenimi tartışmaları sık sık model seçimi üzerine tekil odaklanma ile karakterize edilir. Gerek lojistik regresyon, karar ağaçları, Bayesian methodları veya yapay sinir ağları olsun; makine öğrenmesi uygulayıcıları tercihlerini genellikle hızlı bir şekilde açıklarlar. Bunun nedeni çoğunlukla tarihseldir. Modern üçüncü parti makine öğrenimi kütüphaneleri birçok modelin yayılmasını önemsiz olarak gösterse de, geleneksel olarak bu algoritmalardan birinin bile uygulaması ve ayarlanması yıllar süren çalışma gerektirmiştir. Sonuç olarak makine öğrenmesi uygulayıcıları diğerlerine göre daha belirgin (ve muhtemelen daha yaygın olan) algoritmaları daha çok tercih etmeye yönelmiştir.

Bununla birlikte, model seçimi basit şekilde "doğru" ya da "yanlış" algoritmayı seçmekten biraz daha nüanslıdır. Pratik olarak iş akışı şunları içermektedir:

1. en küçük ve en kestirici tahmin kümesi seçimi ya da oluşturumu
2. bir dizi algoritmaların bir model ailesinden seçimi ve
3. performans optimizesi için algoritma hiperparametlerinin ayarlanması

Kumar et al tarafından **model seçim üçlüsü** 2015 yılı SIGMOD makalesinde ilk defa tanımlanmıştır. Makale içerisinde, tahmin edici modelleme öngörüsü için inşaa edilen yeni nesil veritabanı sistemlerinin gelişimiyle ilgili olarak, makale yazarları pratikte makine öğreniminin büyük ölçüde deneysel yapısı sebebiyle bu tür sistemlere çok fazla ihtiyaç olduğunu ifade ederler. "Model seçimini," şu şekilde açıklarlar, "tekrarlayıcı ve keşifseldir çünkü [model seçim üçlüsü] alanı genellikle sonsuzdur ve analizçiler için yeterli doğruluk ve kavrayış sağlayabilecek bir olası [kombinasyon] bilmek genelde imkansızdır."

Son dönemlerde makine öğrenimi iş akışının büyük bir kısmı; grid search yöntemi, standartlaştırılmıs API ler ve GUI tabanlı uygulamalar yoluyla otomatize edilmiştir. Bununla birlikte, pratikte insan sezgisi ve rehberliği, kaliteli modeller üzerinde detaylı arama yöntemlerine göre daha efektif odaklanma sağlamaktadir. Görsel model seçim işlemi yoluyla, veri bilimcileri; hatalara ve yanılgılara düşmeden finale, açıklanabilir modellere doğru ilerleyiş gösterebilmektedir.

Yellowbrick kütüphanesi, makine öğrenimi için veri bilimcilerine model seçim sürecine yön vermelerine olanak sağlayan bir tanı görselleştirme platformudur. Yellowbrick, Scikit Learn API'sini yeni bir temel obje ile genişletmiştir: Görselleştirici. Görselleştiriciler, çok boyutlu verilerin dönüşümü sırasında görsel tanılar sunarak, Scikit-Learn işlem sürecinin bir parçası olarak görsel modellerin uymasını ve dönüşümünü sağlamaktadır.

4.2.2 Veri Hakkında

Bu eğitselde UCI Machine Learning Repository adresinden alınan mantar veriseti nin düzenlenmiş versiyonu kullanılmaktadır. Amacımız, bir mantarın karakteristik özelliklerine göre zehirli ya da yenilebilir olup olmadığını tahmin etmektir.

Veri, Agaricus ve Lepiota ailesine ait 23 çeşit solungaçlı mantar türüyle ilgili varsayımsal örneklerin açıklamalarını içermektedir. Her tür, kesinlikle yenilebilir kesinlikle yenemez veya yenilebilirliği bilinmeyen ya da tavsiye edilmeyen olarak tanımlanmıştır. (bu son sınıf zehirli sınıfı ile birleştirilmiştir).

Dosyamız, "agaricus-lepiota.txt," 3 nominal değerli öznitelikleri ve 8124 mantar örneğinin hedef değerlerini içermektedir. (4208 yenilebilir, 3916 zehirli).

Haydi Pandas ile verimizi yükleyelim.

```
import os
import pandas as pd

names = [
    'class',
    'cap-shape',
    'cap-surface',
    'cap-color'
]

mushrooms = os.path.join('data','agaricus-lepiota.txt')
dataset   = pd.read_csv(mushrooms)
dataset.columns = names
dataset.head()
```

.	class	cap-shape	cap-surface	cap-color
0	edible	bell	smooth	white
1	poisonous	convex	scaly	white
2	edible	convex	smooth	gray
3	edible	convex	scaly	yellow
4	edible	bell	smooth	white

```
features = ['cap-shape', 'cap-surface', 'cap-color']
target   = ['class']

X = dataset[features]
y = dataset[target]
```

4.2.3 Özellik Çıkarımı

Verimiz, hedef de dahil olmak üzere kategoriktir. Makine öğrenmesi için bu değerleri sayısal değerlere çevirmemiz gerekecek. Sırasıyla veri setimizden bunu sağlamak amacıyla, veri setinde bulunan değerleri bir modele uygun birşeylere dönüştürmemiz için Scikit-Learn dönüştürücülerini kullanmamız gerekmektedir. Neyseki Scikit-Learn, kategorik etiketleri sayısal integer değerlerine çevirecek dönüştürücü sağlamaktadır sklearn.preprocessing.LabelEncoder. Maalesef tek seferde sadece bir vektörü dönüştürebiliriz, bu yüzden birden fazla sütuna sırayla uygulamamız için uyarlama yapmamız gerekiyor.

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```

(continues on next page)

(önceki sayfadan devam)

```
class EncodeCategorical(BaseEstimator, TransformerMixin):
    """
    Encodes a specified list of columns or all columns if None.
    """

    def __init__(self, columns=None):
        self.columns  = [col for col in columns]
        self.encoders = None

    def fit(self, data, target=None):
        """
        Expects a data frame with named columns to encode.
        """
        # Encode all columns if columns is None
        if self.columns is None:
            self.columns = data.columns

        # Fit a label encoder for each column in the data frame
        self.encoders = {
            column: LabelEncoder().fit(data[column])
            for column in self.columns
        }
        return self

    def transform(self, data):
        """
        Uses the encoders to transform a data frame.
        """
        output = data.copy()
        for column, encoder in self.encoders.items():
            output[column] = encoder.transform(data[column])

        return output
```

4.2.4 Modelleme ve Değerlendirme

Sınıflandırıcı Değerlendirmesi için Genel Metrikler

Precision gerçek olan pozitif sonuçların toplam sayısının, tüm pozitif çıkan sonuçların sayısına bölünmesidir. (ör. *Yenilebilir olarak tahmin ettiğimiz mantarların aslinda ne kadarı gerçekten yenilebilir*).

Recall gerçek olan pozitif sonuçların toplam sayısının, tüm pozitif çıkması gereken sonuçların sayısına bölünmesidir. (ör. *Zehirli mantarların ne kadarını kesin zehirli olarak tahmin edebildik*).

F1 score bir testin doğruluğunun ölçüsüdür. Bu skoru hesaplamak için testin hem kesinlik hem de hassasiyeti dikkate alınmaktadır. F1 skoru; en iyi değer 1 ve en kötü değer 0'a ulaştığı yerde, kesinlik ve hassasiyetin ağırlıklı ortalaması olarak da yorumlanabilir.

```
kesinlik = gerçek pozitifler / (gerçek pozitifler + yanlış pozitifler)

hassasiyet = gerçek pozitifler / (yanlış negatifler + gerçek pozitifler)

F1 skoru = 2 * ((kesinlik * hassasiyet) / (kesinlik + hassasiyet))
```

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

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

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

Şimdi bazı tahminleri yapabilmek için hazırız.

Birden fazla tahmin edicilerin değerlendirilmesi için bir yol oluşturalım – Öncelikle klasik sayısal skorları (daha sonra Yellowbrick kütüphanesinden bazı görsel tanı araçlarıyla karşılaştırma yapacağımız) kullanarak.

```
from sklearn.metrics import f1_score
from sklearn.pipeline import Pipeline


def model_selection(X, y, estimator):
    """
    Test various estimators.
    """
    y = LabelEncoder().fit_transform(y.values.ravel())
    model = Pipeline([
         ('label_encoding', EncodeCategorical(X.keys())),
         ('one_hot_encoder', OneHotEncoder()),
         ('estimator', estimator)
    ])

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

    expected  = y
    predicted = model.predict(X)

    # Compute and return the F1 score (the harmonic mean of precision and recall)
    return (f1_score(expected, predicted))
```

```
# Try them all!
from sklearn.svm import LinearSVC, NuSVC, SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegressionCV, LogisticRegression,␣
↪SGDClassifier
from sklearn.ensemble import BaggingClassifier, ExtraTreesClassifier,␣
↪RandomForestClassifier
```

```
model_selection(X, y, LinearSVC())
```

```
0.65846308387744845
```

```
model_selection(X, y, NuSVC())
```

```
0.63838842388991346
```

```
model_selection(X, y, SVC())
```

```
0.66251459711950167
```

```
model_selection(X, y, SGDClassifier())
```

```
0.69944182052382997
```

```
model_selection(X, y, KNeighborsClassifier())
```

```
0.65802139037433149
```

```
model_selection(X, y, LogisticRegressionCV())
```

```
0.65846308387744845
```

```
model_selection(X, y, LogisticRegression())
```

```
0.65812609897010799
```

```
model_selection(X, y, BaggingClassifier())
```

```
0.687643484132343
```

```
model_selection(X, y, ExtraTreesClassifier())
```

```
0.68713648045448383
```

```
model_selection(X, y, RandomForestClassifier())
```

```
0.69317131158367451
```

İlk Model Değerlendirmesi

Yukarıdaki F1 skorlarının sonuçlarını baz aldığınızda hangi model en iyi performansı göstermiştir?

4.2.5 Görsel Model Değerlendirmesi

Haydi şimdi model değerlendirme fonksiyonumuzu, Yellowbrick `ClassificationReport` sınıfını kullanmak için tekrar düzenleyelim, bir model görselleştiricisi; kesinlik, hassasiyet ve F1 skorlarını göstermektedir. Bu görsel model analiz aracı renk kodlu ısı haritasında olduğu gibi sayısal skorları, kolay yorumlama ve saptamaya destek amacıyla; özellikle kullanım durumumuzdaki amaca uygun (hayat kurtarıcı, dengeli) Tip I ve Tip II hata nüanslarını birleştirir.

Tip I hata (veya **"yanlış pozitif"**) mevcut olmayan bir etkiyi tespit eder. (ör. aslında yenilebilir bir mantarın zehirli olarak saptanması).

Tip II hata (veya **"yanlış negatif"**) mevcut olan bir etkiyi tespit edememektir. (ör. aslında zehirli bir mantarın yenilebilir olduğuna inanılması).

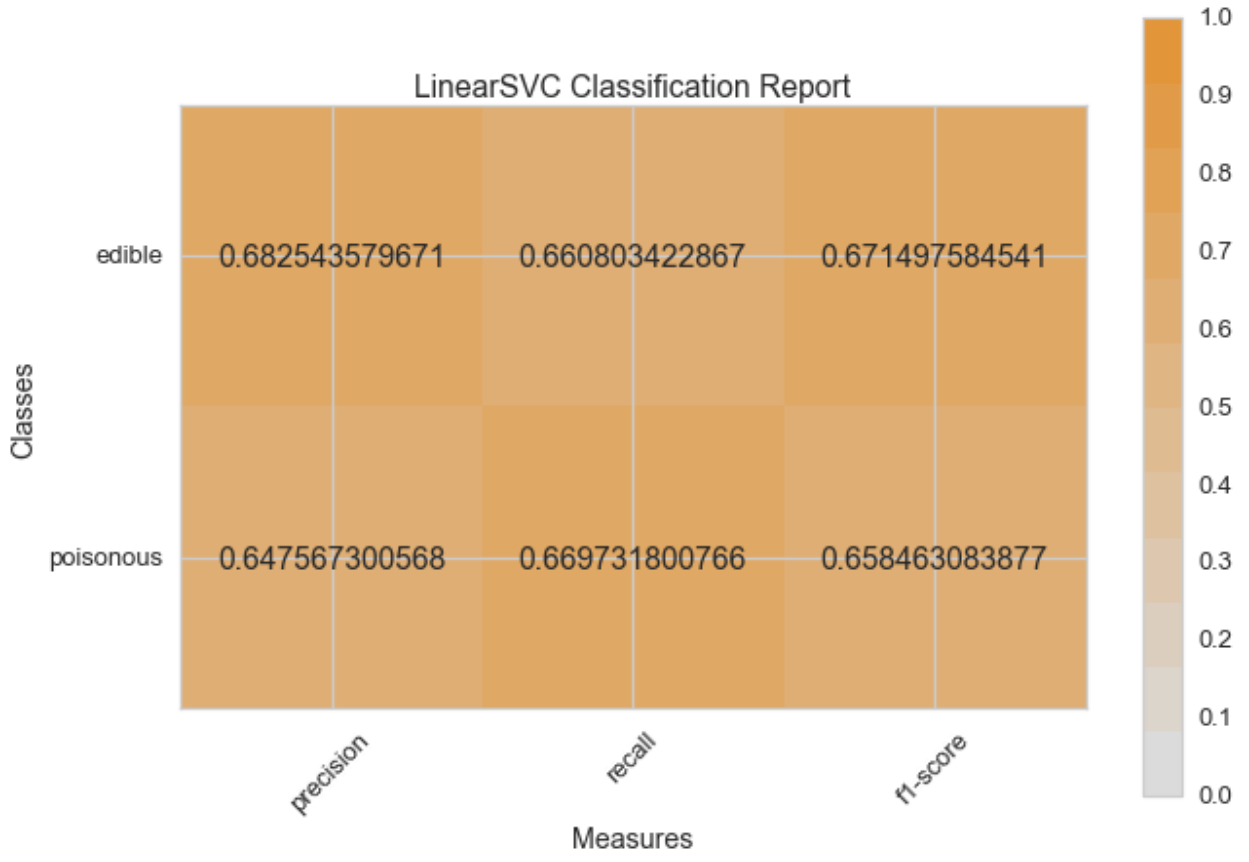
```
from sklearn.pipeline import Pipeline
from yellowbrick.classifier import ClassificationReport
```

(continues on next page)

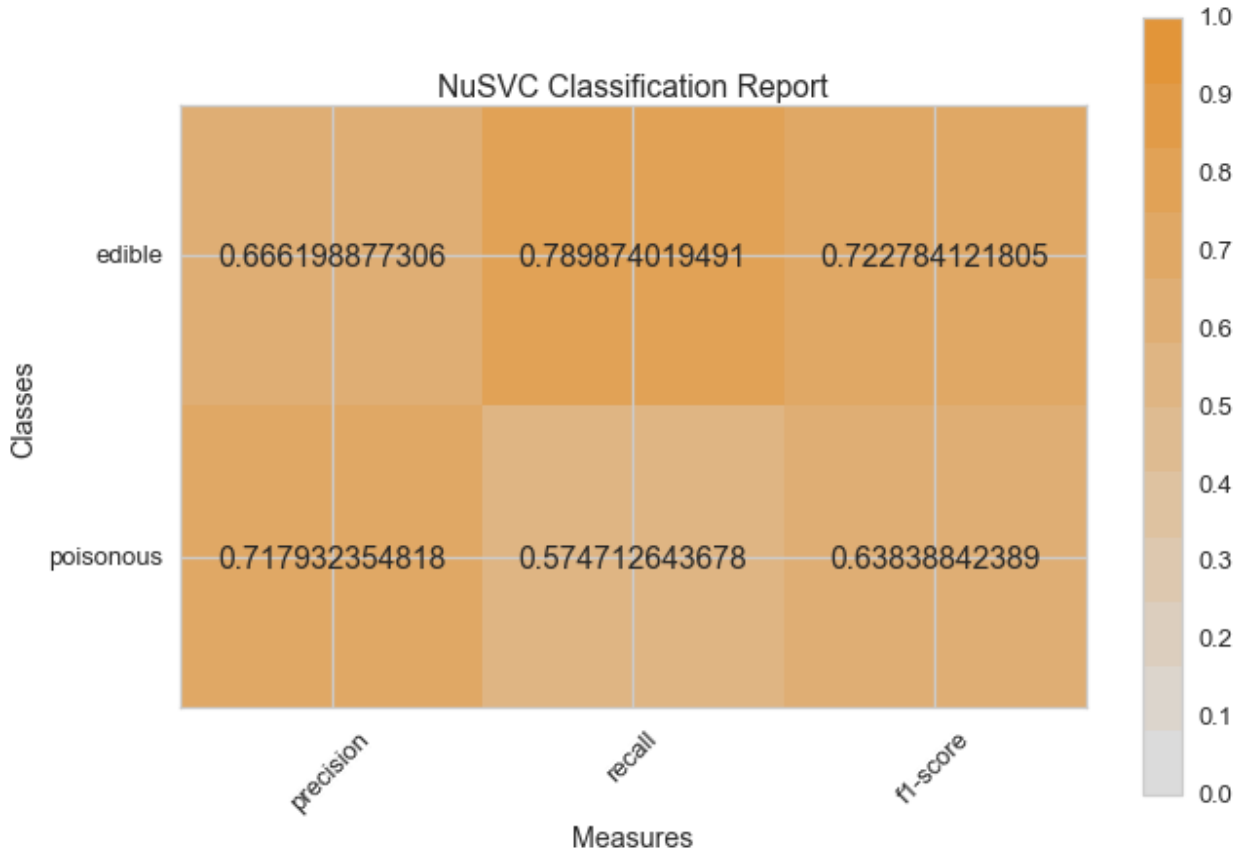
(önceki sayfadan devam)

```
def visual_model_selection(X, y, estimator):
    """
    Test various estimators.
    """
    y = LabelEncoder().fit_transform(y.values.ravel())
    model = Pipeline([
         ('label_encoding', EncodeCategorical(X.keys())),
         ('one_hot_encoder', OneHotEncoder()),
         ('estimator', estimator)
    ])

    # Instantiate the classification model and visualizer
    visualizer = ClassificationReport(model, classes=['edible', 'poisonous'])
    visualizer.fit(X, y)
    visualizer.score(X, y)
    visualizer.poof()
```

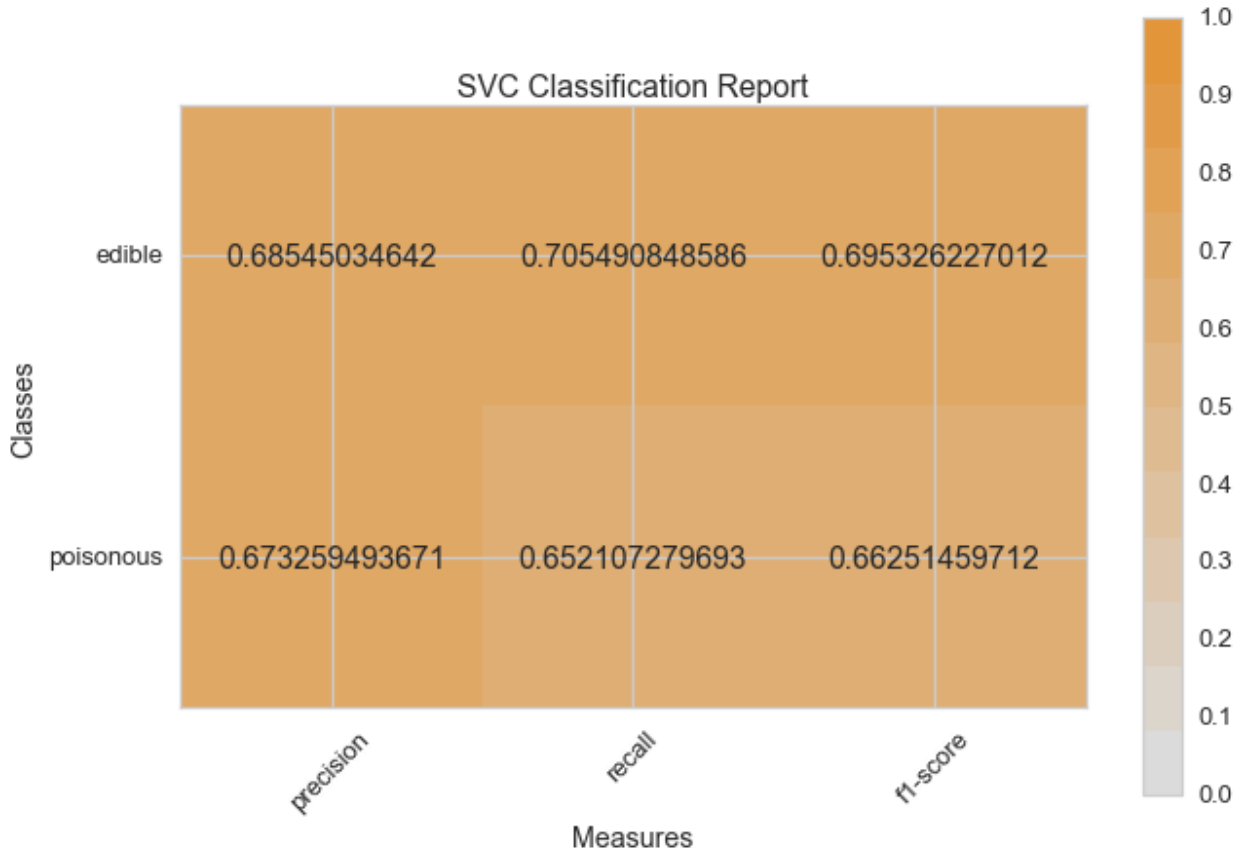
```
visual_model_selection(X, y, LinearSVC())
```



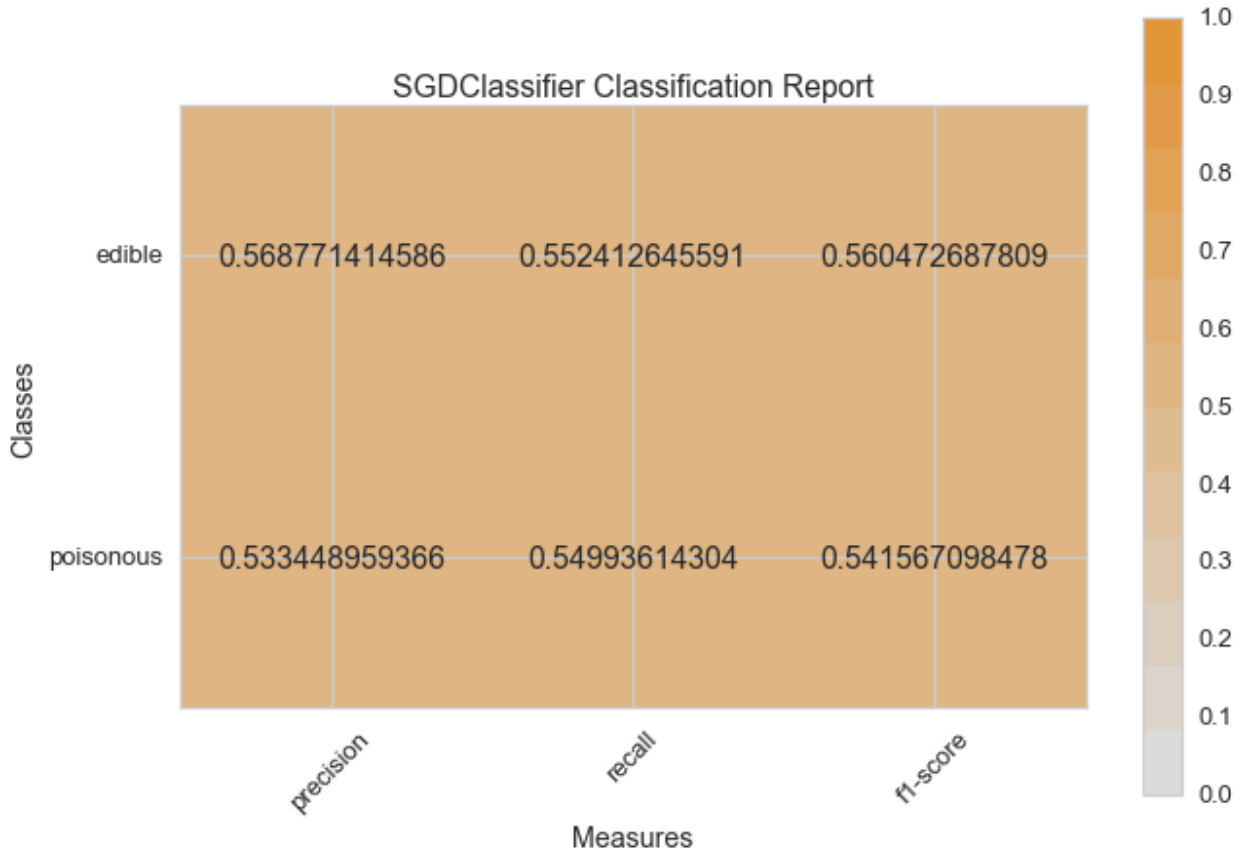

```
visual_model_selection(X, y, NuSVC())
```



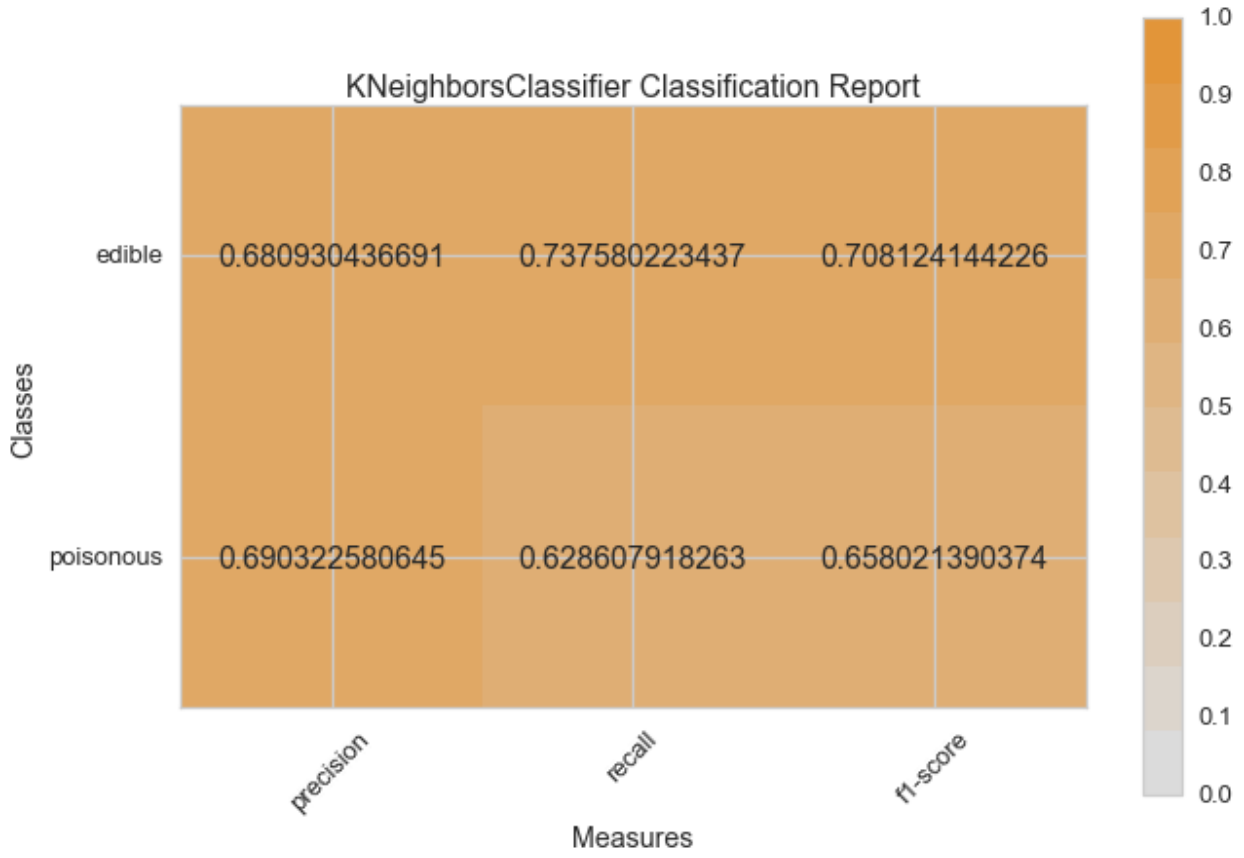

```
visual_model_selection(X, y, SVC())
```



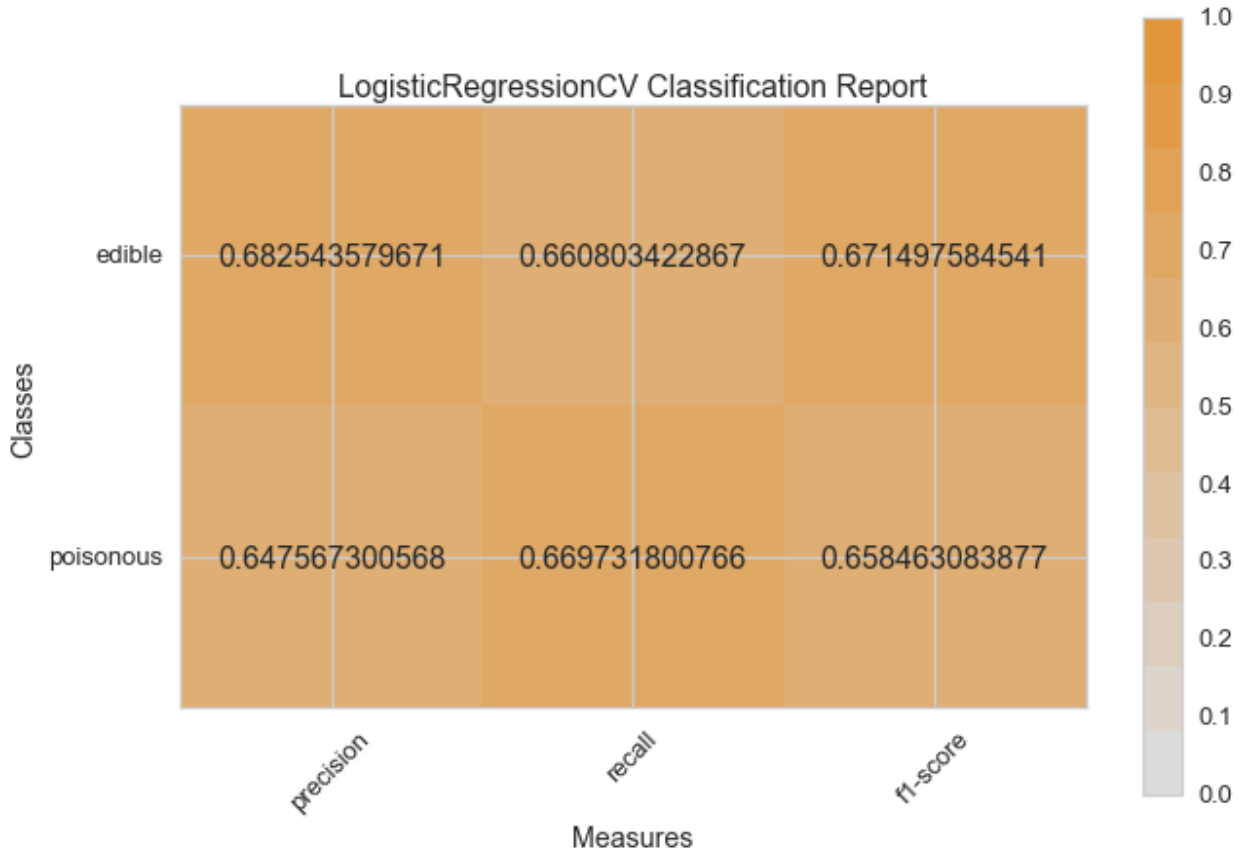

```
visual_model_selection(X, y, SGDClassifier())
```



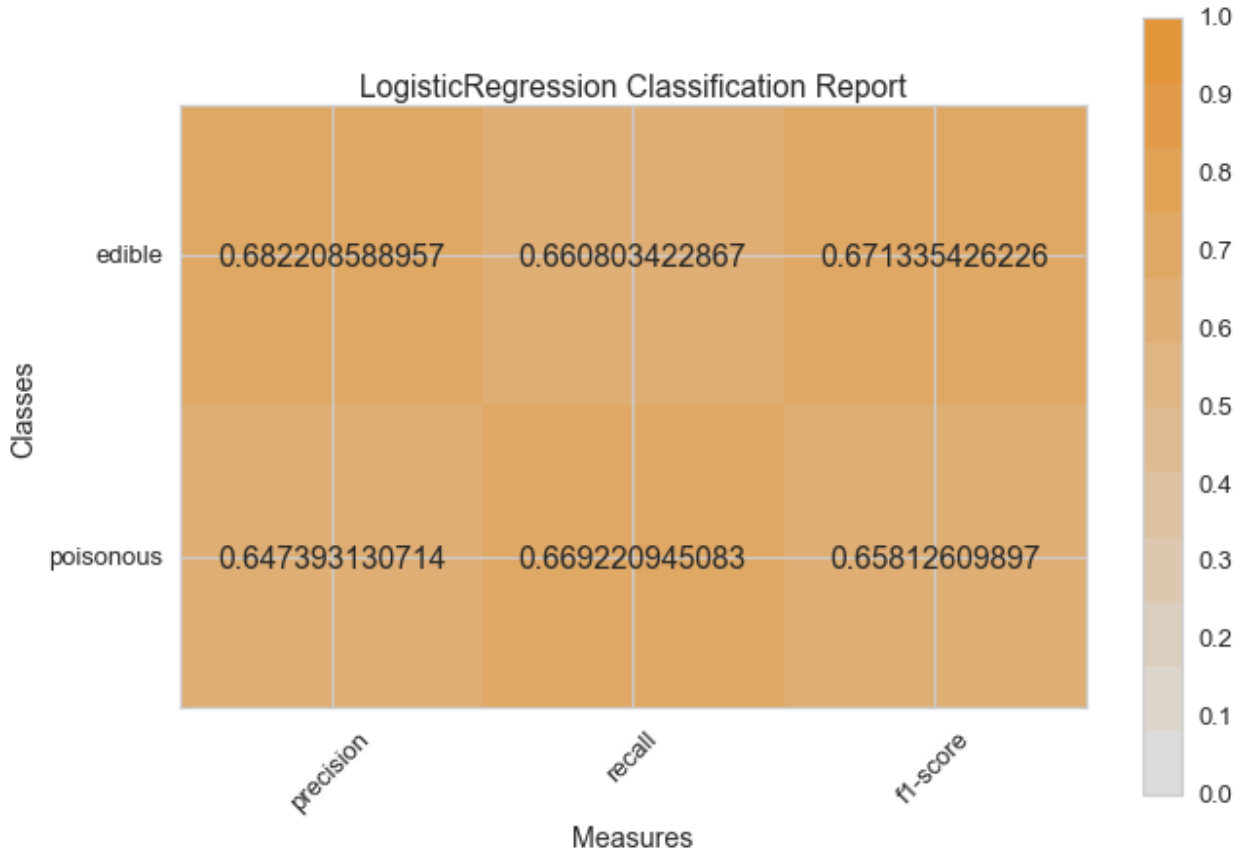

```
visual_model_selection(X, y, KNeighborsClassifier())
```



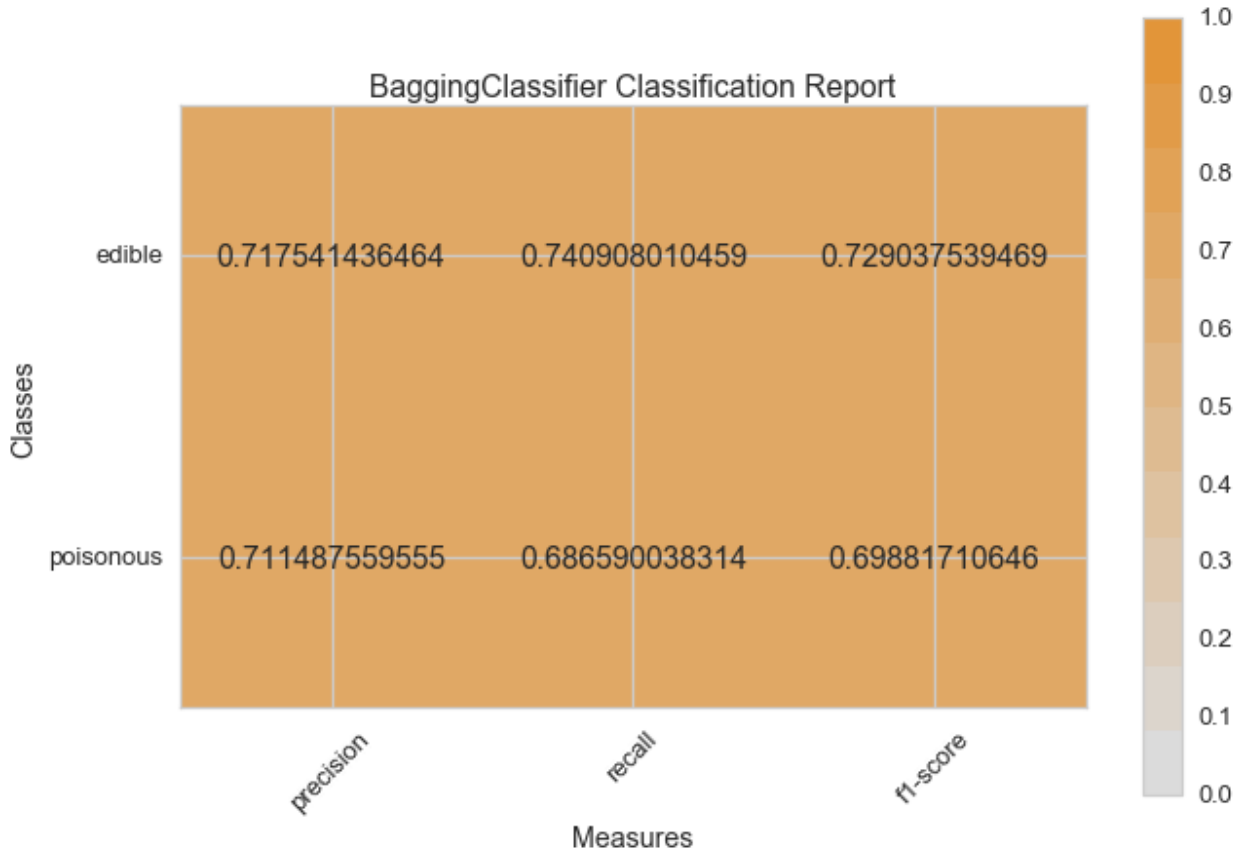

```
visual_model_selection(X, y, LogisticRegressionCV())
```



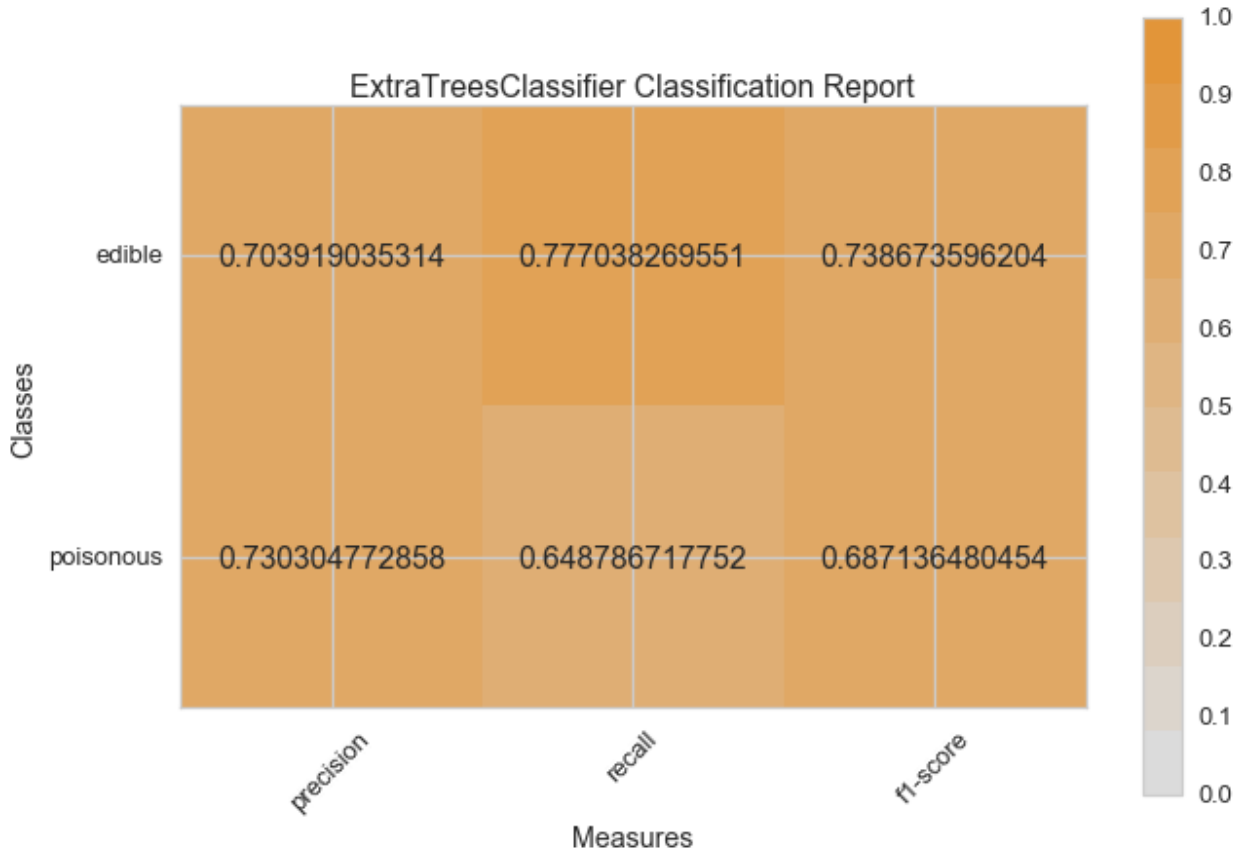

```
visual_model_selection(X, y, LogisticRegression())
```



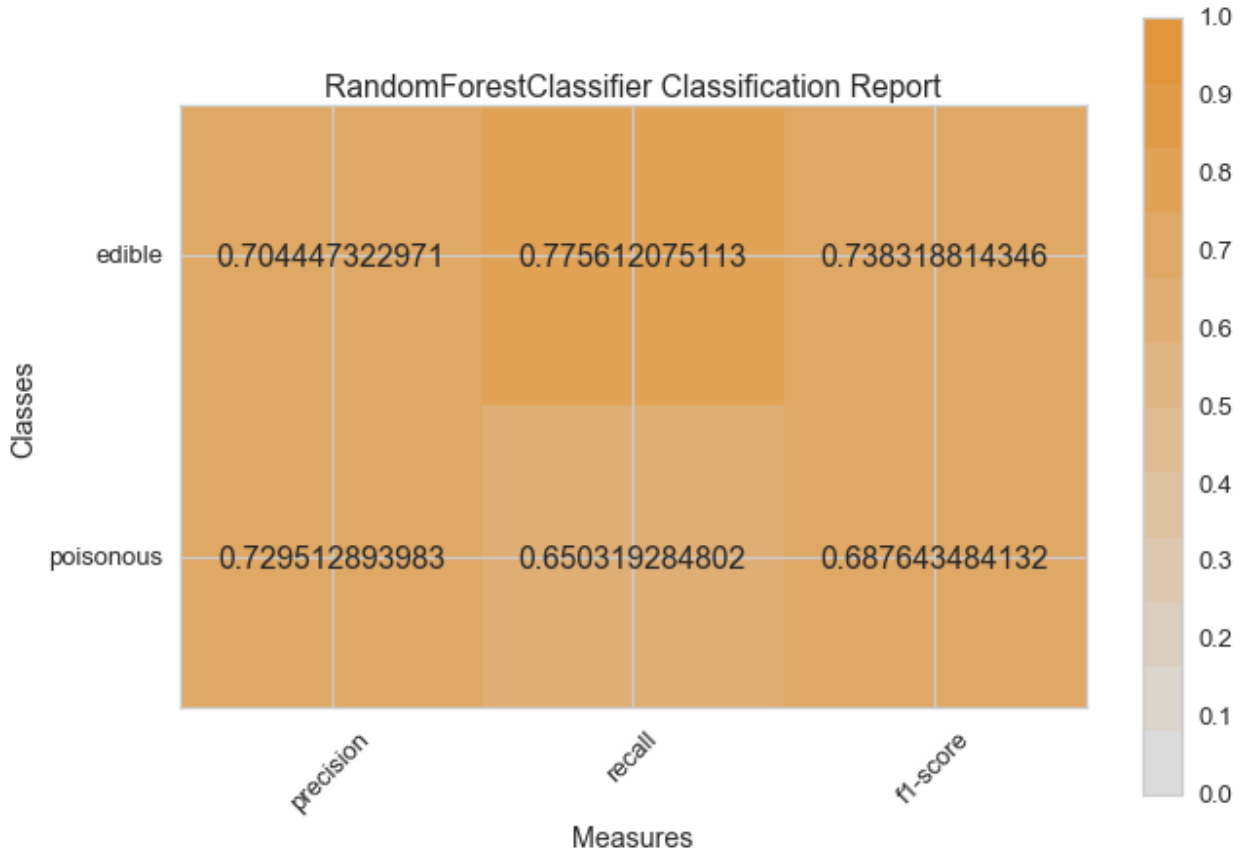

```
visual_model_selection(X, y, BaggingClassifier())
```




```
visual_model_selection(X, y, ExtraTreesClassifier())
```




```
visual_model_selection(X, y, RandomForestClassifier())
```



4.2.6 Değerlendirme

1. Hangi model en iyi gözükmektedir? Niçin?
2. Hangisi büyük ihtimalle hayatınızı kurtaracaktır?
3. Görsel model değerlendirme deneyiminin, sayısal model değerlendirmesinden farkı nedir?

4.3 Görselleştiriciler ve API

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

4.3.1 Örnek Veri Setleri

Yellowbrick hosts several datasets wrangled from the UCI Machine Learning Repository to present the examples in this section. If you haven't downloaded the data, you can do so by running:

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

This should create a folder called `data` in your current working directory with all of the datasets. You can load a specified dataset with `pandas.read_csv` as follows:

```
import pandas as pd

data = pd.read_csv('data/concrete/concrete.csv')
```

The following code snippet can be found at the top of the `examples/examples.ipynb` notebok in Yellowbrick. Please reference this code when trying to load a specific data set:

```
from yellowbrick.download import download_all

## The path to the test data sets
FIXTURES  = os.path.join(os.getcwd(), "data")

## Dataset loading mechanisms
datasets = {
    "bikeshare": os.path.join(FIXTURES, "bikeshare", "bikeshare.csv"),
    "concrete": os.path.join(FIXTURES, "concrete", "concrete.csv"),
    "credit": os.path.join(FIXTURES, "credit", "credit.csv"),
    "energy": os.path.join(FIXTURES, "energy", "energy.csv"),
    "game": os.path.join(FIXTURES, "game", "game.csv"),
    "mushroom": os.path.join(FIXTURES, "mushroom", "mushroom.csv"),
    "occupancy": os.path.join(FIXTURES, "occupancy", "occupancy.csv"),
}


def load_data(name, download=True):
    """
    Loads and wrangles the passed in dataset by name.
    If download is specified, this method will download any missing files.
    """

    # Get the path from the datasets
    path = datasets[name]

    # Check if the data exists, otherwise download or raise
    if not os.path.exists(path):
        if download:
            download_all()
        else:
            raise ValueError((
                "'{}' dataset has not been downloaded, "
                "use the download.py module to fetch datasets"
            ).format(name))


    # Return the data frame
    return pd.read_csv(path)
```

Note that most of the examples currently use one or more of the listed datasets for their examples (unless specifically shown otherwise). Each dataset has a `README.md` with detailed information about the data source, attributes, and target. Here is a complete listing of all datasets in Yellowbrick and their associated analytical tasks:

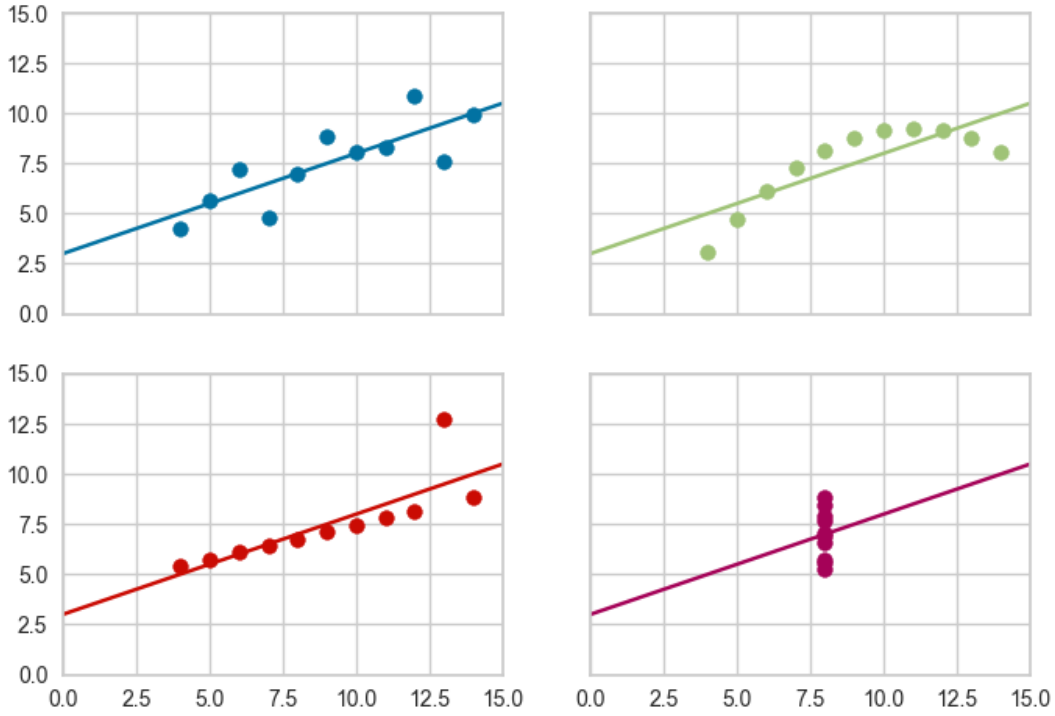
- **bikeshare**: suitable for regression
- **concrete**: suitable for regression
- **credit**: suitable for classification/clustering
- **energy**: suitable for regression
- **game**: suitable for classification

- **hobbies**: suitable for text analysis
- **mushroom**: suitable for classification/clustering
- **occupancy**: suitable for classification

4.3.2 Anscombe Dörtlüsü

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 Referansı

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.

4.3.3 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 five 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
- *Feature Importances*: rank features by relative importance in a model
- *Direct Data Visualization*: plot instances by selecting subsets of features

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 `poof` or `show` is called which 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.importances import FeatureImportances
from yellowbrick.features.scatter import ScatterVisualizer
```

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 scikit-learn Imputer.

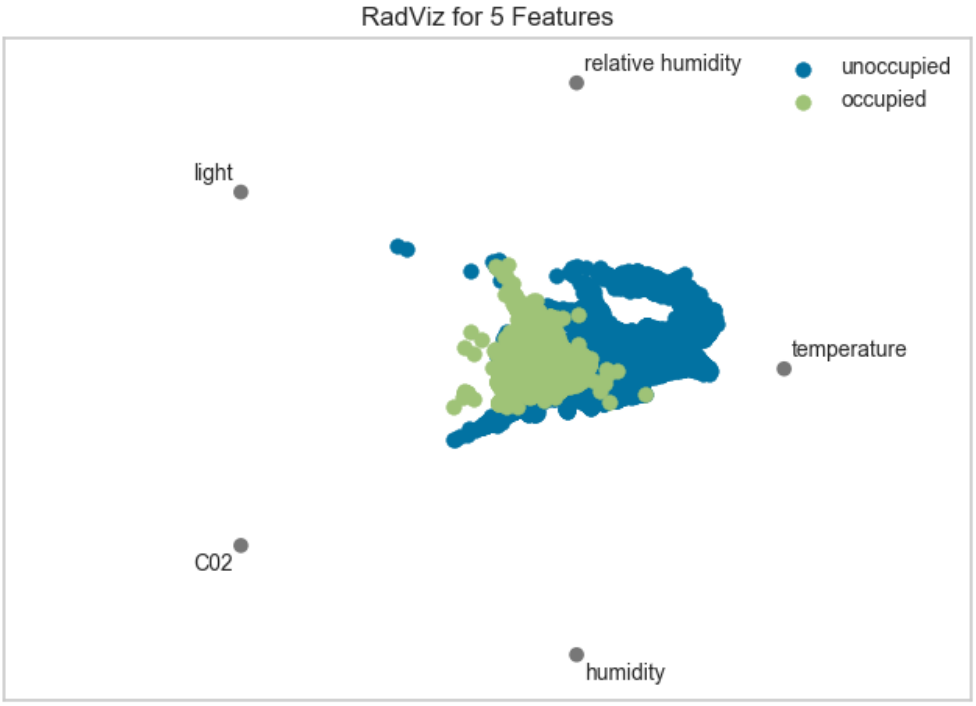
```
# Load the classification data set
data = load_data('occupancy')

# Specify the features of interest and the classes of the target
features = ["temperature", "relative humidity", "light", "C02", "humidity"]
```

(continues on next page)

(önceki sayfadan devam)

```
classes = ['unoccupied', 'occupied']

# Extract the numpy arrays from the data frame
X = data[features].as_matrix()
y = data.occupancy.as_matrix()
```

```
# Import the visualizer
from yellowbrick.features import RadViz

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

visualizer.fit(X, y)       # Fit the data to the visualizer
visualizer.transform(X)    # Transform the data
visualizer.poof()          # Draw/show/poof the data
```



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

API Reference

Implements radviz for feature analysis.

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

Bases: yellowbrick.features.base.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 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.

color [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.

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

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*)
Finalize executes any subclass-specific axes finalization steps. The user calls poof and poof calls finalize.

Parameters

kwargs: generic keyword arguments.

static normalize(*X*)
MinMax normalization to fit a matrix in the space [0,1] by column.

yellowbrick.features.radviz.**RadViz**
şunun takma adı: *yellowbrick.features.radviz.RadialVisualizer*

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.

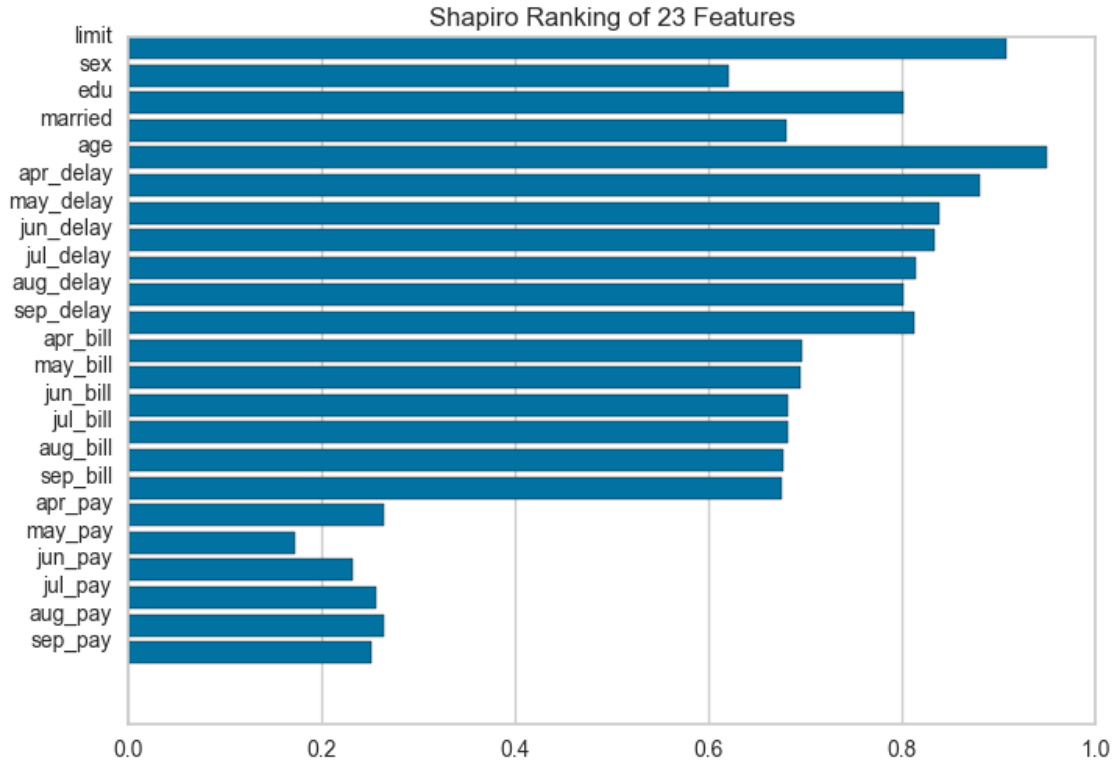
In this example, we'll use the credit default data set from the UCI Machine Learning repository to rank features. The code below creates our instance matrix and target vector.

```
# Load the dataset
data = load_data('credit')

# Specify the features of interest
features = [
        'limit', 'sex', 'edu', 'married', 'age', 'apr_delay', 'may_delay',
        'jun_delay', 'jul_delay', 'aug_delay', 'sep_delay', 'apr_bill', 'may_bill',
        'jun_bill', 'jul_bill', 'aug_bill', 'sep_bill', 'apr_pay', 'may_pay', 'jun_pay
↪',
        'jul_pay', 'aug_pay', 'sep_pay',
    ]

# Extract the numpy arrays from the data frame
X = data[features].as_matrix()
y = data.default.as_matrix()
```

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.

```
# Instantiate the 1D visualizer with the Sharpiro ranking algorithm
visualizer = Rank1D(features=features, algorithm='shapiro')

visualizer.fit(X, y)                # Fit the data to the visualizer
visualizer.transform(X)             # Transform the data
visualizer.poof()                   # Draw/show/poof the data
```

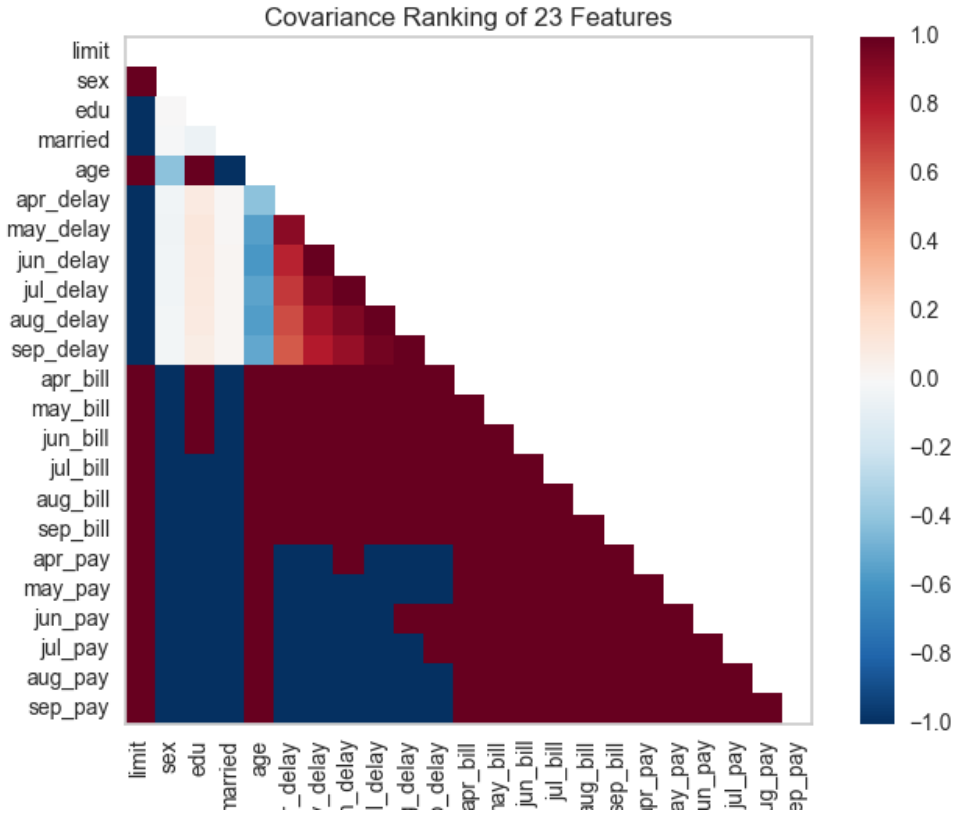


Rank 2D

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.

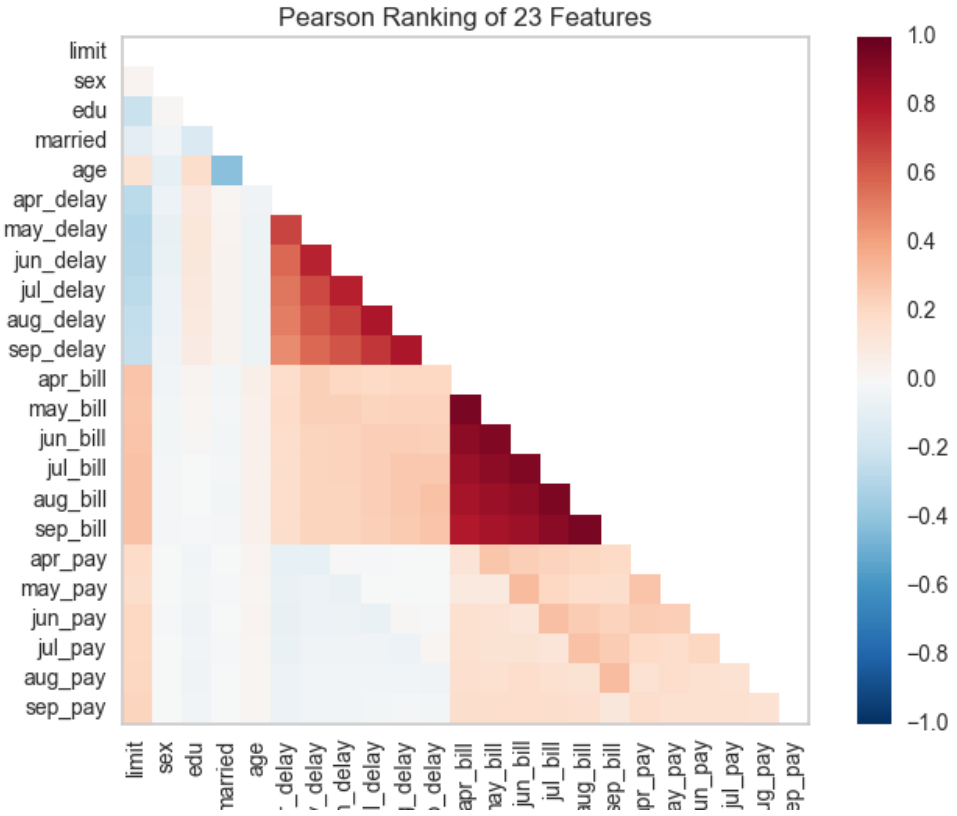
The default ranking algorithm is covariance, which attempts to compute the mean value of the product of deviations of variates from their respective means. Covariance loosely attempts to detect a colinear relationship between features.

```
# Instantiate the visualizer with the Covariance ranking algorithm
visualizer = Rank2D(features=features, algorithm='covariance')

visualizer.fit(X, y)                # Fit the data to the visualizer
visualizer.transform(X)             # Transform the data
visualizer.poof()                   # Draw/show/poof the data
```



Alternatively we can utilize a linear correlation algorithm such as a Pearson score to similarly detect colinear relationships. Compare the output from Pearson below to the covariance ranking above.

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

visualizer.fit(X, y)                # Fit the data to the visualizer
visualizer.transform(X)             # Transform the data
visualizer.poof()                   # Draw/show/poof the data
```



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*, ***kwargs*)

Bases: yellowbrick.features.rankd.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'] 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.

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.poof()
```

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> at 0x7f8f2c89b1e0>}

class yellowbrick.features.rankd.**Rank2D**(*ax=None*, *algorithm='pearson'*, *features=None*, *colormap='RdBu_r'*, *show_feature_names=True*, ***kwargs*)
Bases: yellowbrick.features.rankd.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 [one of {'pearson', 'covariance'}, default: 'pearson'] The ranking algorithm to use, default is Pearson correlation.

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.poof()
```

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> at 0x7f8f2c89b400>, 'pearso

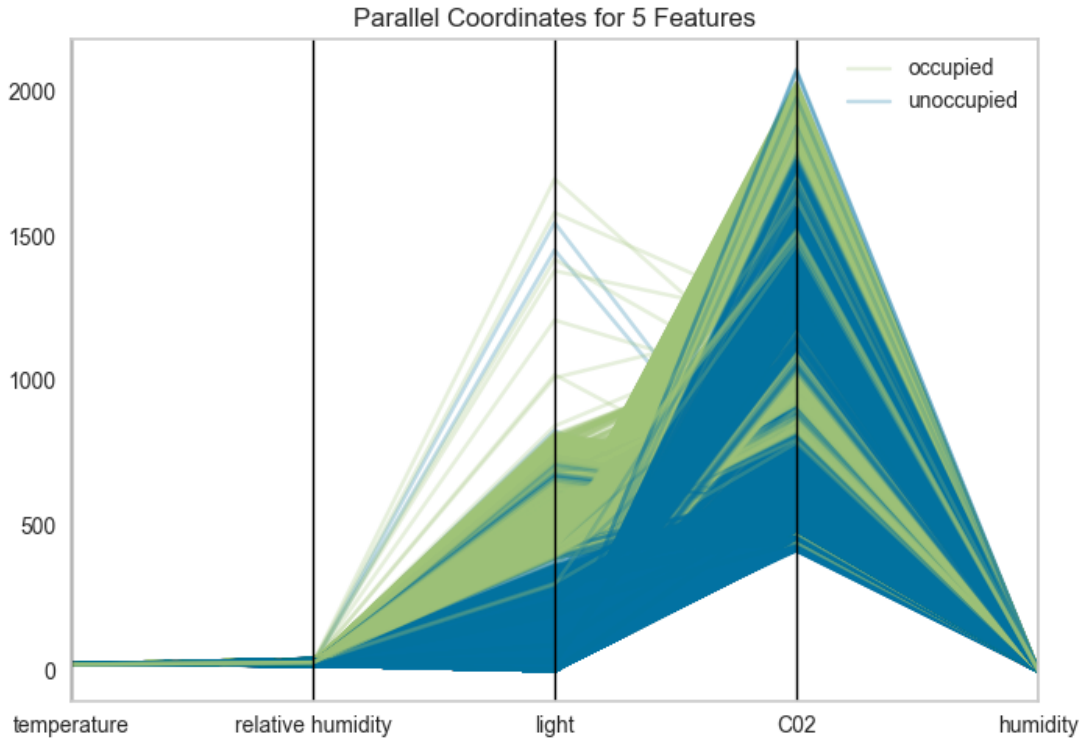
Parallel Coordinates

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 many dimensions; in fact given infinite horizontal space (e.g. a scrollbar), 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.

```
# Load the classification data set
data = load_data('occupancy')

# Specify the features of interest and the classes of the target
features = ["temperature", "relative humidity", "light", "C02", "humidity"]
classes = ['unoccupied', 'occupied']

# Extract the numpy arrays from the data frame
X = data[features].as_matrix()
y = data.occupancy.as_matrix()
```

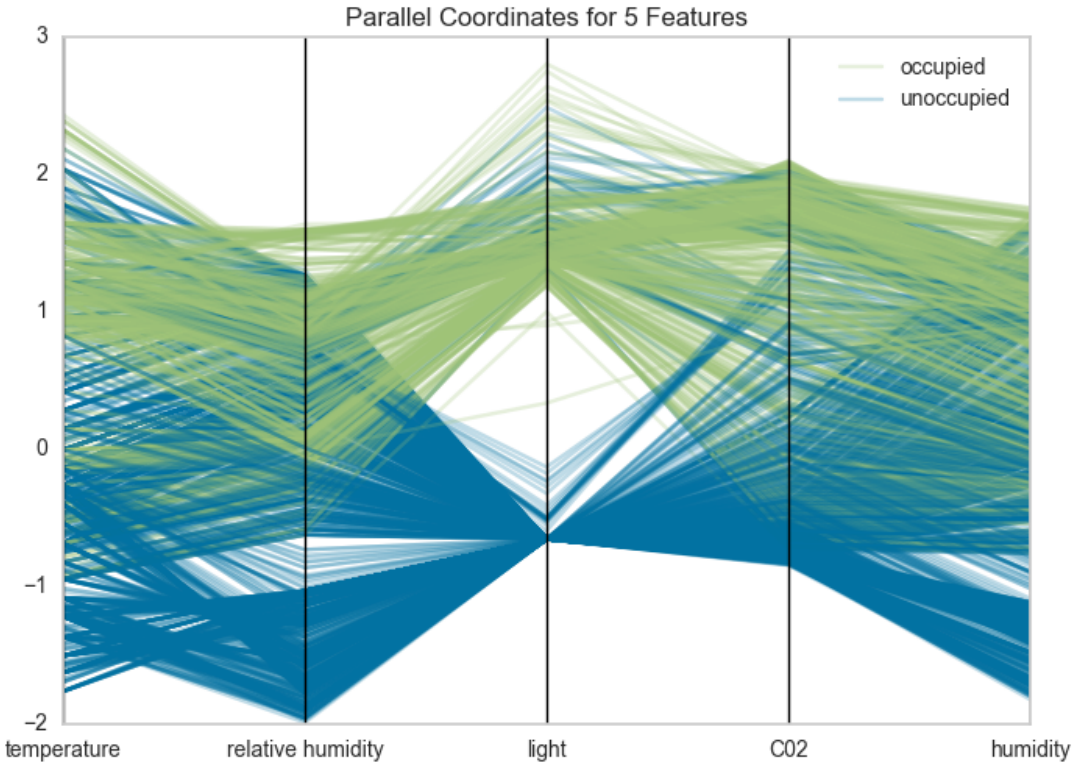
```
# Instantiate the visualizer
visualizer = ParallelCoordinates(classes=classes, features=features)

visualizer.fit(X, y)       # Fit the data to the visualizer
visualizer.transform(X)    # Transform the data
visualizer.poof()          # Draw/show/poof the data
```



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. To fix this, pass the `sample` keyword argument to the visualizer with a percentage to randomly sample from the dataset.

Additionally 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]`. A normalization methodology can be applied to change the range of features to `[0,1]`. Try using `minmax`, `minabs`, `standard`, `l1`, or `l2` normalization to change perspectives in the parallel coordinates:

```
# Instantiate the visualizer
visualizer = ParallelCoordinates(
    classes=classes, features=features,
    normalize='standard', sample=0.1,
)

visualizer.fit(X, y)       # Fit the data to the visualizer
visualizer.transform(X)    # Transform the data
visualizer.poof()          # Draw/show/poof the data
```



API Reference

Implementations of parallel coordinates for multi-dimensional feature analysis. There are a variety of parallel coordinates from Andrews Curves to coordinates that optimize column order.

class yellowbrick.features.pcoords.**ParallelCoordinates**(*ax=None*, *features=None*, *classes=None*, *normalize=None*, *sample=1.0*, *color=None*, *colormap=None*, *vlines=True*, *vlines_kwds=None*, ***kwargs*)

Bases: yellowbrick.features.base.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.

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 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.

color [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.

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.

Notes

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

Examples

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

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.

finalize(***kwargs*)
Finalize executes any subclass-specific axes finalization steps. The user calls poof and poof calls finalize.

Parameters

kwargs: generic keyword arguments.

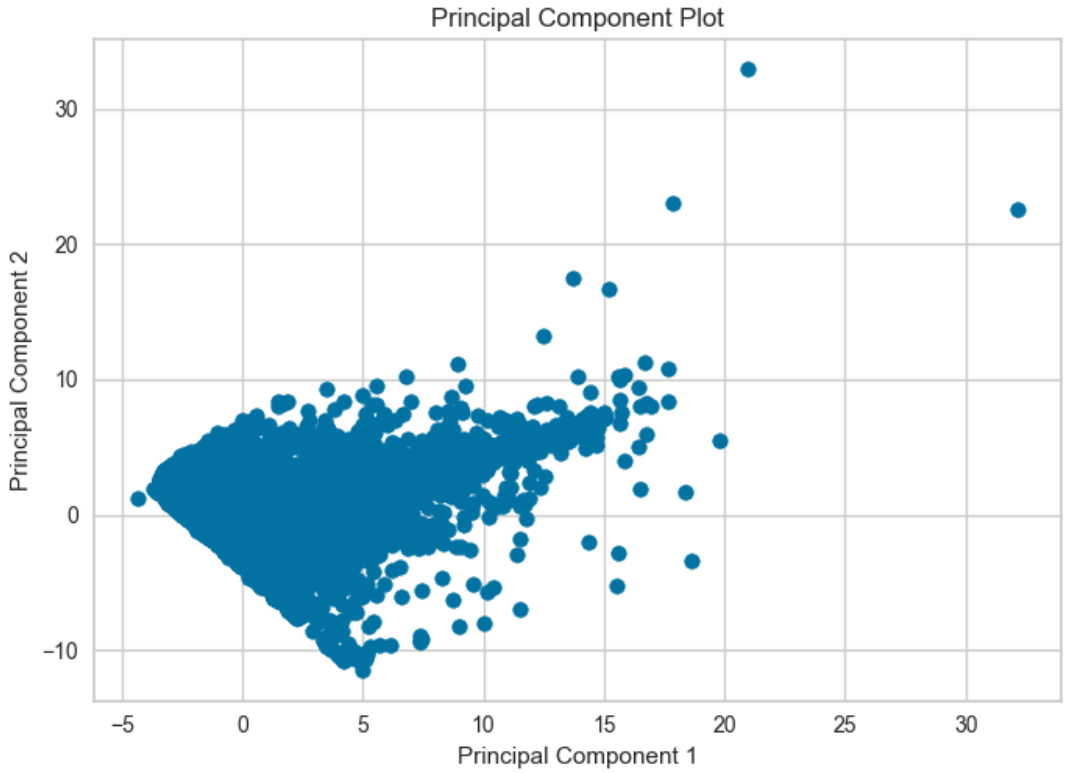
```
normalizers = {'l1': Normalizer(copy=True, norm='l1'), 'l2': Normalizer(copy=True, nor
```

PCA Projection

The PCA Decomposition visualizer utilizes principle 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 principle variation and can be interpreted to determine if spherical distance metrics can be utilized.

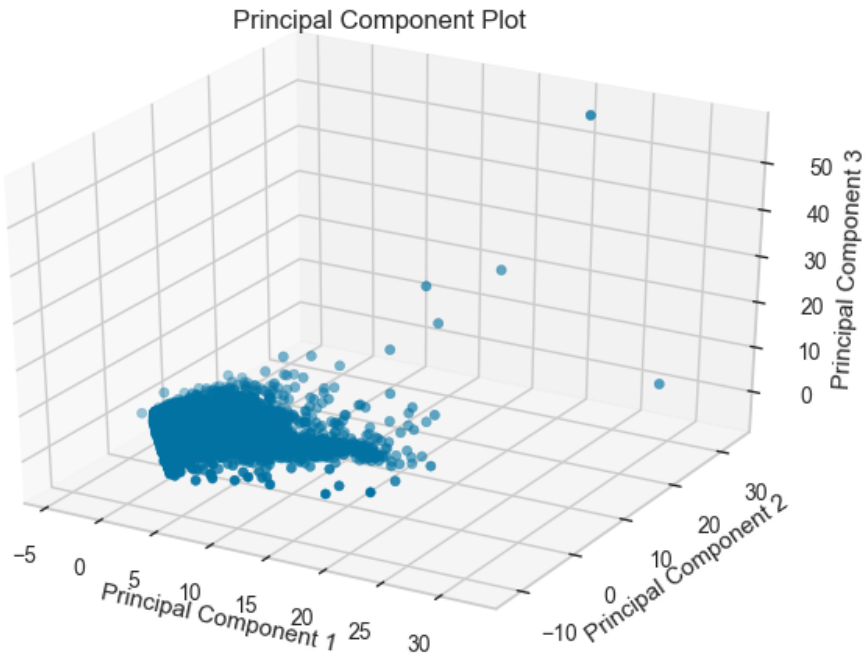
```
# Load the classification data set
data = load_data('credit')

# Specify the features of interest
features = [
    'limit', 'sex', 'edu', 'married', 'age', 'apr_delay', 'may_delay',
    'jun_delay', 'jul_delay', 'aug_delay', 'sep_delay', 'apr_bill', 'may_bill',
    'jun_bill', 'jul_bill', 'aug_bill', 'sep_bill', 'apr_pay', 'may_pay', 'jun_pay',
    'jul_pay', 'aug_pay', 'sep_pay',
]

# Extract the numpy arrays from the data frame
X = data[features].as_matrix()
y = data.default.as_matrix()
```

```
visualizer = PCADecomposition(scale=True, center=False, col=y)
visualizer.fit_transform(X,y)
visualizer.poof()
```



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

```
visualizer = PCADecomposition(scale=True, center=False, col=y, proj_dim=3)
visualizer.fit_transform(X,y)
visualizer.poof()
```



API Reference

Decomposition based feature visualization with PCA.

class yellowbrick.features.pca.**PCADecomposition**(*ax=None*, *scale=True*, *color=None*, *proj_dim=2*, *colormap='RdBu'*, ***kwargs*)

Bases: yellowbrick.features.base.FeatureVisualizer

Produce a two or three dimensional principal component plot of the data array `X` projected onto it's 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).

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

proj_dim [int, default: 2] Dimension of the PCA visualizer.

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

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.

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
>>> params = {'scale': True, 'center': False, 'col': y}
>>> visualizer = PCADecomposition(**params)
>>> visualizer.fit(X)
>>> visualizer.transform(X)
>>> visualizer.poof()
```

draw(***kwargs*)

The fitting or transformation process usually calls draw (not the user). This function is implemented for developers to hook into the matplotlib interface and to create an internal representation of the data the visualizer was trained on in the form of a figure or axes.

Parameters

kwargs: dict generic keyword arguments.

finalize(***kwargs*)

Finalize executes any subclass-specific axes finalization steps.

Parameters

kwargs: dict generic keyword arguments.

Notes

The user calls poof and poof calls finalize. Developers should implement visualizer-specific finalization methods like setting titles or axes labels, etc.

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

Fits a visualizer to data and is the primary entry point for producing a visualization. Visualizers are Scikit-Learn Estimator objects, which learn from data in order to produce a visual analysis or diagnostic. They can do this either by fitting features related data or by fitting an underlying model (or models) and visualizing their results.

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 drawing functionality or to the Scikit-Learn API. See visualizer specific details for how to use the kwargs to modify the visualization or fitting process.

Returns

self [visualizer] The fit method must always return self to support pipelines.

transform(*X*, *y=None*, ***kwargs*)
Primarily a pass-through to ensure that the feature visualizer will work in a pipeline setting. This method can also call drawing methods in order to ensure that the visualization is constructed.

This method must return a numpy array with the same shape as X.

Feature Importances

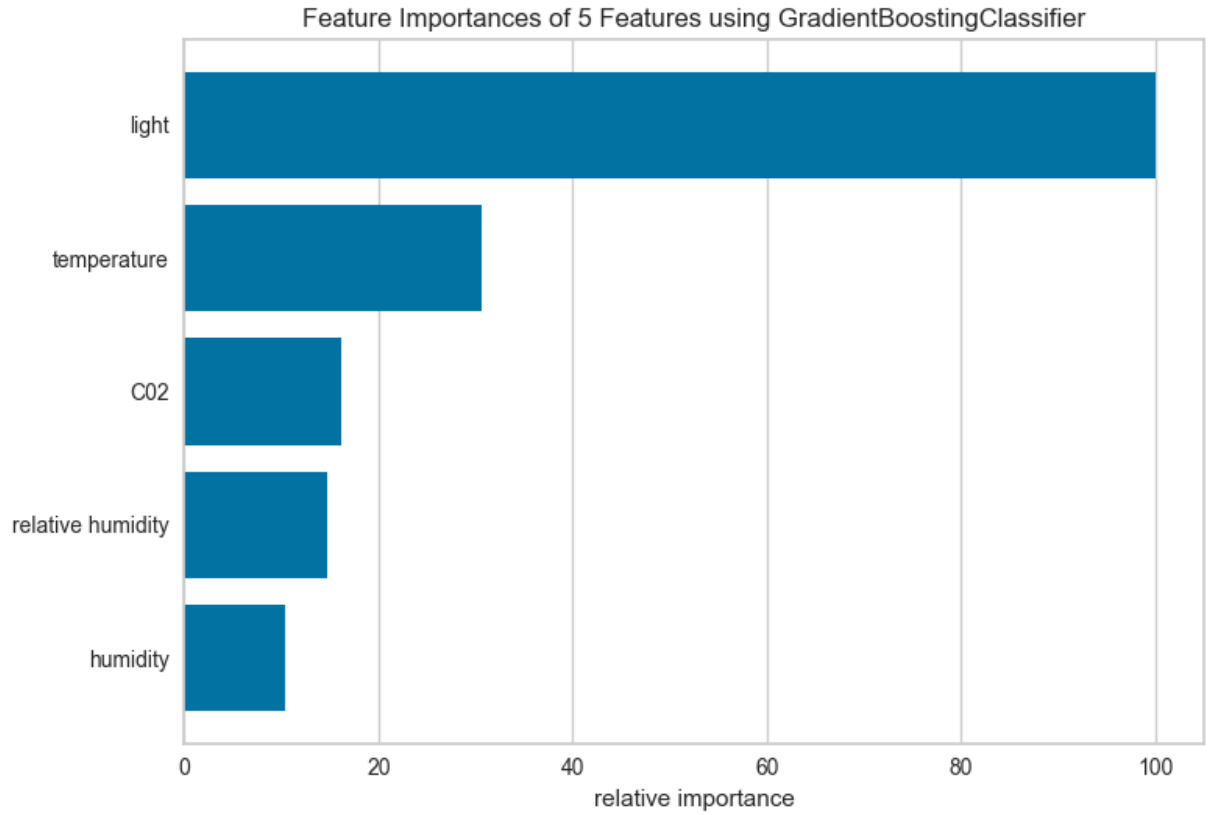
The feature engineering process involves selecting the *minimum* required features to produce a valid model because the more features a model contains, the more complex it is (and the more sparse the data), therefore the more sensitive the model is to errors due to variance. A common approach to eliminating 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. Let's start with an example; first load a classification dataset as follows:

```
# Load the classification data set
data = load_data('occupancy')

# Specify the features of interest
features = [
    "temperature", "relative humidity", "light", "C02", "humidity"
]

# Extract the instances and target
X = data[features]
y = data.occupancy
```

Once the dataset has been loaded, 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 GradientBoostingClassifier
from yellowbrick.features import FeatureImportances

# Create a new matplotlib figure
fig = plt.figure()
ax = fig.add_subplot()

viz = FeatureImportances(GradientBoostingClassifier(), ax=ax)
viz.fit(X, y)
viz.poof()
```



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. First we start by loading a regression dataset:

```
# Load a regression data set
data = load_data("concrete")

# Specify the features of interest
features = [
    'cement','slag','ash','water','splast','coarse','fine','age'
]

# Extract the instances and target
X = concrete[feats]
y = concrete.strength
```

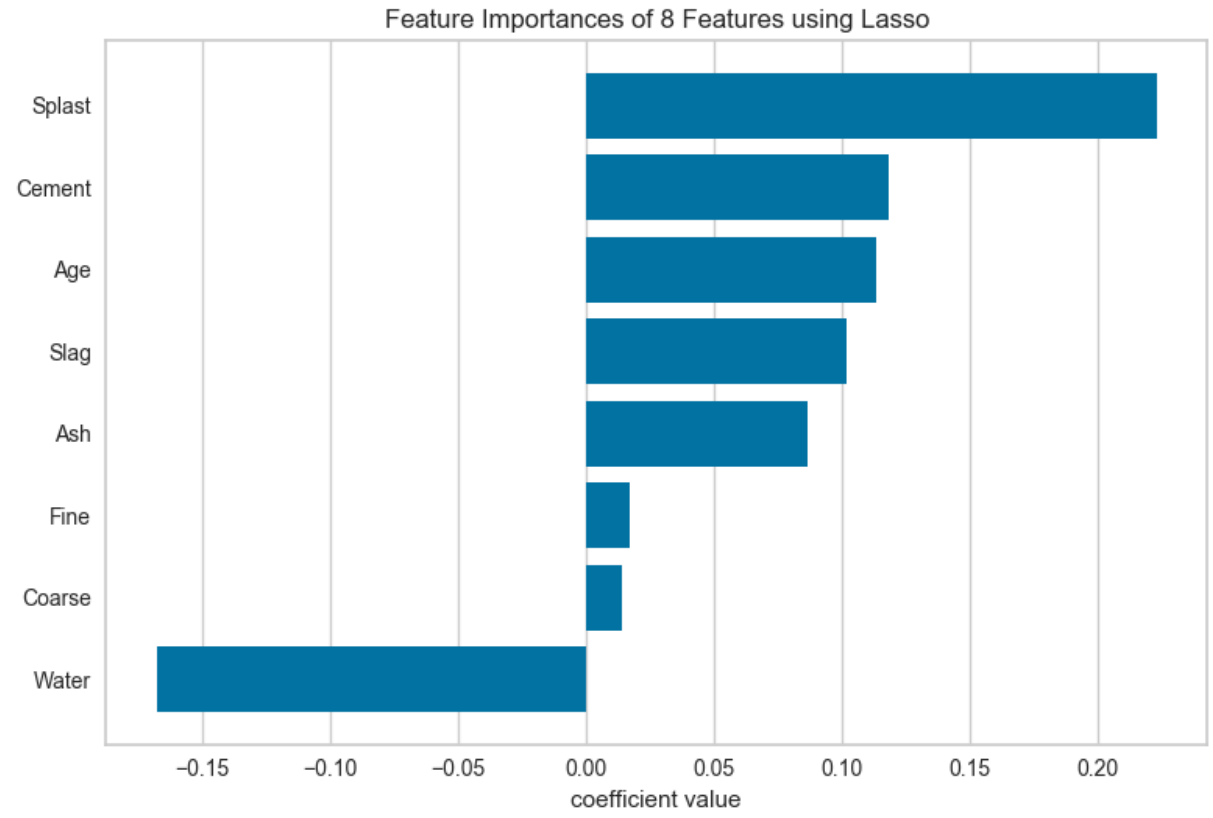
When using a model with a `coef_` attribute, it is better to set `relative=False` to draw the true magnitude of the coefficient (which may be negative). We can also specify our own set of labels if the dataset does not have column names or to print better titles. In the example below we title case our features for better readability:

```
# Create a new figure
fig = plt.figure()
ax = fig.add_subplot()
```

(continues on next page)

(önceki sayfadan devam)

```
# Title case the feature for better display and create the visualizer
labels = list(map(lambda s: s.title(), features))
viz = FeatureImportances(Lasso(), ax=ax, labels=labels, relative=False)

# Fit and show the feature importances
viz.fit(X, y)
viz.poof()
```



Not: The interpretation of the importance of coeficients depends on the model; see the discussion below for more details.

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.

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.features.importances.**FeatureImportances**(*model*, *ax=None*, *labels=None*, *relative=True*, *absolute=False*, *xlabel=None*, ***kwargs*)

Bases: yellowbrick.base.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.

Parameters

model [Estimator] A Scikit-Learn estimator that learns feature importances. Must support either coef_ or feature_importances_ 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, 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.

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.poof()
```

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

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.

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.

Scatter Visualization

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.

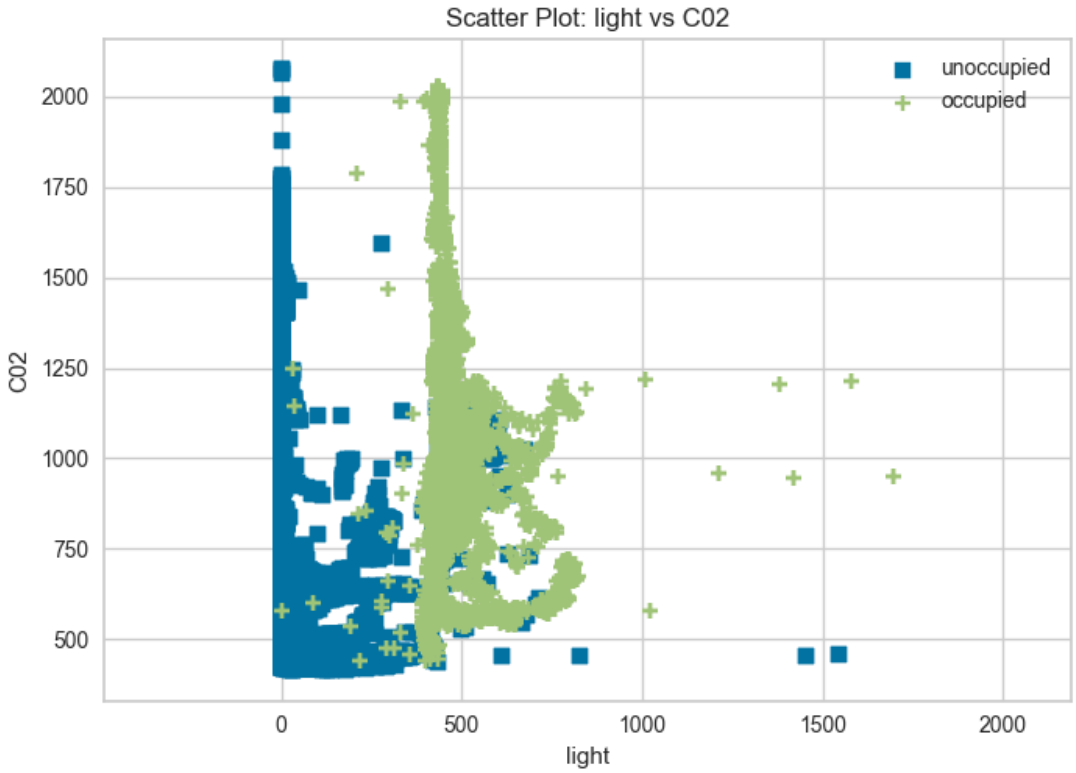
```
# Load the classification data set
data = load_data('occupancy')

# Specify the features of interest and the classes of the target
features = ["temperature", "relative humidity", "light", "C02", "humidity"]
classes = ['unoccupied', 'occupied']

# Extract the numpy arrays from the data frame
```

(continues on next page)

(önceki sayfadan devam)

```
X = data[features]
y = data.occupancy
```

```
from yellowbrick.features import ScatterVisualizer

visualizer = ScatterVisualizer(x='light', y='C02', classes=classes)

visualizer.fit(X, y)
visualizer.transform(X)
visualizer.poof()
```

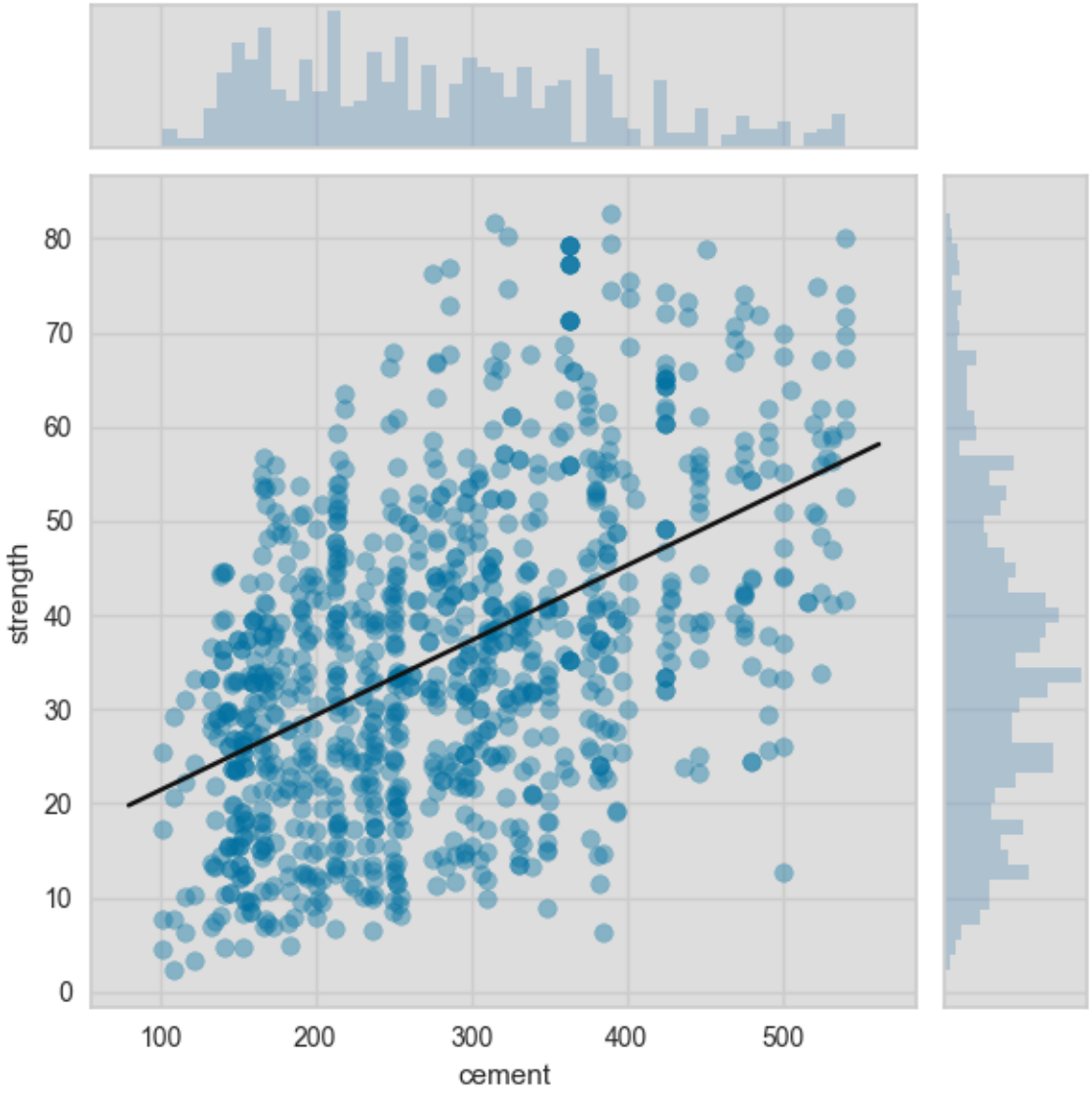


Joint Plot Visualization

A joint plot visualizer plots a feature against the target and shows the distribution of each via a histogram on each axis.

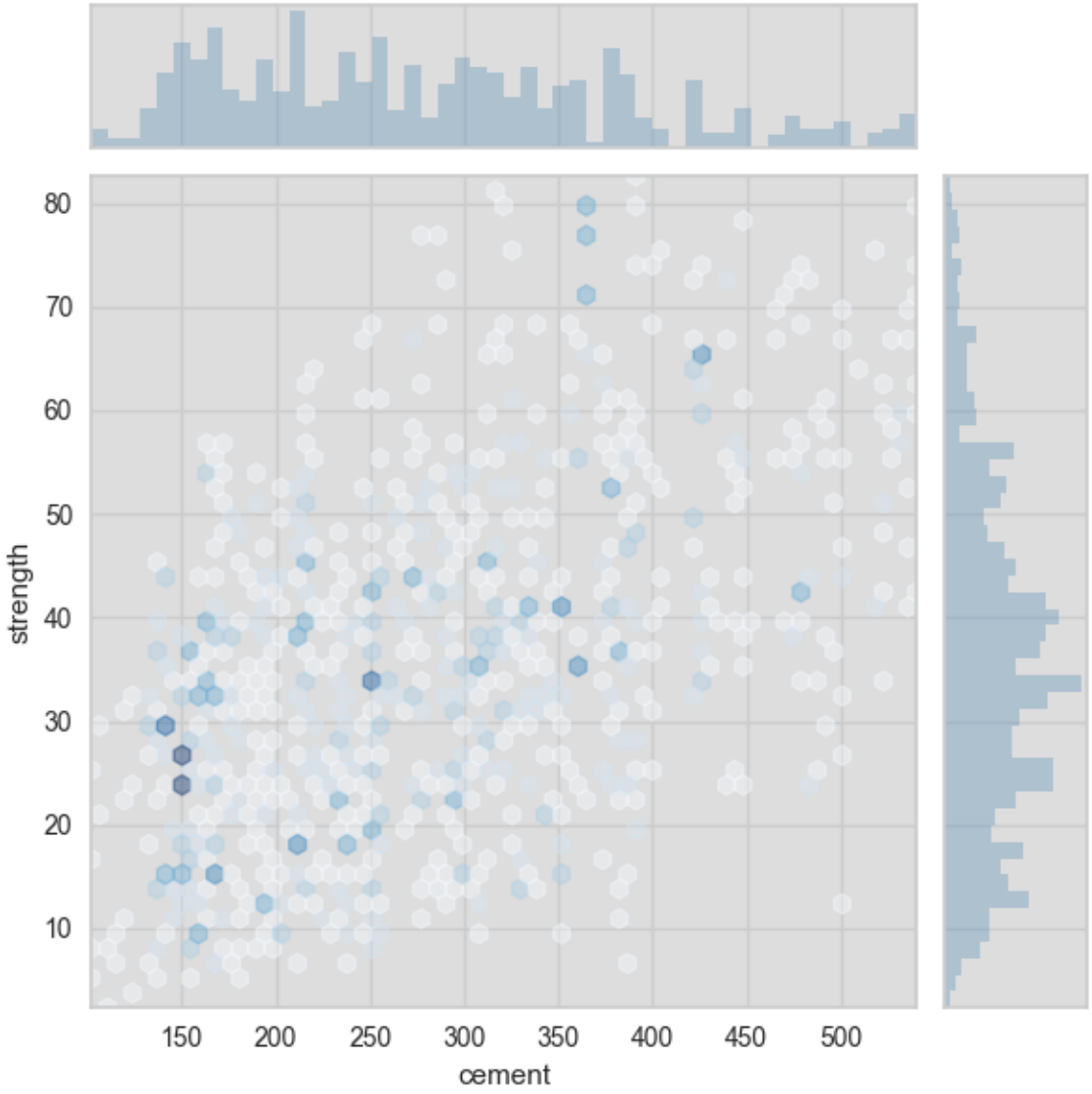
```
# Load the data
df = load_data('concrete')
feature = 'cement'
target = 'strength'

# Get the X and y data from the DataFrame
X = df[feature]
y = df[target]
```

```
visualizer = JointPlotVisualizer(feature=feature, target=target)

visualizer.fit(X, y)
visualizer.poof()
```



The joint plot visualizer can also be plotted with hexbins in the case of many, many points.

```
visualizer = JointPlotVisualizer(
    feature=feature, target=target, joint_plot='hex'
)

visualizer.fit(X, y)
visualizer.poof()
```



API Reference

Implements a 2D scatter plot for feature analysis.

class yellowbrick.features.scatter.**ScatterVisualizer**(*ax=None*, *x=None*, *y=None*, *features=None*, *classes=None*, *color=None*, *colormap=None*, *markers=None*, ***kwargs*)

Bases: yellowbrick.features.base.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

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*)

Finalize executes any subclass-specific axes finalization steps. The user calls poof and poof calls finalize.

Parameters

kwargs: generic keyword arguments.

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

class yellowbrick.features.jointplot.**JointPlotVisualizer**(*ax=None*, *feature=None*, *target=None*, *joint_plot='scatter'*, *joint_args=None*, *xy_plot='hist'*, *xy_args=None*, *size=600*, *ratio=5*, *space=0.2*, ***kwargs*)

Bases: yellowbrick.features.base.FeatureVisualizer

JointPlotVisualizer allows for a simultaneous visualization of the relationship between two variables and the distrbution of each individual variable. The relationship is plotted along the joint axis and univariate distributions are plotted on top of the x axis and to the right of the y axis.

Parameters

ax: matplotlib Axes, default: None This is inherited from FeatureVisualizer but is defined within JointPlotVisualizer since there are three axes objects.

feature: string, default: None The name of the X variable If a DataFrame is passed to fit and feature is None, feature is selected as the column of the DataFrame. There must be only one column in the DataFrame.

target: string, default: None The name of the Y variable If target is None and a y value is passed to fit then the target is selected from the target vector.

joint_plot: one of {'scatter', 'hex'}, default: 'scatter' The type of plot to render in the joint axis Currently, the choices are scatter and hex. Use scatter for small datasets and hex for large datasets

joint_args: dict, default: None Keyword arguments used for customizing the joint plot:

Property	Description
alpha	transparency
facecolor	background color of the joint axis
aspect	aspect ratio
fit	used if scatter is selected for joint_plot to draw a best fit line - values can be True or False. Uses Yellowbrick.bestfit
estimator	used if scatter is selected for joint_plot to determine the type of best fit line to use. Refer to Yellowbrick.bestfit for types of estimators that can be used.
x_bins	used if hex is selected to set the number of bins for the x value
y_bins	used if hex is selected to set the number of bins for the y value
cmap	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.

xy_plot: one of {'hist'}, default: 'hist' The type of plot to render along the x and y axes Currently, the choice is hist

xy_args: dict, default: None Keyword arguments used for customizing the x and y plots:

Property	Description
alpha	transparency
facecolor_x	background color of the x axis
facecolor_y	background color of the y axis
bins	used to set up the number of bins for the hist plot
histcolor_x	used to set the color for the histogram on the x axis
histcolor_y	used to set the color for the histogram on the y axis

size: float, default: 600 Size of each side of the figure in pixels

ratio: float, default: 5 Ratio of joint axis size to the x and y axes height

space: float, default: 0.2 Space between the joint axis and the x and y axes

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 = JointPlotVisualizer()
>>> visualizer.fit(X,y)
>>> visualizer.poof()
```

draw(*X*, *y*, ***kwargs*)
Sets up the layout for the joint plot draw calls `draw_joint` and `draw_xy` to render the visualizations.

draw_joint(*X*, *y*, ***kwargs*)
Draws the visualization for the joint axis.

draw_xy(*X*, *y*, ***kwargs*)
Draws the visualization for the x and y axes

finalize(***kwargs*)
Finalize executes any subclass-specific axes finalization steps. The user calls poof and poof calls finalize.

Parameters

kwargs: generic keyword arguments.

fit(*X*, *y*, ***kwargs*)
Sets up the X and y variables for the jointplot and checks to ensure that X and y are of the correct data type

Fit calls draw

Parameters

X [ndarray or DataFrame of shape n x 1] A matrix of n instances with 1 feature

y [ndarray or Series of length n] An array or series of the target value

kwargs: dict keyword arguments passed to Scikit-Learn API.

poof(***kwargs*)
Creates the labels for the feature and target variables

4.3.4 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`.

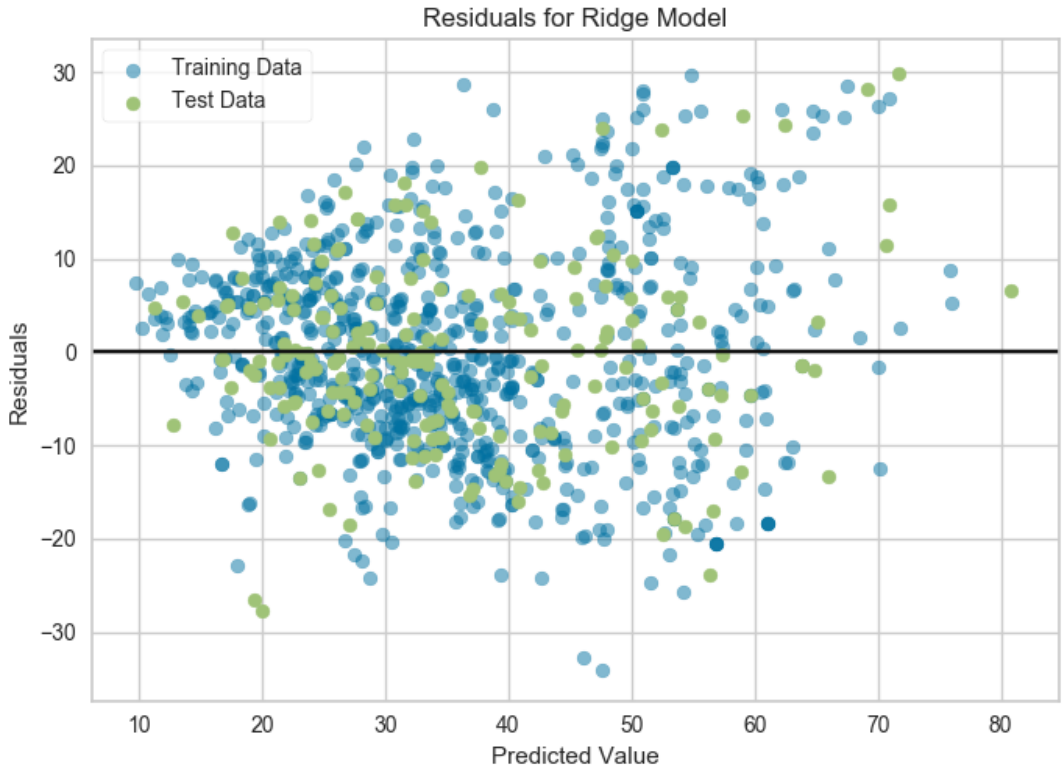
```
# Regression Evaluation Imports

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

A residuals 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.

```
# Load the data
df = load_data('concrete')
feature_names = ['cement', 'slag', 'ash', 'water', 'splast', 'coarse', 'fine', 'age']
target_name = 'strength'

# Get the X and y data from the DataFrame
X = df[feature_names].as_matrix()
y = df[target_name].as_matrix()

# Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
# Instantiate the linear model and visualizer
ridge = Ridge()
visualizer = ResidualsPlot(ridge)

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
g = visualizer.poof()             # Draw/show/poof the data
```



API Reference

Regressor visualizers that score residuals: prediction vs. actual data.

class yellowbrick.regressor.residuals.**ResidualsPlot**(*model*, *ax=None*, ***kwargs*)

Bases: yellowbrick.regressor.base.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

model [a Scikit-Learn regressor] Should be an instance of a regressor, otherwise a will raise a YellowbrickTypeError exception on instantiation.

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).

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 hvaing 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.

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.

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.poof()
```

draw(*y_pred*, *residuals*, *train=False*, ***kwargs*)

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] 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 [the axis with the plotted figure]

finalize(***kwargs*)

Finalize executes any subclass-specific axes finalization steps. The user calls poof and poof calls finalize.

Parameters

kwargs: generic keyword arguments.

fit(*X*, *y=None*, ***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.

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

——

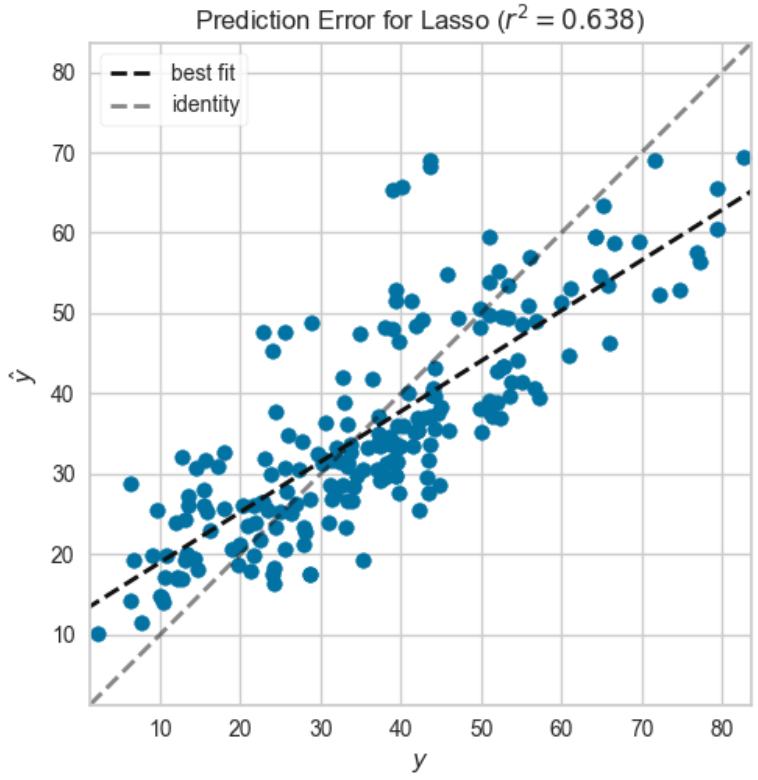
ax [the axis with the plotted figure]

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.

```
# Load the data
df = load_data('concrete')
feature_names = ['cement', 'slag', 'ash', 'water', 'splast', 'coarse', 'fine', 'age']
target_name = 'strength'

# Get the X and y data from the DataFrame
X = df[feature_names].as_matrix()
y = df[target_name].as_matrix()

# Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
# Instantiate the linear model and visualizer
lasso = Lasso()
visualizer = PredictionError(lasso)

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
g = visualizer.poof()             # Draw/show/poof the data
```



API Reference

Regressor visualizers that score residuals: prediction vs. actual data.

class yellowbrick.regressor.residuals.**PredictionError**(*model*, *ax=None*, *shared_limits=True*, *bestfit=True*, *identity=True*, ***kwargs*)

Bases: yellowbrick.regressor.base.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 dectect noise or heteroscedasticity along a range of the target domain.

Parameters

model [a Scikit-Learn regressor] Should be an instance of a regressor, otherwise a will raise a YellowbrickTypeError exception on instantiation.

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.

besfit [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.

point_color [color] Defines the color of the error points; can be any matplotlib color.

line_color [color] Defines the color of the best fit line; can be any matplotlib color.

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.poof()
```

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 [the axis with the plotted figure]

finalize(***kwargs*)

Finalize executes any subclass-specific axes finalization steps. The user calls poof and poof calls finalize.

Parameters

kwargs: generic keyword arguments.

score(*X*, *y=None*, ***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]

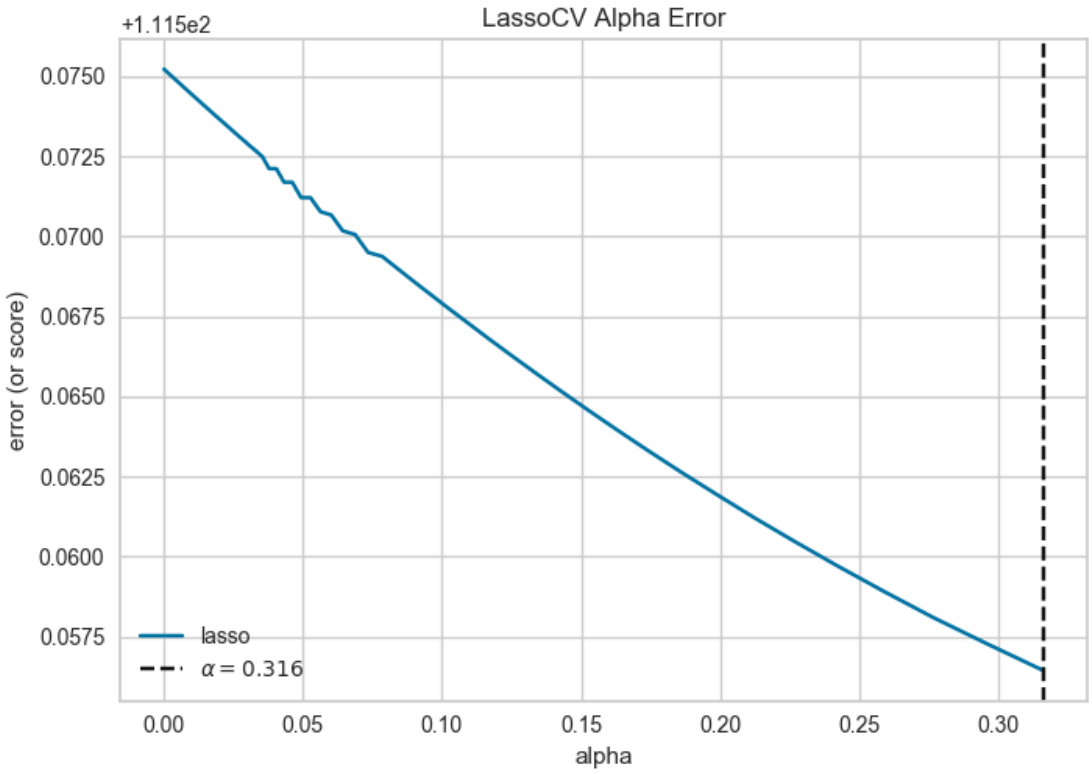
Alpha Selection

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.

```
# Load the data
df = load_data('concrete')
feature_names = ['cement', 'slag', 'ash', 'water', 'splast', 'coarse', 'fine', 'age']
target_name = 'strength'

# Get the X and y data from the DataFrame
X = df[feature_names].as_matrix()
y = df[target_name].as_matrix()

# Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
# Create a list of alphas to cross-validate against
alphas = np.logspace(-12, -0.5, 400)

# Instantiate the linear model and visualizer
model = LassoCV(alphas=alphas)
visualizer = AlphaSelection(model)

visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
g = visualizer.poof()             # Draw/show/poof the data
```



API Reference

Implements alpha selection visualizers for regularization

class `yellowbrick.regressor.alphas.`**AlphaSelection**(*model*, *ax=None*, ***kwargs*)

Bases: `yellowbrick.regressor.base.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

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.

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 hoooks 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.poof()
```

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**(*model*, *ax=None*, *alphas=None*, *cv=None*, *scoring=None*, ***kwargs*)

Bases: *yellowbrick.regressor.alphas.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

model [a Scikit-Learn regressor] Should be an instance of a 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.poof()
```

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.

4.3.5 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 four classifier evaluations:

- *Classification Report*: Presents the classification report of the classifier as a heatmap
- *Confusion Matrix*: Presents the confusion matrix of the classifier as a heatmap
- *ROCAUC*: Presents the graph of receiver operating characteristics along with area under the curve
- *Class Balance*: Displays the difference between the class balances and support
- *Threshold*: Shows the bounds of precision, recall and queue rate after a number of trials.

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.

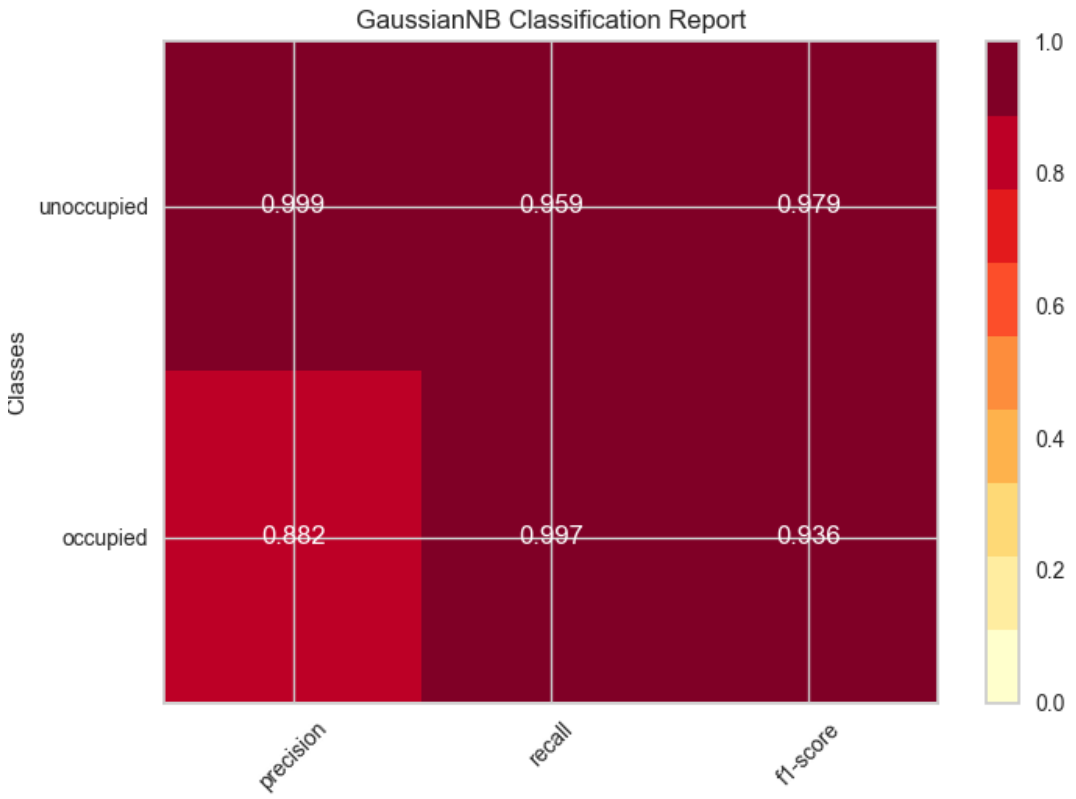
```
# Classifier Evaluation Imports

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.classifier import ClassificationReport, ROCAUC, ClassBalance,␣
↪ThresholdViz
```

Classification Report

The classification report visualizer displays the precision, recall, and F1 scores for the model. In order to support easier interpretation and problem detection, the report integrates numerical scores with a color-coded heatmap.

```
# Load the classification data set
data = load_data('occupancy')

# Specify the features of interest and the classes of the target
features = ["temperature", "relative humidity", "light", "C02", "humidity"]
classes = ['unoccupied', 'occupied']

# Extract the numpy arrays from the data frame
X = data[features].as_matrix()
y = data.occupancy.as_matrix()

# Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
# Instantiate the classification model and visualizer
bayes = GaussianNB()
visualizer = ClassificationReport(bayes, classes=classes)

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
g = visualizer.poof()             # Draw/show/poof the data
```



API Reference

Visual classification report for classifier scoring.

class yellowbrick.classifier.classification_report.**ClassificationReport**(*model*, *ax=None*, *classes=None*, ***kwargs*)

Bases: yellowbrick.classifier.base.ClassificationScoreVisualizer

Classification report that shows the precision, recall, and F1 scores for the model. Integrates numerical scores as well as a color-coded heatmap.

Parameters

ax [The axis to plot the figure on.]

model [the Scikit-Learn estimator] Should be an instance of a classifier, else the __init__ will return an error.

classes [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.

colormap [optional string or matplotlib cmap to colorize lines] Use sequential heatmap.

kwargs [keyword arguments passed to the super class.]

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.poof()
```

draw(*y*, *y_pred*)
Renders the classification report across each axis.

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

finalize(***kwargs*)
Finalize executes any subclass-specific axes finalization steps. The user calls poof and poof calls finalize.

Parameters

kwargs: generic keyword arguments.

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

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.

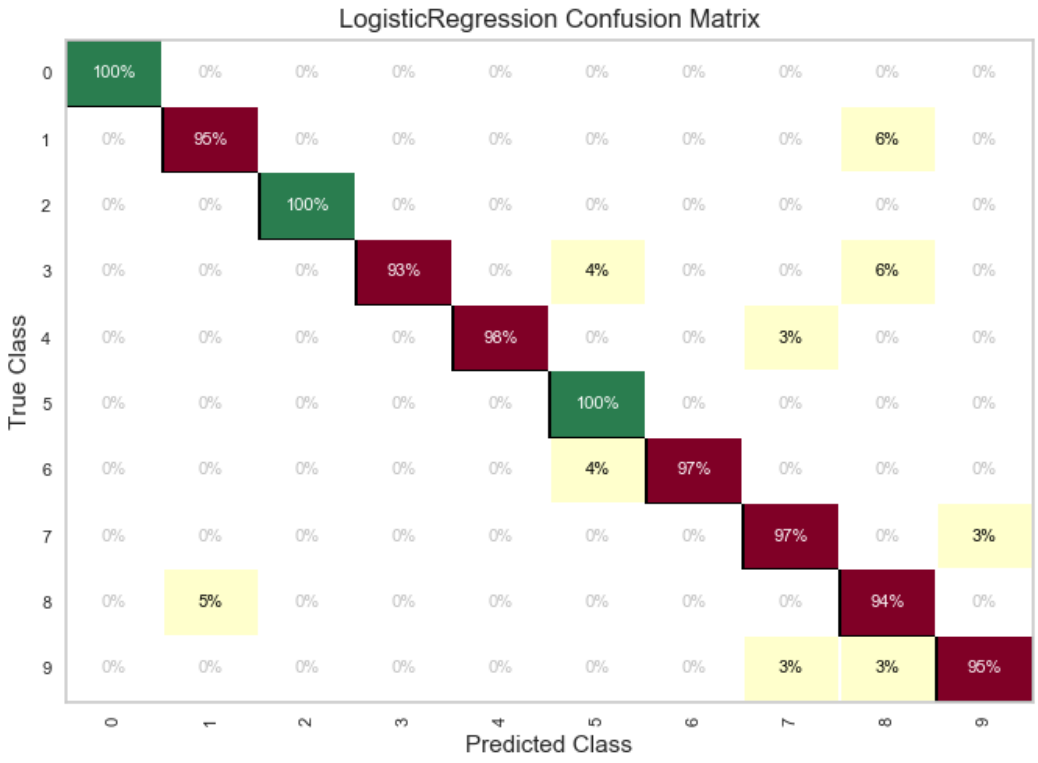
```
#First do our imports
import yellowbrick

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
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
```

(continues on next page)

(önceki sayfadan devam)

```
y = digits.target

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size =0.2, random_
↪state=11)

model = LogisticRegression()

#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.poof()
```



API Reference

Visual confusion matrix for classifier scoring.

class yellowbrick.classifier.confusion_matrix.**ConfusionMatrix**(*model*, *ax=None*, *classes=None*, *label_encoder=None*, ***kwargs*)

Bases: yellowbrick.classifier.base.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

model [the Scikit-Learn estimator] Should be an instance of a classifier or __init__ will return an error.

ax [the matplotlib axis to plot the figure on (if None, a new axis will be created)]

classes [list, default: None] a list of class names to use in the confusion_matrix. This is passed to the 'labels' parameter of sklearn.metrics.confusion_matrix(), and follows the behaviour indicated by that function. It may be used to reorder or select a subset of labels. If None, values that appear at least once in y_true or y_pred are used in sorted order.

label_encoder [dict or LabelEncoder, default: None] When specifying the `classes` argument, the input to `fit()` and `score()` must match the expected labels. If the `X` and `y` datasets have been encoded prior to training and the labels must be preserved for the visualization, use this argument to provide a mapping from the encoded class to the correct label. Because typically a Scikit-Learn `LabelEncoder` is used to perform this operation, you may provide it directly to the class to utilize its fitted encoding.

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.poof()
```

draw(*percent=True*)

Renders the classification report Should only be called internally, as it uses values calculated in Score and score calls this method.

Parameters

percent: Boolean Whether the heatmap should represent "% of True" or raw counts

finalize(***kwargs*)

Finalize executes any subclass-specific axes finalization steps.

Parameters

kwargs: dict generic keyword arguments.

Notes

The user calls poof and poof calls finalize. Developers should implement visualizer-specific finalization methods like setting titles or axes labels, etc.

score(*X*, *y*, *sample_weight=None*, *percent=True*)

Generates the Scikit-Learn confusion_matrix and applies this to the appropriate 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

sample_weight: optional, passed to the confusion_matrix

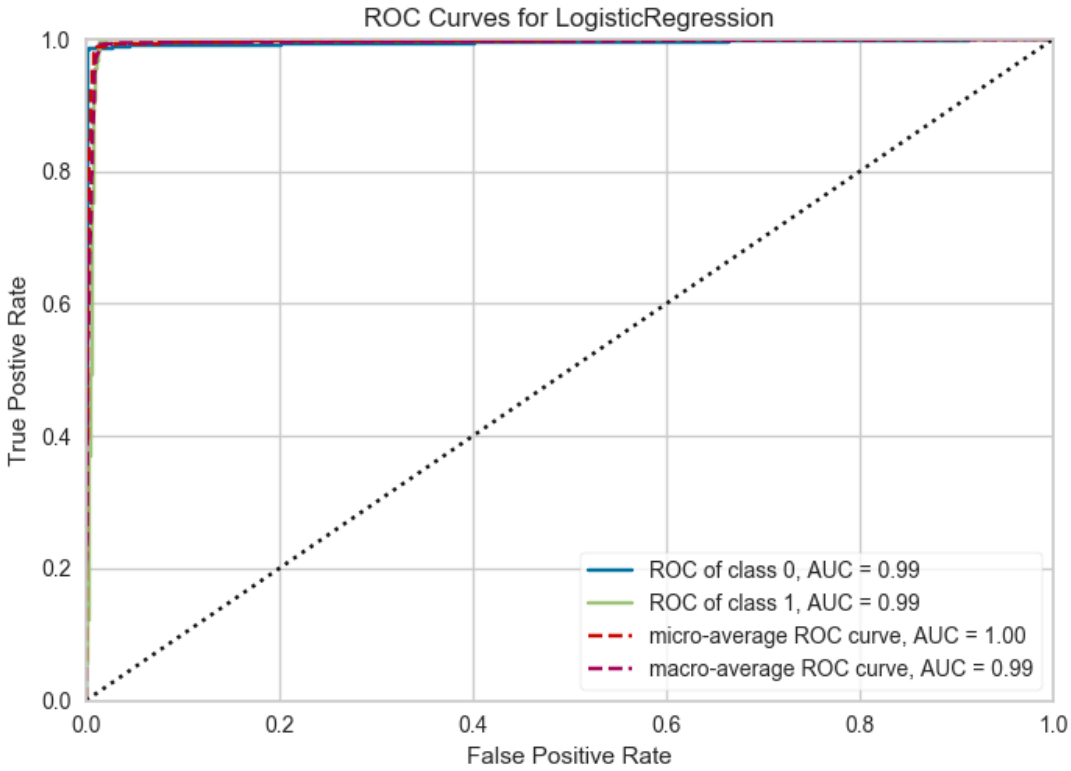
percent: optional, Boolean. Determines whether or not the confusion_matrix should be displayed as raw numbers or as a percent of the true predictions. Note, if using a subset of classes in __init__, percent should be set to False or inaccurate percents will be displayed.

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.

```
# Load the classification data set
data = load_data('occupancy')

# Specify the features of interest and the classes of the target
features = ["temperature", "relative humidity", "light", "C02", "humidity"]
classes = ['unoccupied', 'occupied']

# Extract the numpy arrays from the data frame
X = data[features].as_matrix()
y = data.occupancy.as_matrix()

# Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
# Instantiate the classification model and visualizer
logistic = LogisticRegression()
visualizer = ROCAUC(logistic)

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
g = visualizer.poof()             # Draw/show/poof the data
```



API Reference

Implements visual ROC/AUC curves for classification evaluation.

class yellowbrick.classifier.rocauc.**ROCAUC**(*model*, *ax=None*, *classes=None*, *micro=True*, *macro=True*, *per_class=True*, ***kwargs*)

Bases: yellowbrick.classifier.base.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

ax [the axis to plot the figure on.]

model [the Scikit-Learn estimator] Should be an instance of a classifier, else the __init__ will return an error.

classes [list] 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. Note that the curves must be computed

based on what is in the target vector passed to the `score()` method. Class names are used for labeling only and must be in the correct order to prevent confusion.

micro [bool, default = True] Plot the micro-averages ROC curve, computed from the sum of all true positives and false positives across all classes.

macro [bool, default = True] Plot the macro-averages ROC curve, which simply takes the average of curves across all classes.

per_class [bool, default = True] Plot the ROC curves for each individual class. Primarily this is set to false if only the macro or micro average curves are required.

kwargs [keyword arguments passed to the super class.] Currently passing in hard-coded colors for the Receiver Operating Characteristic curve and the diagonal. These will be refactored to a default Yellowbrick style.

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.

Ayrıca bkz.:

http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from yellowbrick.classifier import ROCAUC
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.model_selection import train_test_split
>>> data = load_breast_cancer()
>>> X = data['data']
>>> y = data['target']
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)
>>> viz = ROCAUC(LogisticRegression())
>>> viz.fit(X_train, y_train)
>>> viz.score(X_test, y_test)
>>> viz.poof()
```

draw()
Renders ROC-AUC plot. Called internally by score, possibly more than once

Returns

ax [the axis with the plotted figure]

finalize(***kwargs*)
Finalize executes any subclass-specific axes finalization steps. The user calls poof and poof calls finalize.

Parameters

kwargs: generic keyword arguments.

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

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

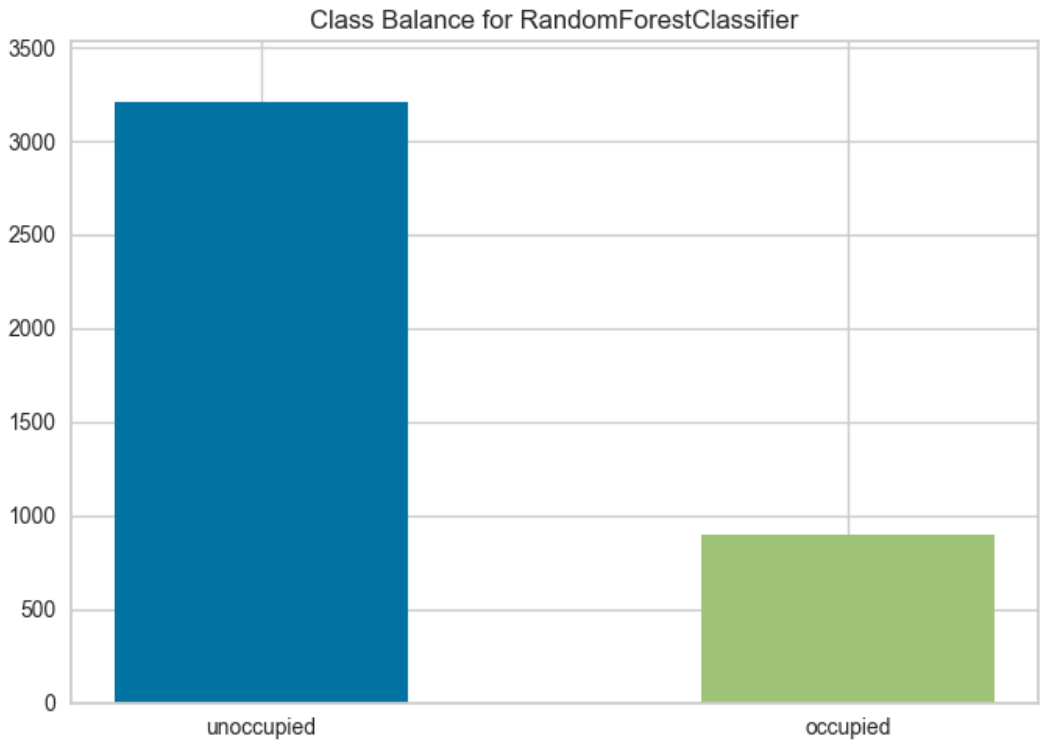
score [float] The micro-average area under the curve of all classes.

Class Balance

Oftentimes classifiers perform badly because of a class imbalance. A class balance chart can help prepare the user for such a case by showing the support for each class in the fitted classification model.

```
# Load the classification data set
data = load_data('occupancy')

# Specify the features of interest and the classes of the target
features = ["temperature", "relative humidity", "light", "C02", "humidity"]
classes = ['unoccupied', 'occupied']

# Extract the numpy arrays from the data frame
X = data[features].as_matrix()
y = data.occupancy.as_matrix()

# Create the train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
# Instantiate the classification model and visualizer
forest = RandomForestClassifier()
visualizer = ClassBalance(forest, classes=classes)

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
g = visualizer.poof()             # Draw/show/poof the data
```



API Reference

Class balance visualizer for showing per-class support.

class yellowbrick.classifier.class_balance.**ClassBalance**(*model*, *ax=None*, *classes=None*, ***kwargs*)

Bases: yellowbrick.classifier.base.ClassificationScoreVisualizer

Class balance chart that shows the support for each class in the fitted classification model displayed as a bar plot. It is initialized with a fitted model and generates a class balance chart on draw.

Parameters

ax: axes the axis to plot the figure on.

model: estimator Scikit-Learn estimator object. Should be an instance of a classifier, else `__init__()` will raise an exception.

classes: list 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 passed to the super class. Here, used to colorize the bars in the histogram.

Notes

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

draw()
Renders the class balance chart across the axis.

Returns

ax [the axis with the plotted figure]

finalize(***kwargs*)
Finalize executes any subclass-specific axes finalization steps. The user calls poof and poof calls finalize.

Parameters

kwargs: generic keyword arguments.

score(*X*, *y=None*, ***kwargs*)
Generates the Scikit-Learn precision_recall_fscore_support

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

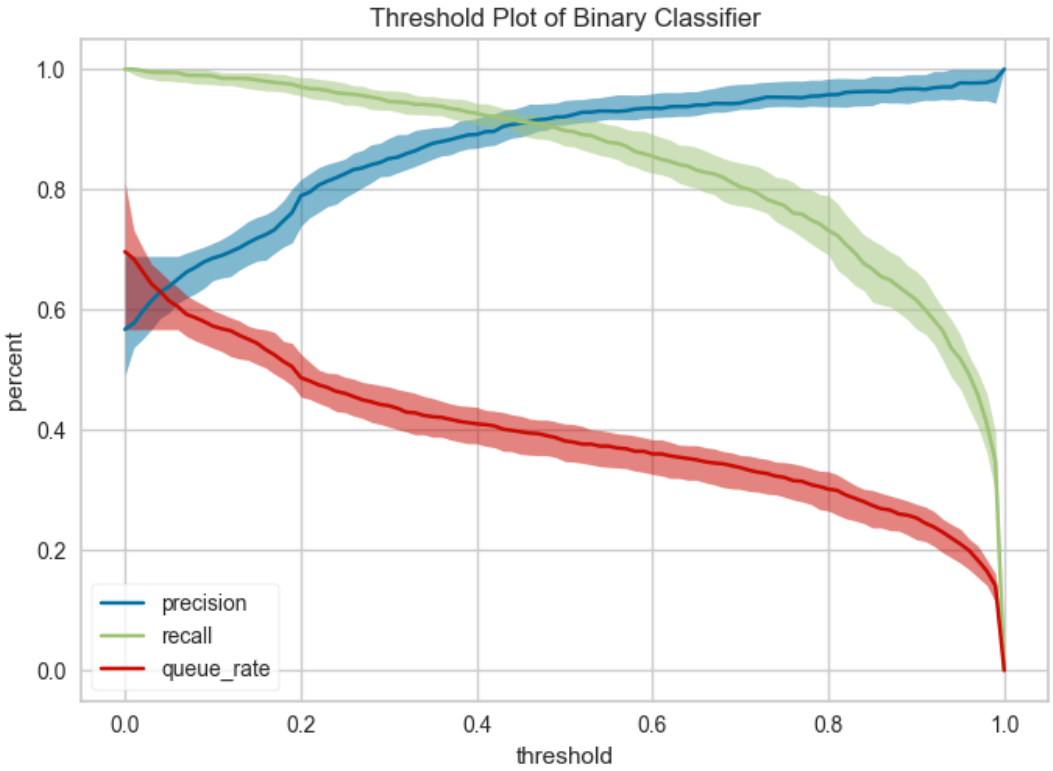
ax [the axis with the plotted figure]

Threshold

The Threshold visualizer shows the bounds of precision, recall and queue rate for different thresholds for binary targets after a given number of trials.

```
# Load the data set
data = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/
↪spambase/spambase.data', header=None)
data.rename(columns={57:'is_spam'}, inplace=True)

features = [col for col in data.columns if col != 'is_spam']

# Extract the numpy arrays from the data frame
X = data[features].as_matrix()
y = data.is_spam.as_matrix()
```

```
# Instantiate the classification model and visualizer
logistic = LogisticRegression()
visualizer = ThreshViz(logistic)

visualizer.fit(X, y)  # Fit the training data to the visualizer
g = visualizer.poof() # Draw/show/poof the data
```



API Reference

`yellowbrick.classifier.threshold.`**`ThreshViz`**
şunun takma adı: `yellowbrick.classifier.threshold.ThresholdVisualizer`

4.3.6 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 two 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.

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 *poof()* to display the clustering evaluation metric.

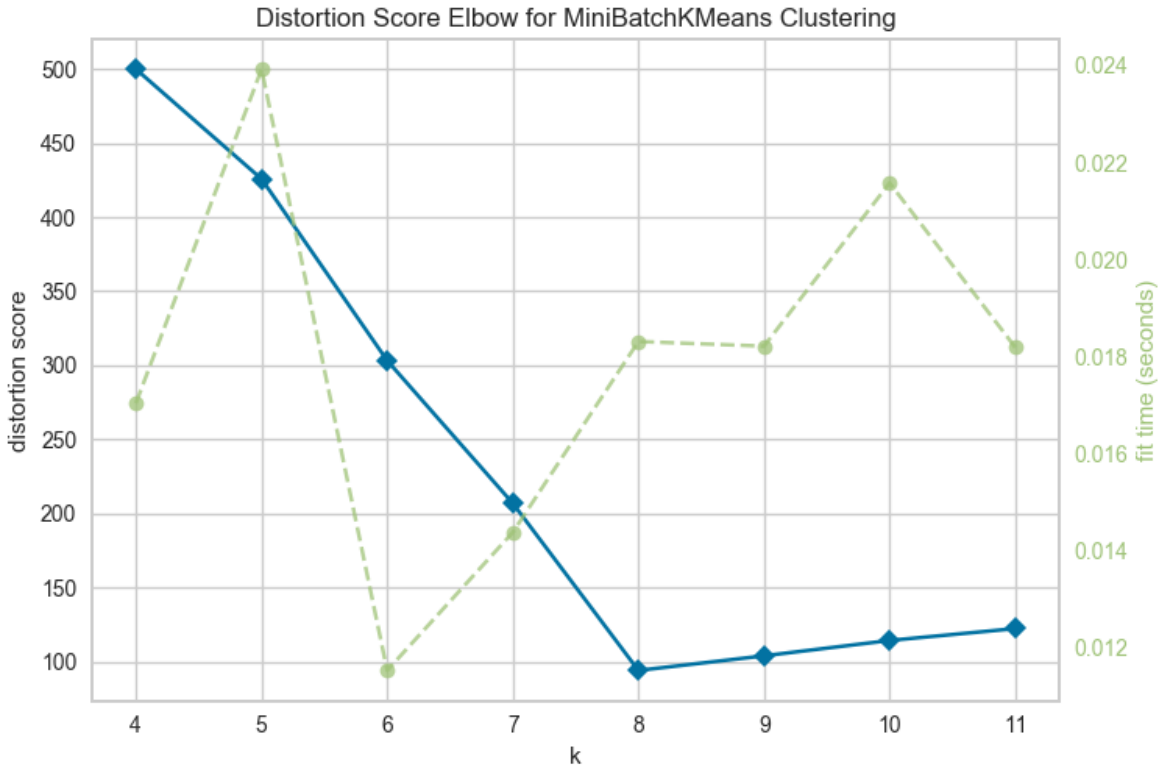
Elbow Method

The elbow method for K selection visualizes multiple clustering models with different values for K. Model selection is based on whether or not there is an "elbow" in the curve; e.g. if the curve looks like an arm, if there is a clear change

in angle from one part of the curve to another.

```
# Make 8 blobs dataset
X, y = make_blobs(centers=8)
```

```
# Instantiate the clustering model and visualizer
visualizer = KElbowVisualizer(MiniBatchKMeans(), k=(4,12))

visualizer.fit(X) # Fit the training data to the visualizer
visualizer.poof() # Draw/show/poof the data
```



API Reference

Implements the elbow method for determining the optimal number of clusters. https://bl.ocks.org/rpgove/0060ff3b656618e9136b

class yellowbrick.cluster.elbow.**KElbowVisualizer**(*model*, *ax=None*, *k=10*, *metric='distortion'*, *timings=True*, ***kwargs*)

Bases: yellowbrick.cluster.base.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_harabaz_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

model [a Scikit-Learn clusterer] Should be an instance of a 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 or tuple] The range of k to compute silhouette scores for. If a single integer is specified, then will compute the range (2,k) otherwise the specified range in the tuple is used.

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_harabaz**: ratio of within to between cluster dispersion

timings [bool, default: True] Display the fitting time per k to evaluate the amount of time required to train the clustering model.

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.

Examples

```
>>> from yellowbrick.cluster import KElbowVisualizer
>>> from sklearn.cluster import KMeans
>>> model = KElbowVisualizer(KMeans(), k=10)
>>> model.fit(X)
>>> model.poof()
```

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 silhoutte scores in the `self.k_scores_` attribute. This method finishes up by calling draw to create the plot.

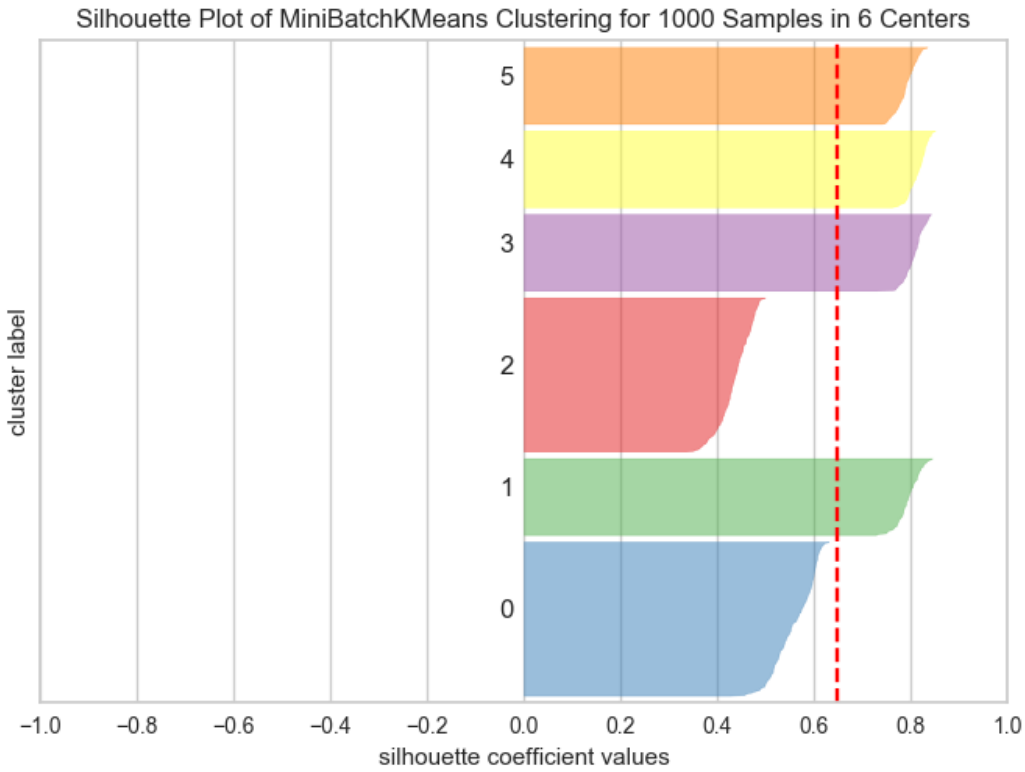
Silhouette Visualizer

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.

```
# Make 8 blobs dataset
X, y = make_blobs(centers=8)
```

```
# Instantiate the clustering model and visualizer
model = MiniBatchKMeans(6)
visualizer = SilhouetteVisualizer(model)

visualizer.fit(X) # Fit the training data to the visualizer
visualizer.poof() # Draw/show/poof the data
```



API Reference

Implements visualizers that use the silhouette metric for cluster evaluation.

class yellowbrick.cluster.silhouette.**SilhouetteVisualizer**(*model*, *ax=None*, ***kwargs*)

Bases: yellowbrick.cluster.base.ClusteringScoreVisualizer

TODO: Document this class!

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 the silhouette visualization.

TODO: decide to use this method or the score method to draw. NOTE: Probably this would be better in score, but the standard score is a little different and I'm not sure how it's used.

4.3.7 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 distribution, showing document similarity, or simply wrapping some of the other standard visualizers with text-specific display properties.

We currently have two 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

Note that the examples in this section require a corpus of text data, see *loading a text corpus* for more information.

```
from yellowbrick.text import FreqDistVisualizer
from yellowbrick.text import TSNEVisualizer

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
```

Loading a Text Corpus

As in the previous sections, Yellowbrick has provided a sample dataset to run the following cells. In particular, we are going to use a text corpus wrangled from the Baleen RSS Corpus to present the following examples. If you haven't already downloaded the data, you can do so by running:

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

Note that this will create a directory called `data` in your current working directory that contains subdirectories with the provided datasets.

Not: If you've already followed the instructions from *downloading example datasets*, you don't have to repeat these steps here. Simply check to ensure there is a directory called `hobbies` in your data directory.

The following code snippet creates a utility that will load the corpus from disk into a Scikit-Learn Bunch object. This method creates a corpus that is exactly the same as the one found in the "working with text data" example on the Scikit-Learn website, hopefully making the examples easier to use.

```
import os
from sklearn.datasets.base import Bunch

def load_corpus(path):
    """
    Loads and wrangles the passed in text corpus by path.
    """

    # Check if the data exists, otherwise download or raise
    if not os.path.exists(path):
        raise ValueError((
            "'{}' dataset has not been downloaded, "
            "use the yellowbrick.download module to fetch datasets"
        ).format(path))

    # Read the directories in the directory as the categories.
    categories = [
        cat for cat in os.listdir(path)
        if os.path.isdir(os.path.join(path, cat))
    ]

    files  = [] # holds the file names relative to the root
    data   = [] # holds the text read from the file
    target = [] # holds the string of the category

    # Load the data from the files in the corpus
    for cat in categories:
        for name in os.listdir(os.path.join(path, cat)):
            files.append(os.path.join(path, cat, name))
            target.append(cat)

            with open(os.path.join(path, cat, name), 'r') as f:
                data.append(f.read())


    # Return the data bunch for use similar to the newsgroups example
    return Bunch(
        categories=categories,
        files=files,
        data=data,
        target=target,
    )
```

This is a fairly long ibt of code, so let's walk through it step by step. The data in the corpus directory is stored 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
```

Each of the documents in the corpus is stored in a text file labeled with its hash signature in a directory that specifies its label or category. Therefore the first step after checking to make sure the specified path exists is to list all the directories in the `hobbies` directory – this gives us each of our categories, which we will store later in the bunch.

The second step is to create placeholders for holding filenames, text data, and labels. We can then loop through the list of categories, list the files in each category directory, add those files to the files list, add the category name to the target list, then open and read the file to add it to data.

To load the corpus into memory, we can simply use the following snippet:

```
corpus = load_corpus("data/hobbies")
```

We'll use this snippet in all of the text examples in this section!

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.

```
from yellowbrick.text.freqdist import FreqDistVisualizer
from sklearn.feature_extraction.text import CountVectorizer
```

Note that the `FreqDistVisualizer` does not perform any normalization or vectorization, and it expects text that has already be count vectorized.

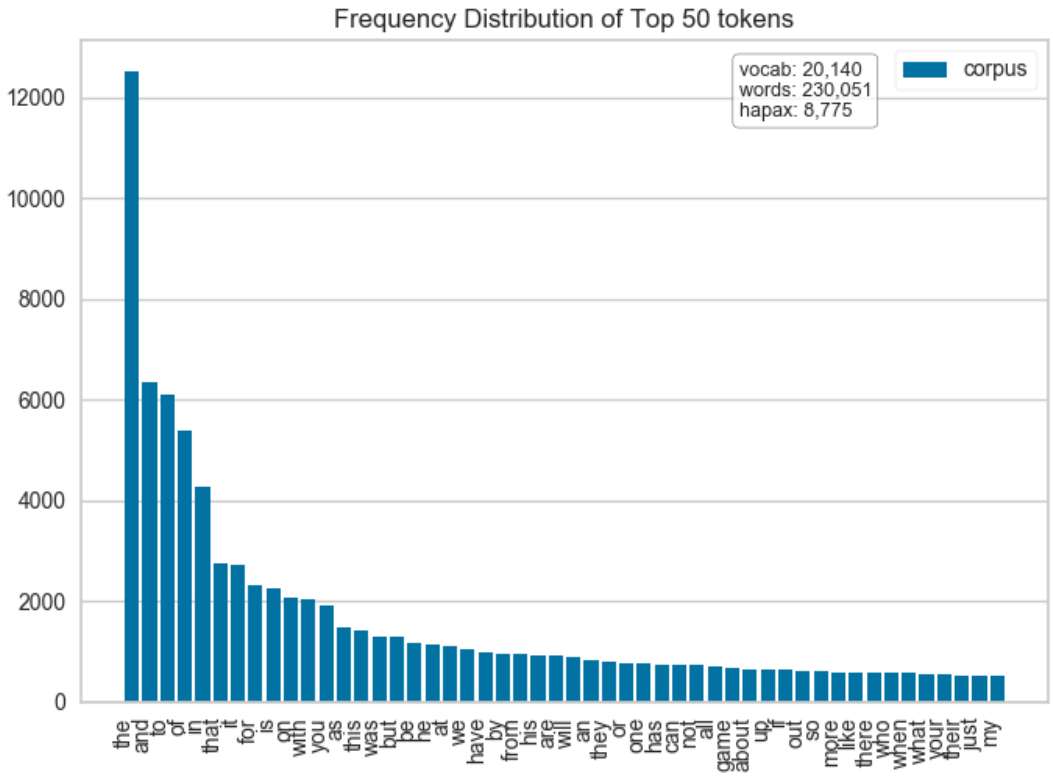
We first instantiate a `FreqDistVisualizer` object, and then call `fit()` on that object with the count vectorized documents and the features (i.e. the words from the corpus), which computes the frequency distribution. The visualizer then plots a bar chart of the top 50 most frequent terms in the corpus, 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 `poof()`, the finalized visualization is shown.

```
vectorizer = CountVectorizer()
docs       = vectorizer.fit_transform(corpus.data)
features   = vectorizer.get_feature_names()

visualizer = FreqDistVisualizer(features=features)
```

(continues on next page)

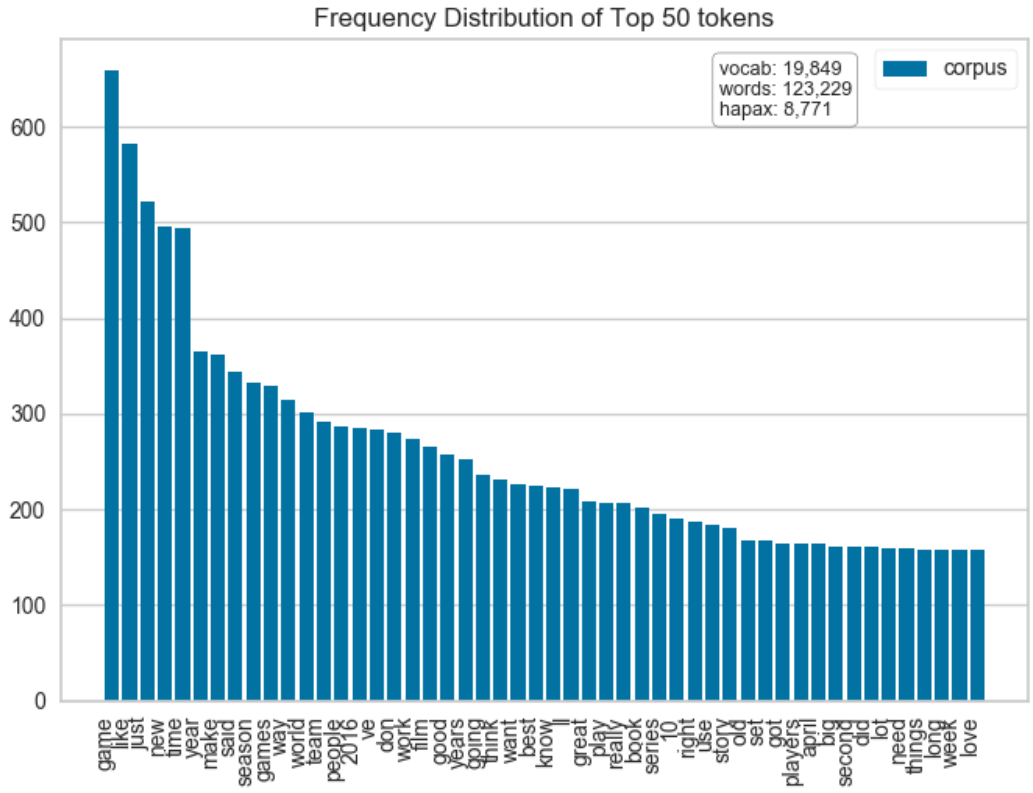
(önceki sayfadan devam)

```
visualizer.fit(docs)
visualizer.poof()
```



It is interesting to compare the results of the `FreqDistVisualizer` before and after stopwords have been removed from the corpus:

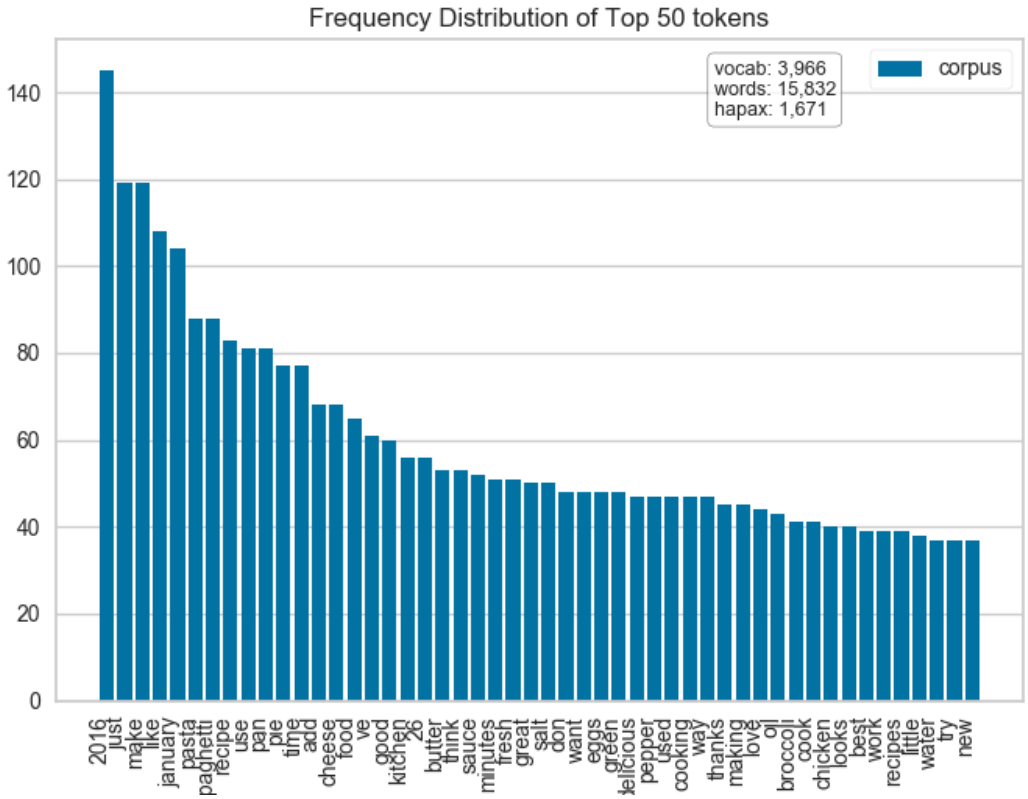
```
vectorizer = CountVectorizer(stop_words='english')
docs       = vectorizer.fit_transform(corpus.data)
features   = vectorizer.get_feature_names()

visualizer = FreqDistVisualizer(features=features)
visualizer.fit(docs)
visualizer.poof()
```



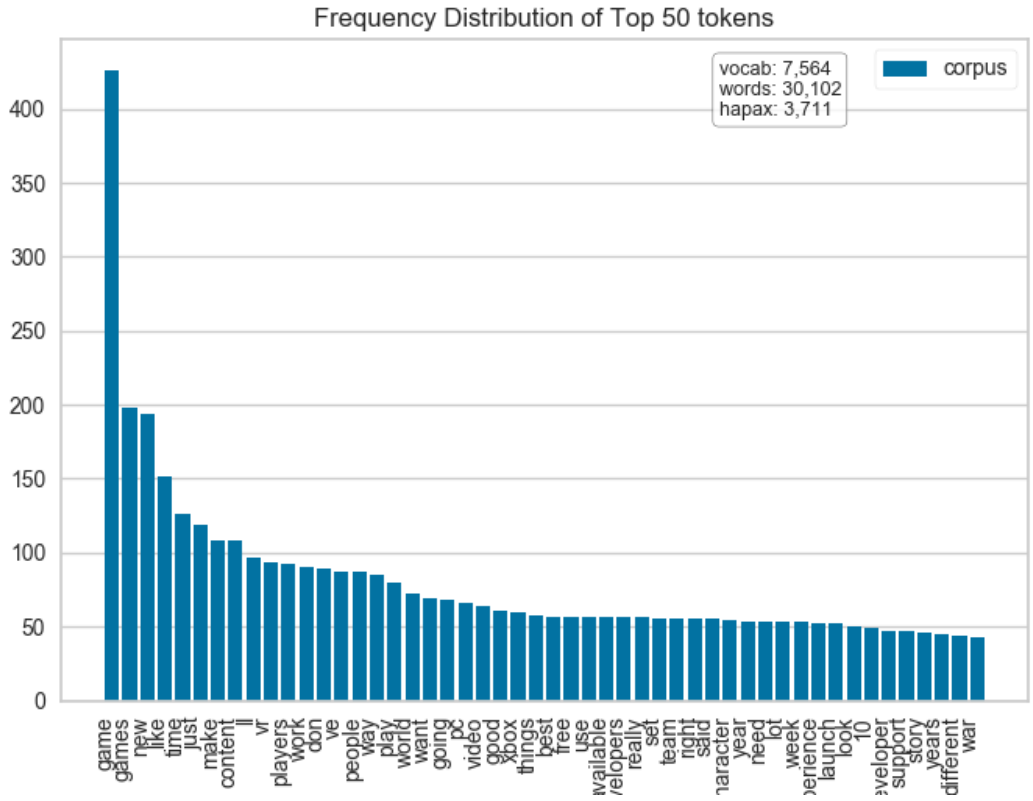
It is also interesting to explore the differences in tokens across a corpus. The hobbies corpus that comes with Yellowbrick has already been categorized (try `corpus['categories']`), so let's visually compare the differences in the frequency distributions for two of the categories: *"cooking"* and *"gaming"*.

```
from collections import defaultdict

hobbies = defaultdict(list)
for text, label in zip(corpus.data, corpus.label):
    hobbies[label].append(text)
```

```
vectorizer = CountVectorizer(stop_words='english')
docs       = vectorizer.fit_transform(text for text in hobbies['cooking'])
features   = vectorizer.get_feature_names()

visualizer = FreqDistVisualizer(features=features)
visualizer.fit(docs)
visualizer.poof()
```




```
vectorizer = CountVectorizer(stop_words='english')
docs       = vectorizer.fit_transform(text for text in hobbies['gaming'])
features   = vectorizer.get_feature_names()

visualizer = FreqDistVisualizer(features=features)
visualizer.fit(docs)
visualizer.poof()
```



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: yellowbrick.text.base.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 [list or tuple of colors] 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 poof & poof 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.

.. note:: Text documents must be vectorized before ‘‘fit()‘‘.

t-SNE Corpus Visualization

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.

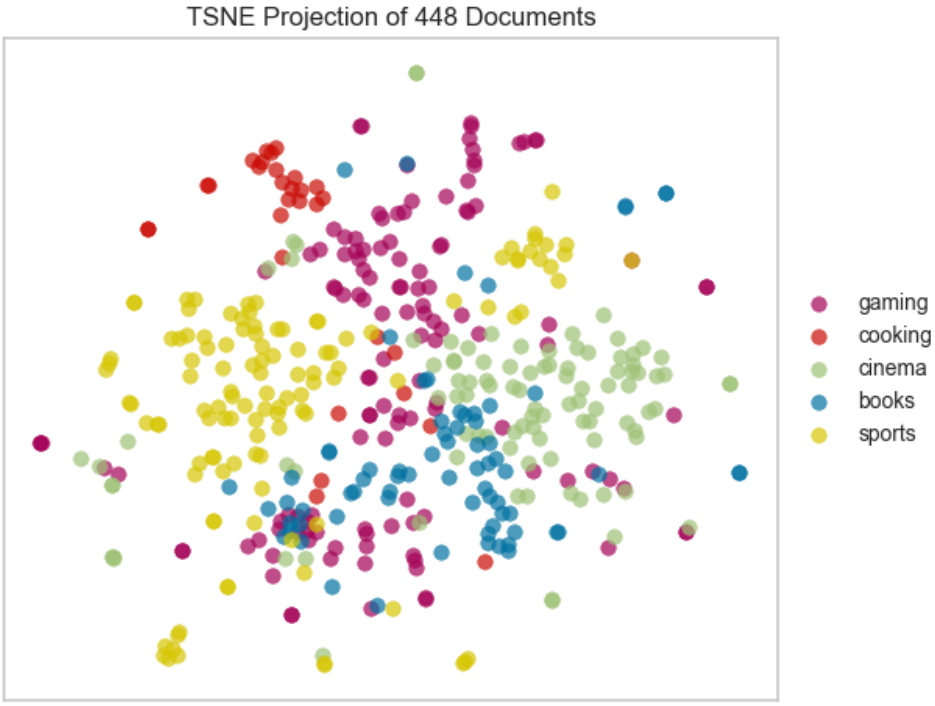
```
from yellowbrick.text import TSNEVisualizer
from sklearn.feature_extraction.text import TfidfVectorizer
```

After importing the required tools, we can *load the corpus* and vectorize the text using TF-IDF.

```
# Load the data and create document vectors
corpus = load_corpus('hobbies')
tfidf  = TfidfVectorizer()

docs   = tfidf.fit_transform(corpus.data)
labels = corpus.target
```

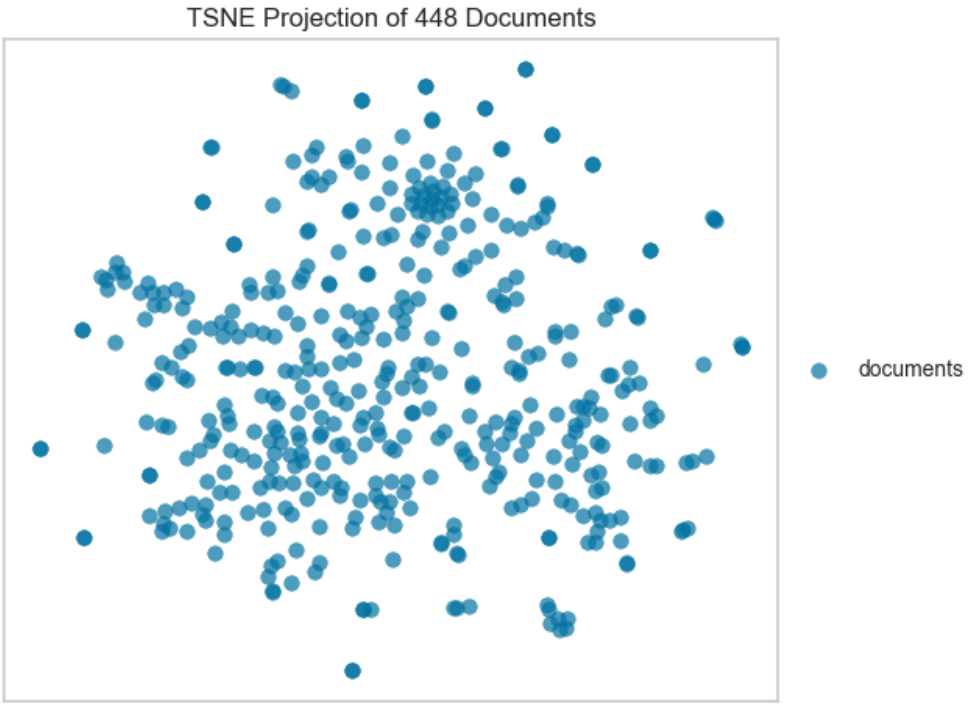
Now that the corpus is vectorized we can visualize it, showing the distribution of classes.

```
# Create the visualizer and draw the vectors
tsne = TSNEVisualizer()
tsne.fit(docs, labels)
tsne.poof()
```



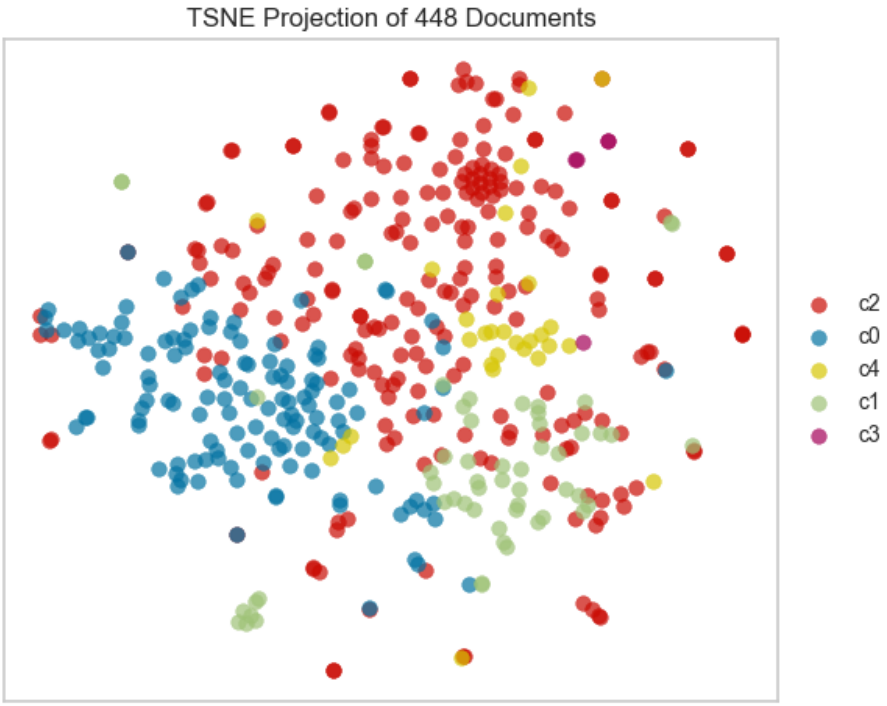
If we omit the target during fit, we can visualize the whole dataset to see if any meaningful patterns are observed.

```
# Don't color points with their classes
tsne = TSNEVisualizer(labels=["documents"])
tsne.fit(docs)
tsne.poof()
```



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, looking for clusters of related text by their contents:

```
# Apply clustering instead of class names.
from sklearn.cluster import KMeans

clusters = KMeans(n_clusters=5)
clusters.fit(docs)

tsne = TSNEVisualizer()
tsne.fit(docs, ["c{}".format(c) for c in clusters.labels_])
tsne.poof()
```



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*, ***kwargs*)

Bases: yellowbrick.text.base.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.

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

4.3.8 Renkler ve Stil

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 of course you can customize your own look and feel by directly modifying the visualization with matplotlib.

Yellowbrick prioritizes color in its visualizations for most visualizers. 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 about how each color argument is handled. In the next two sections we will show every possible color palette and sequence currently available in Yellowbrick.

Renk Paletleri

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 or 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')
```

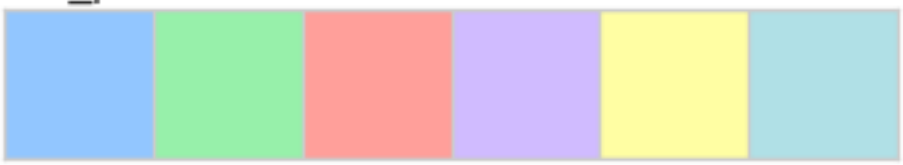
Color palettes are most often used for classifiers to show the relationship between discrete class labels. They can also be used for clustering algorithms to show membership in discrete clusters.

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')
```


reset
pastel
colorblind
bold
flatui
muted

sns_pastel



sns_deep

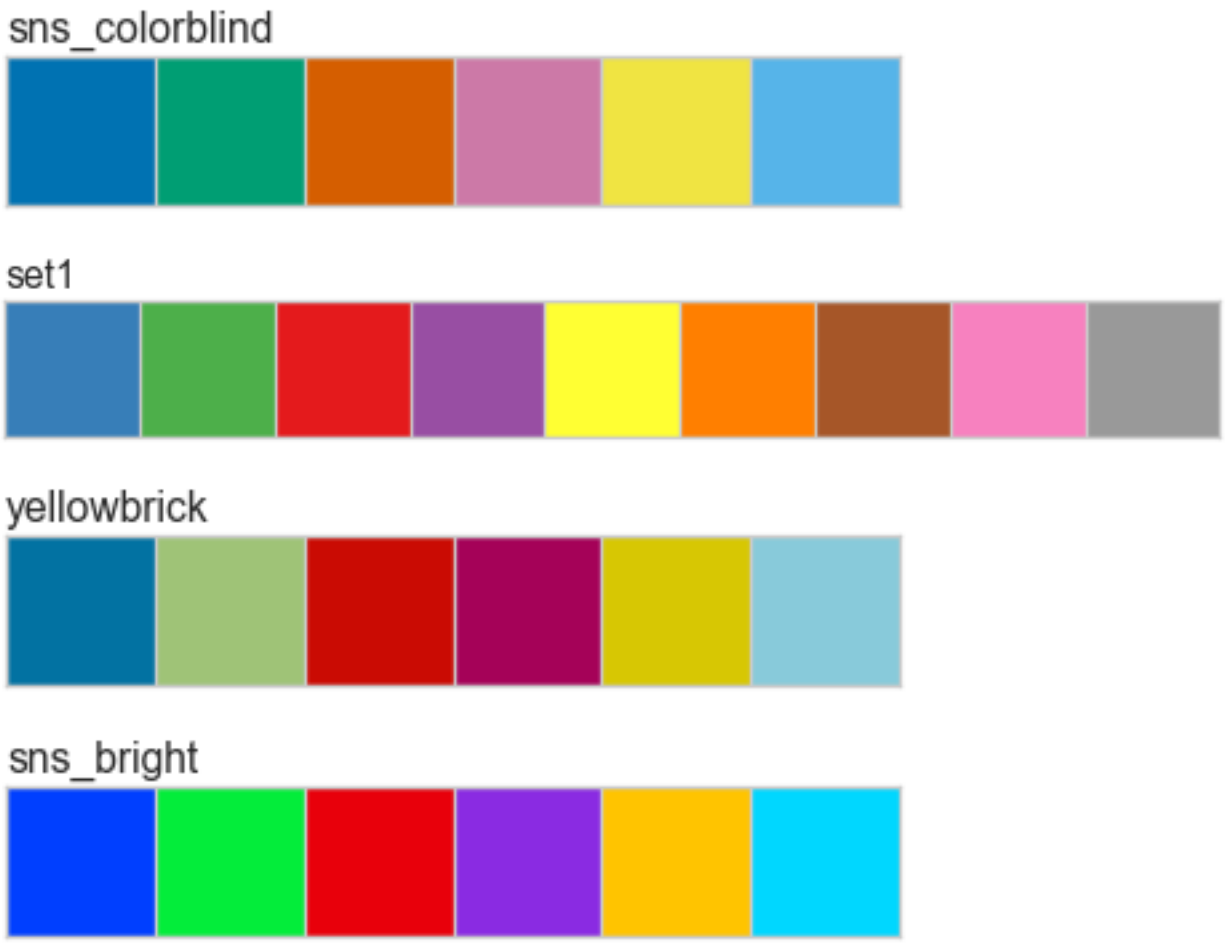
accent

sns_dark

dark

paired

sns_muted



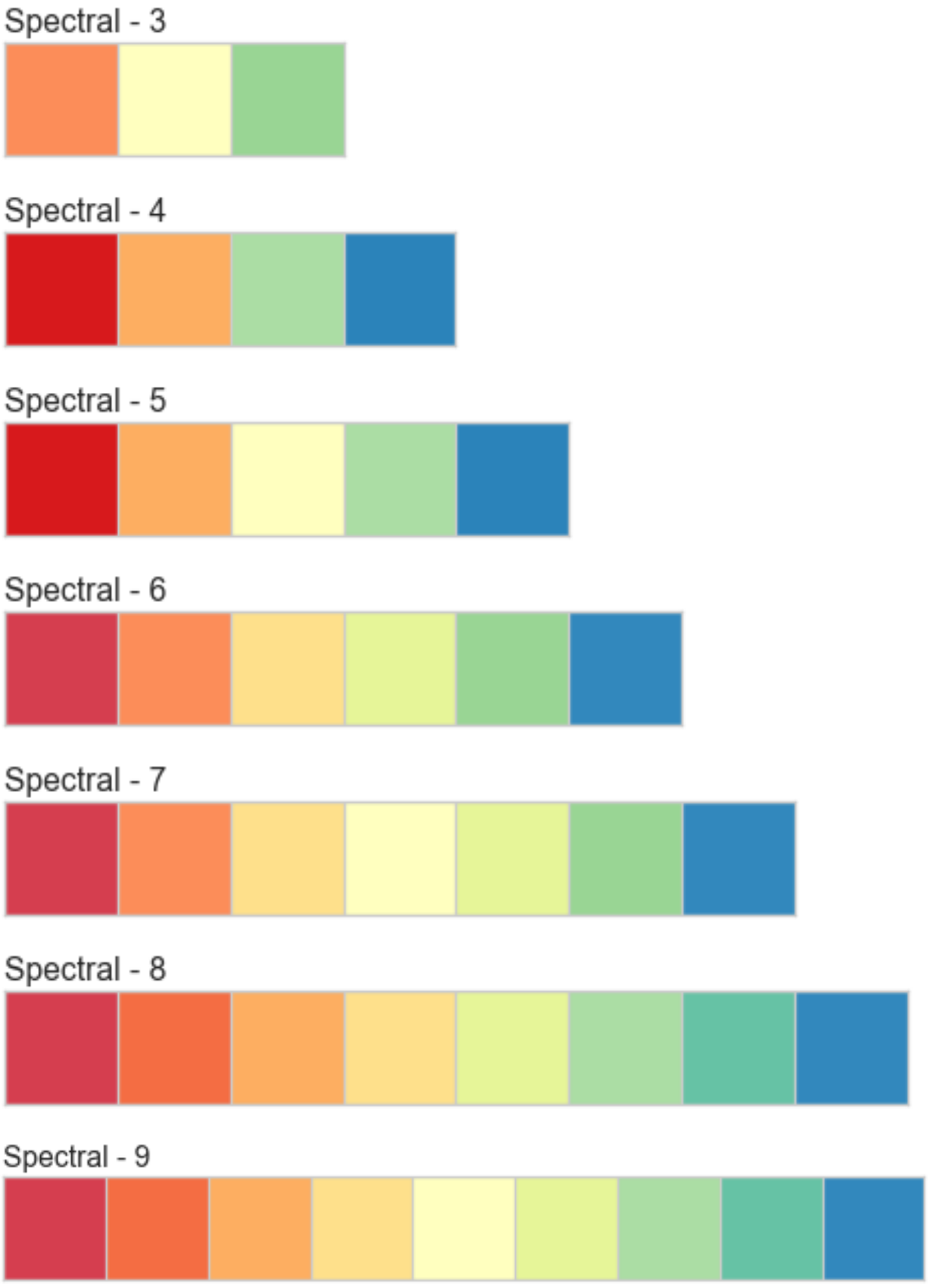
Renk Dizileri

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 values are assigned correctly. In the listing below, each sequence is shown with varying lengths to describe the range of colors in detail.

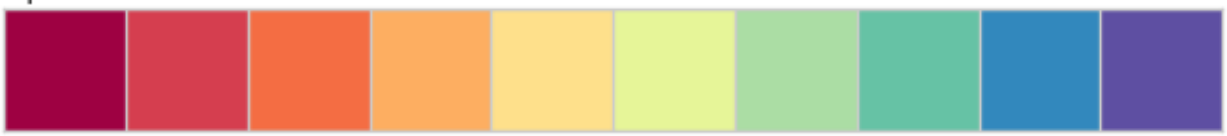
Color sequences are most often used in regressions to show the distribution in the range of target values. They can also be used in clustering and distribution analysis to show distance or histogram data.

Below is a complete listing of all the sequence names available in Yellowbrick:

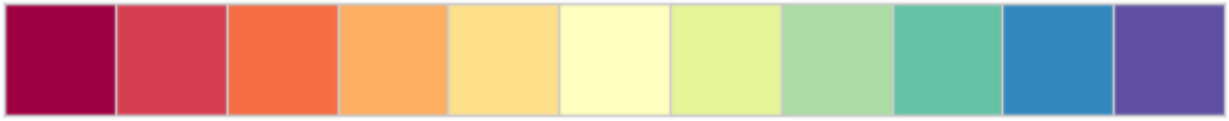
```
for name, maps in SEQUENCES.items():
    for num, palette in maps.items():
        color_palette(palette).plot()
        plt.title("{} - {}".format(name, num), loc='left')
```


Spectral - 3
Spectral - 4
Spectral - 5
Spectral - 6
Spectral - 7
Spectral - 8
Spectral - 9

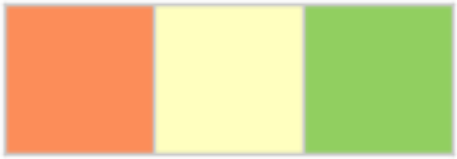
Spectral - 10



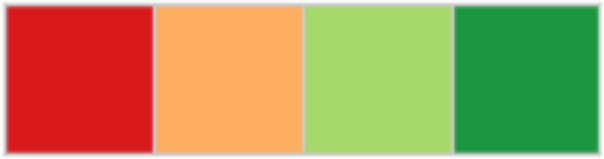
Spectral - 11



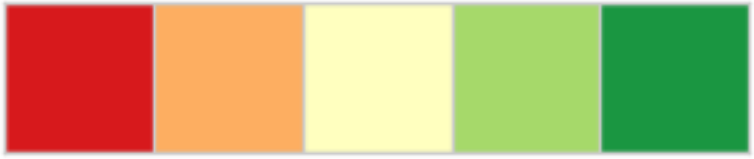
RdYlGn - 3



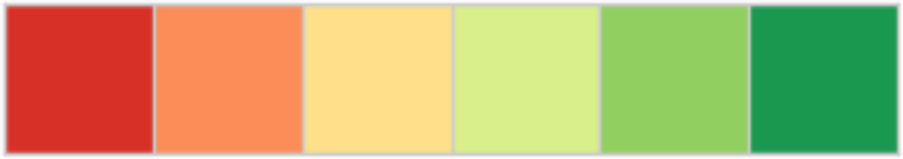
RdYlGn - 4



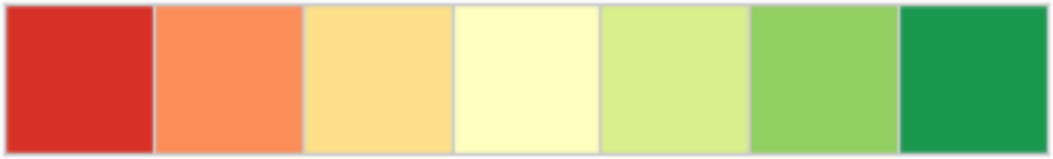
RdYlGn - 5



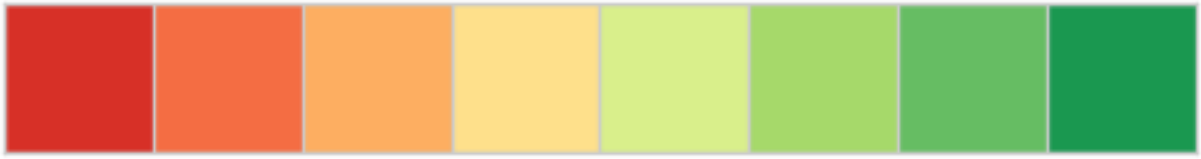
RdYlGn - 6



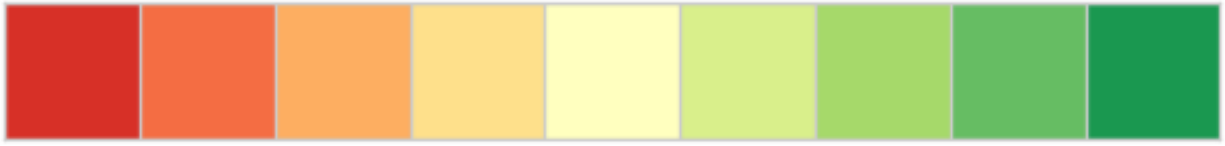
RdYlGn - 7



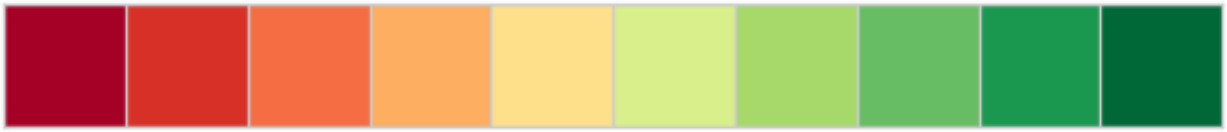
RdYlGn - 8



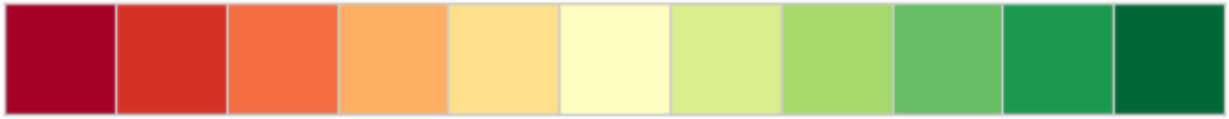
RdYlGn - 9



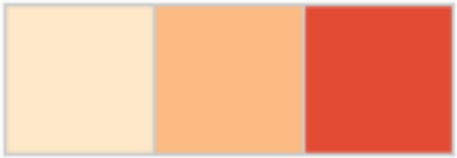
RdYlGn - 10



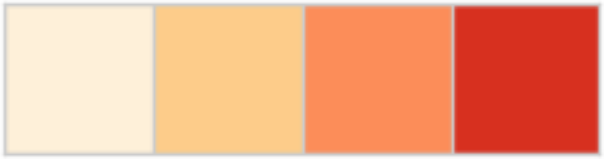
RdYlGn - 11



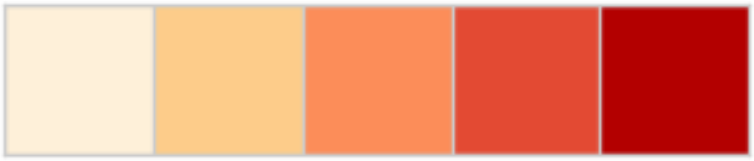
OrRd - 3



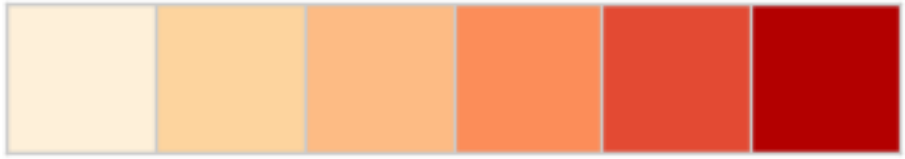
OrRd - 4



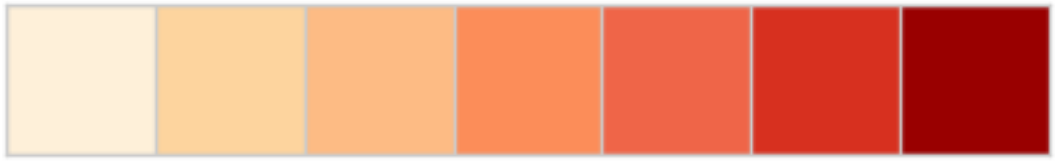
OrRd - 5



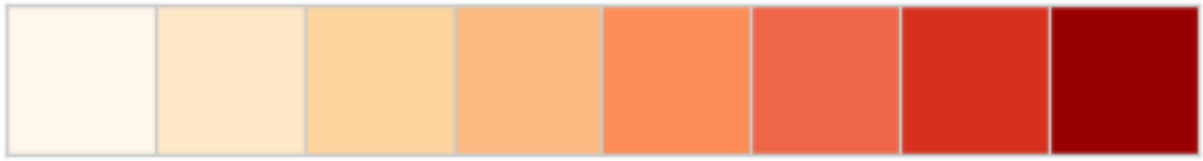
OrRd - 6



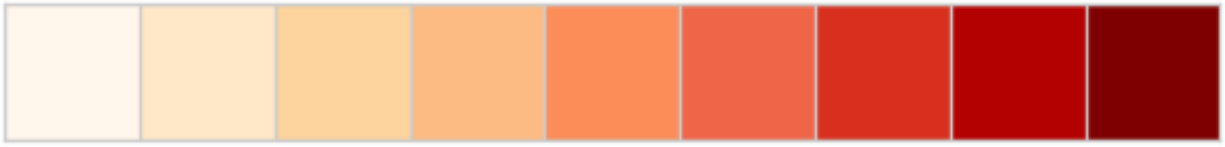
OrRd - 7



OrRd - 8



OrRd - 9



PuBu - 3

PuBu - 4

PuBu - 5

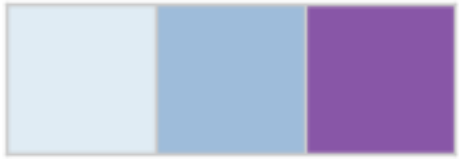
PuBu - 6

PuBu - 7

PuBu - 8

PuBu - 9

BuPu - 3



BuPu - 4

BuPu - 5

BuPu - 6

BuPu - 7

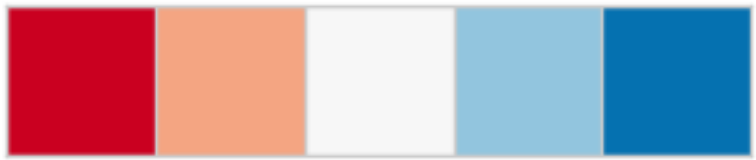
BuPu - 8

BuPu - 9

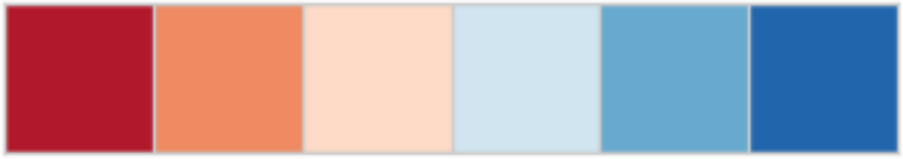
RdBu - 3

RdBu - 4

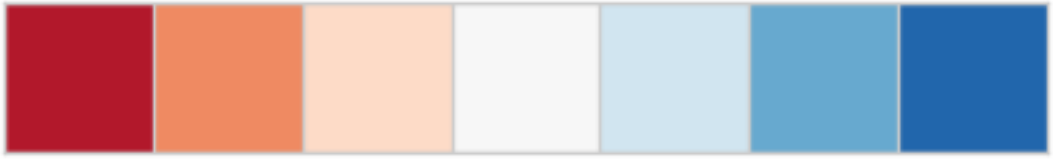
RdBu - 5



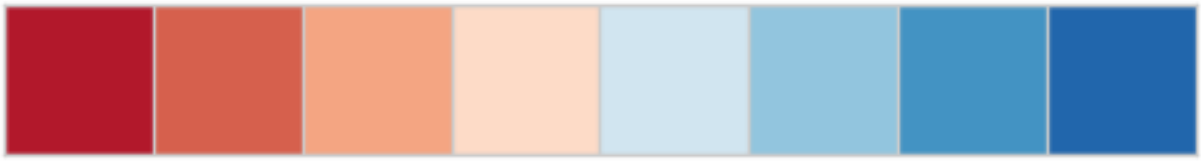
RdBu - 6



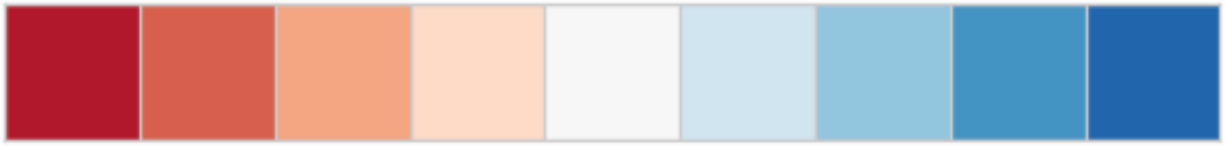
RdBu - 7



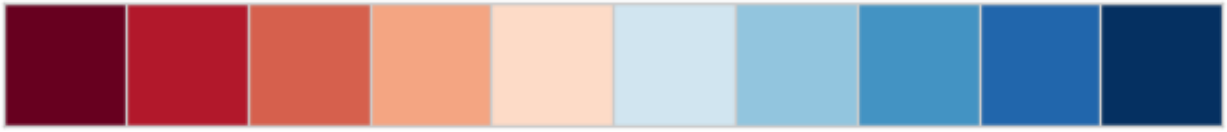
RdBu - 8



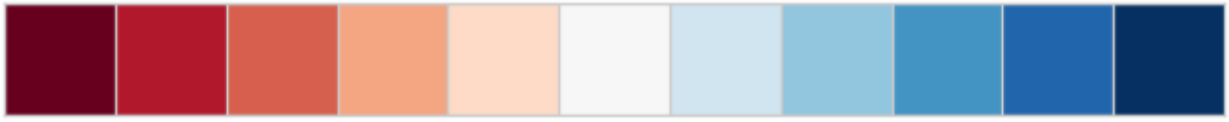
RdBu - 9



RdBu - 10



RdBu - 11



Oranges - 3

Oranges - 4

Oranges - 5

Oranges - 6

Oranges - 7

Oranges - 8

Oranges - 9

BuGn - 3

BuGn - 4

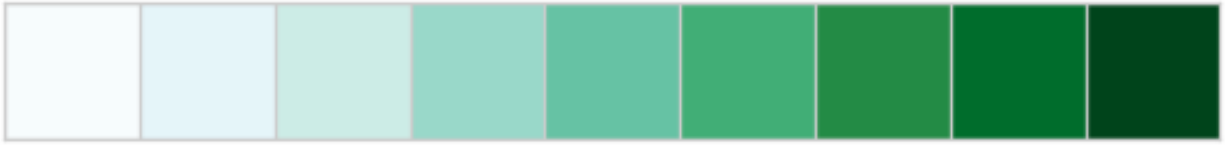
BuGn - 5

BuGn - 6

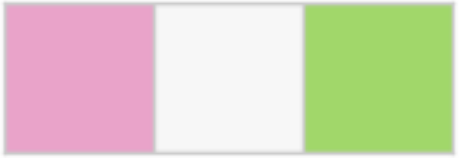
BuGn - 7

BuGn - 8

BuGn - 9



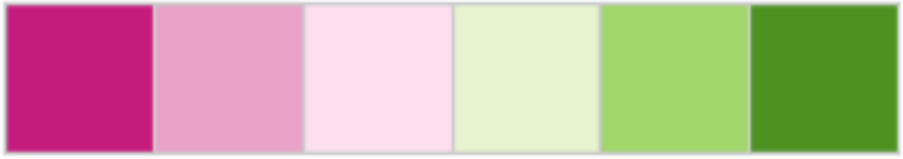
PiYG - 3



PiYG - 4

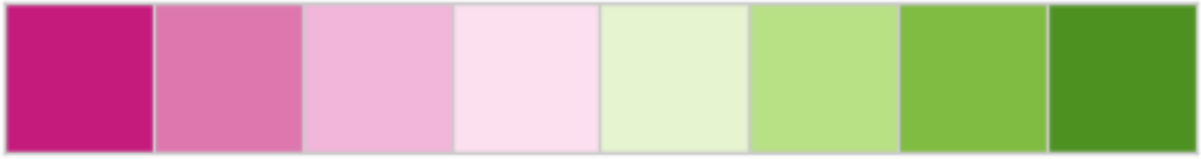
PiYG - 5

PiYG - 6

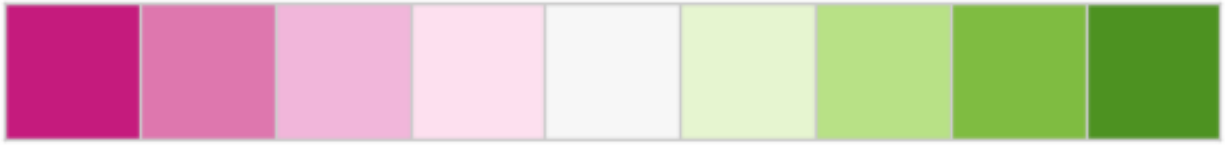


PiYG - 7

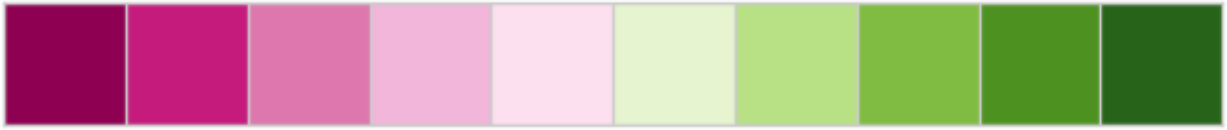
PiYG - 8



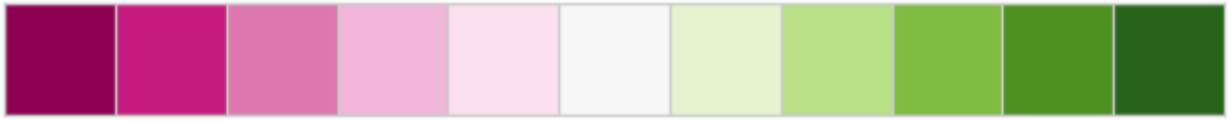
PiYG - 9



PiYG - 10

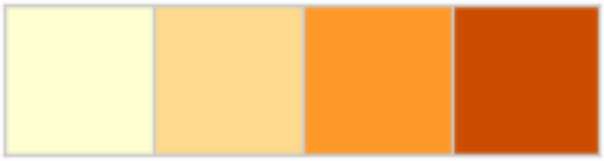


PiYG - 11



YlOrBr - 3

YlOrBr - 4



YlOrBr - 5

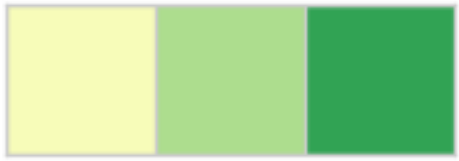
YlOrBr - 6

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

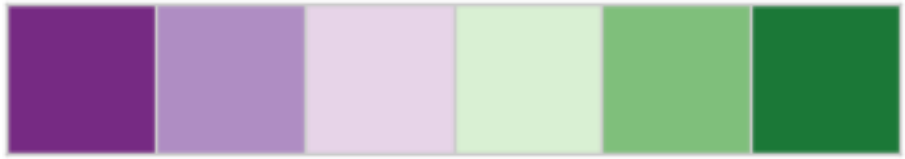
Greens - 9

PRGn - 3

PRGn - 4

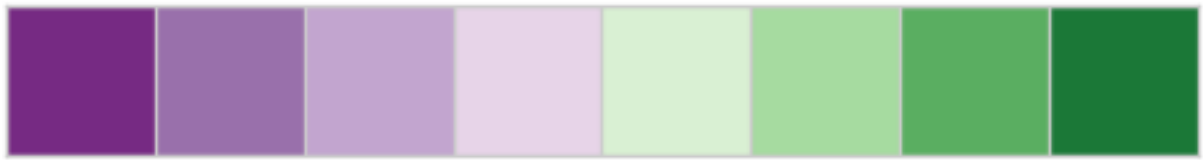
PRGn - 5

PRGn - 6



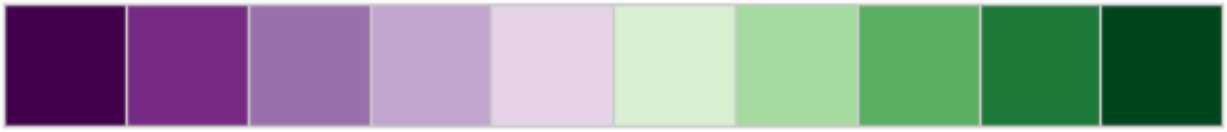
PRGn - 7

PRGn - 8

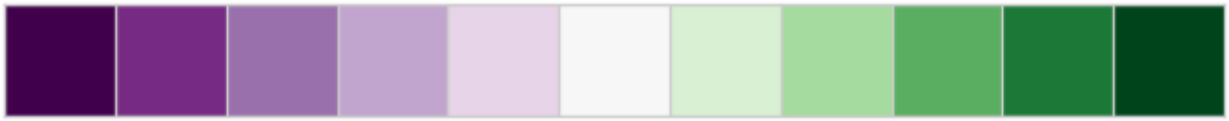


PRGn - 9

PRGn - 10



PRGn - 11



YlGnBu - 3

YlGnBu - 4

YlGnBu - 5

YlGnBu - 6

YlGnBu - 7

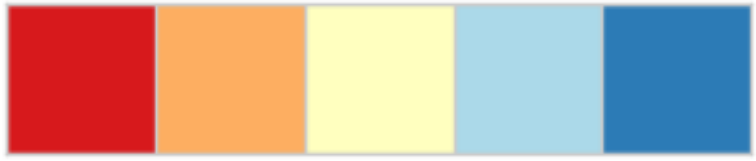
YlGnBu - 8

YlGnBu - 9

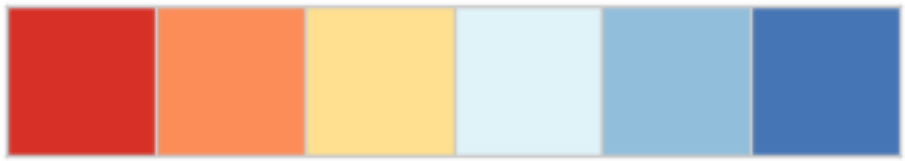
RdYlBu - 3

RdYlBu - 4

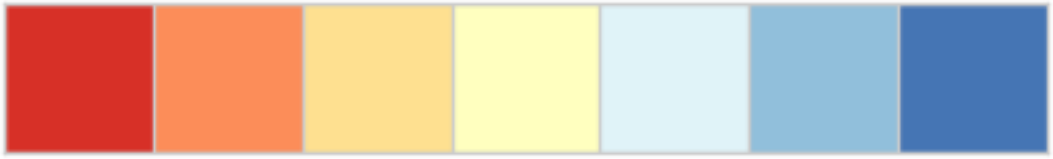
RdYlBu - 5



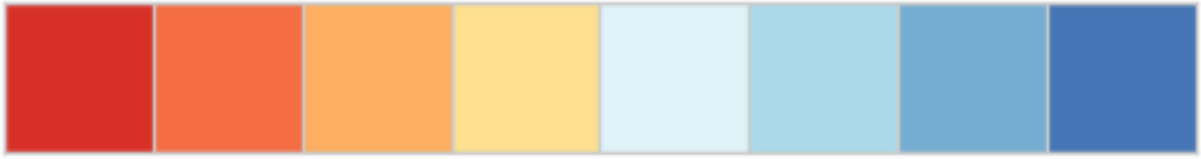
RdYlBu - 6



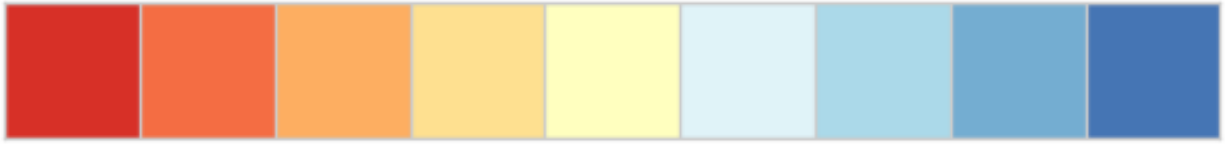
RdYlBu - 7



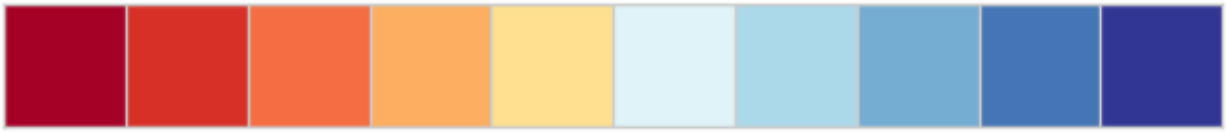
RdYlBu - 8



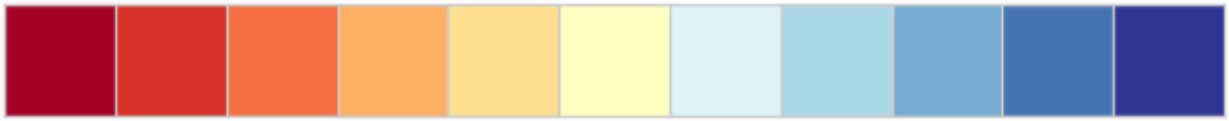
RdYlBu - 9



RdYlBu - 10

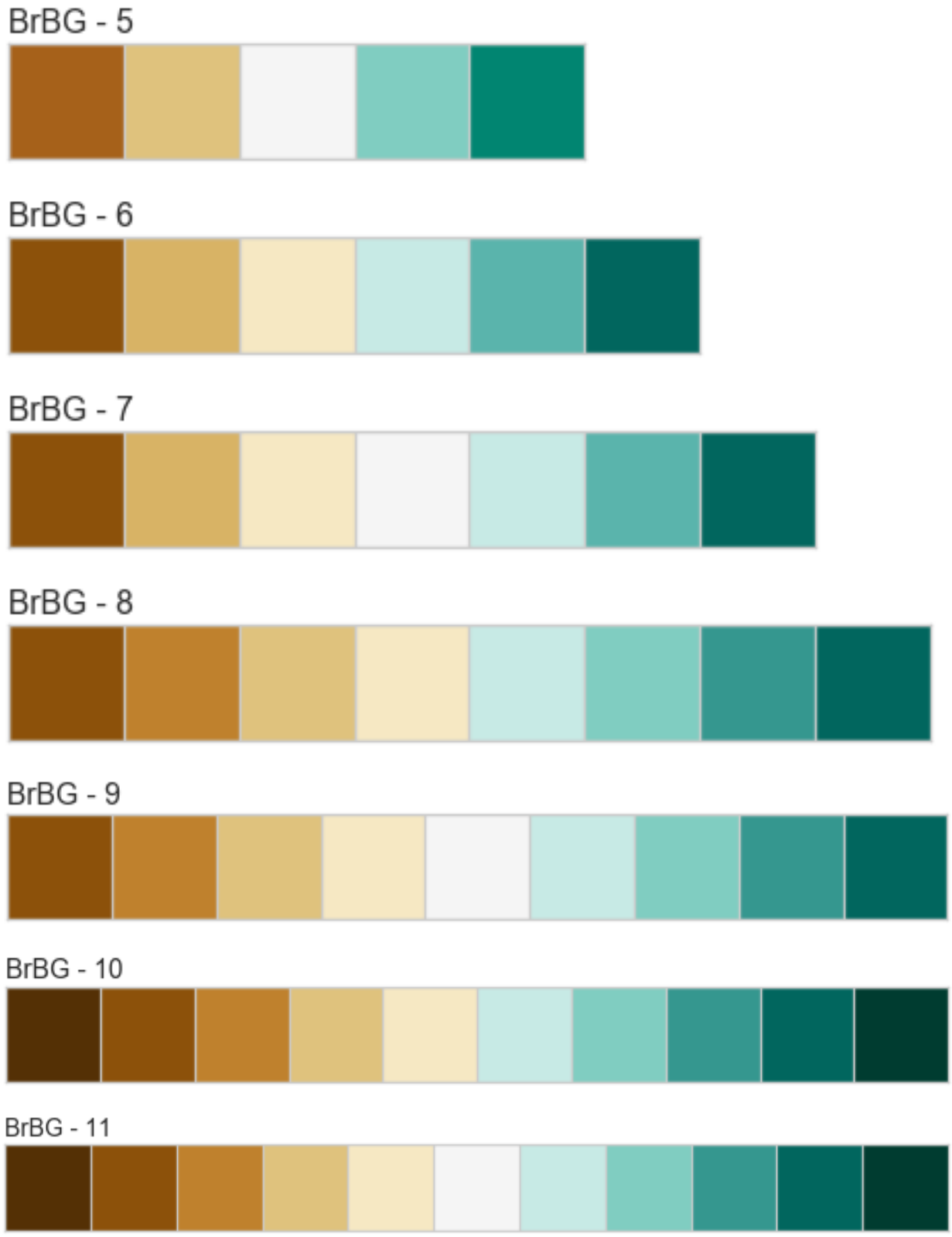


RdYlBu - 11



BrBG - 3

BrBG - 4


BrBG - 5
BrBG - 6
BrBG - 7
BrBG - 8
BrBG - 9
BrBG - 10
BrBG - 11

Purples - 3

Purples - 4

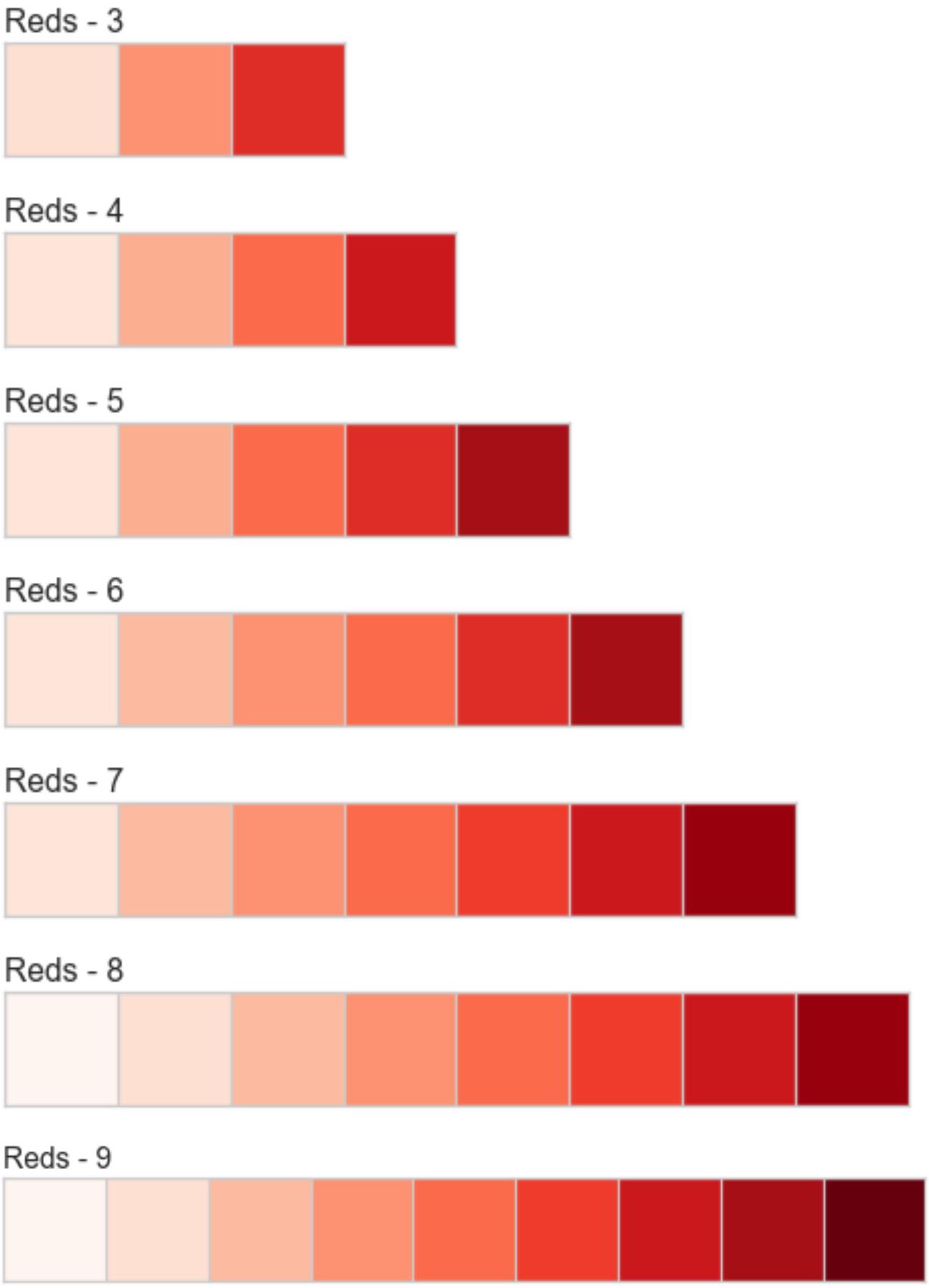
Purples - 5

Purples - 6

Purples - 7

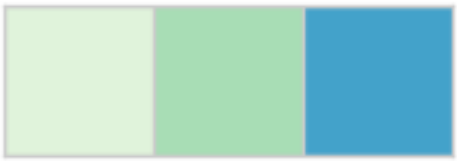
Purples - 8

Purples - 9


Reds - 3
Reds - 4
Reds - 5
Reds - 6
Reds - 7
Reds - 8
Reds - 9

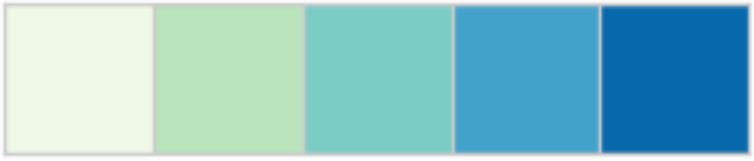
ddl_heat - 12

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

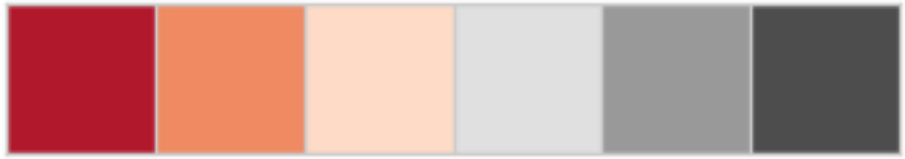
Greys - 9

RdGy - 3

RdGy - 4

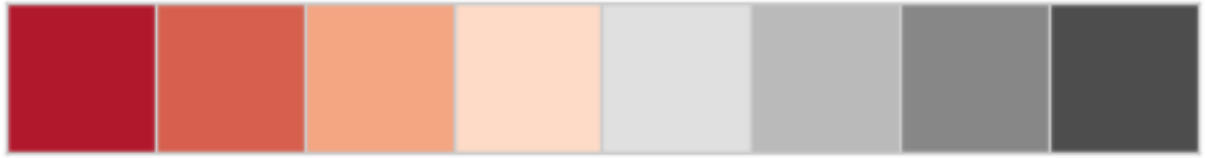
RdGy - 5

RdGy - 6



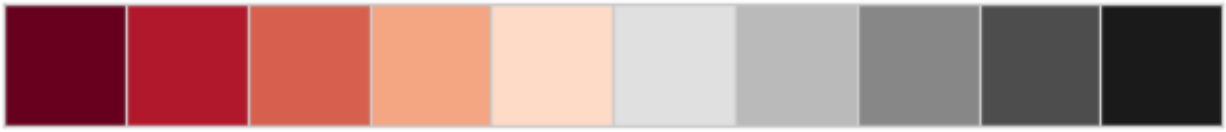
RdGy - 7

RdGy - 8

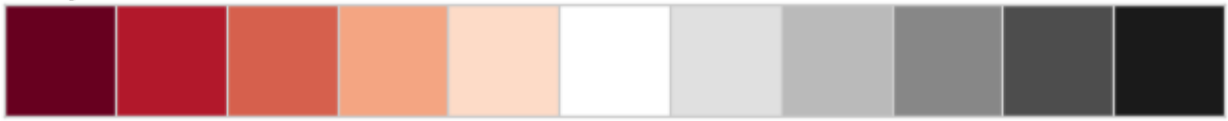


RdGy - 9

RdGy - 10

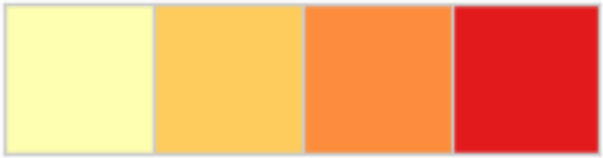


RdGy - 11



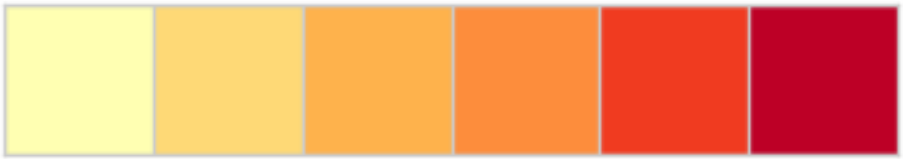
YlOrRd - 3

YlOrRd - 4



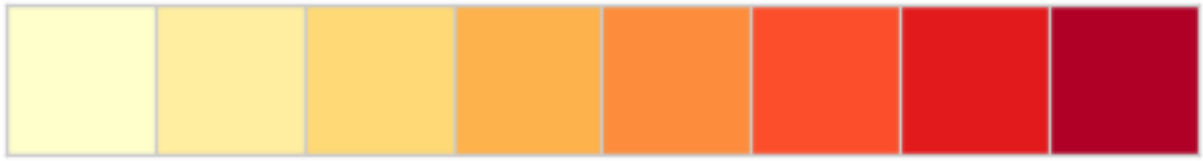
YlOrRd - 5

YlOrRd - 6



YlOrRd - 7

YlOrRd - 8



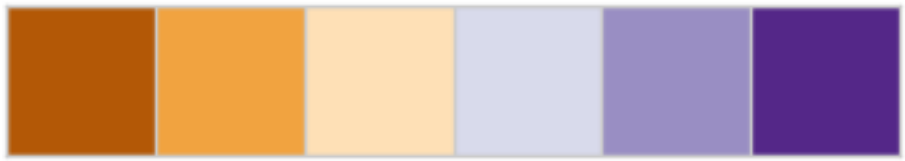
YlOrRd - 9

PuOr - 3

PuOr - 4

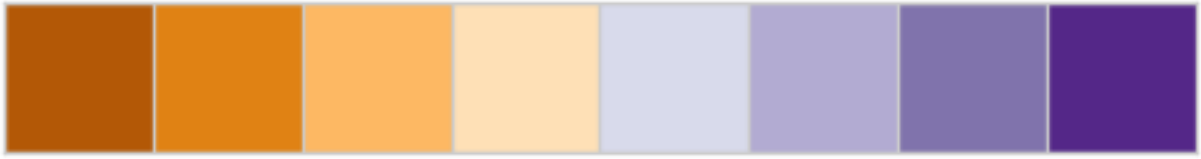
PuOr - 5

PuOr - 6

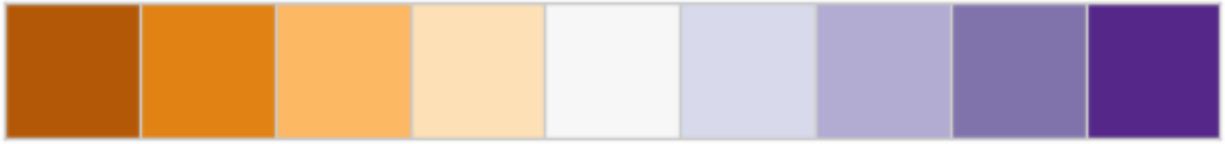


PuOr - 7

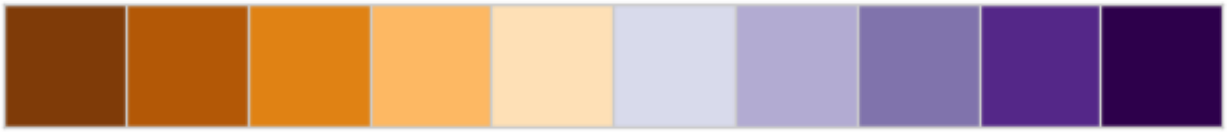
PuOr - 8



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

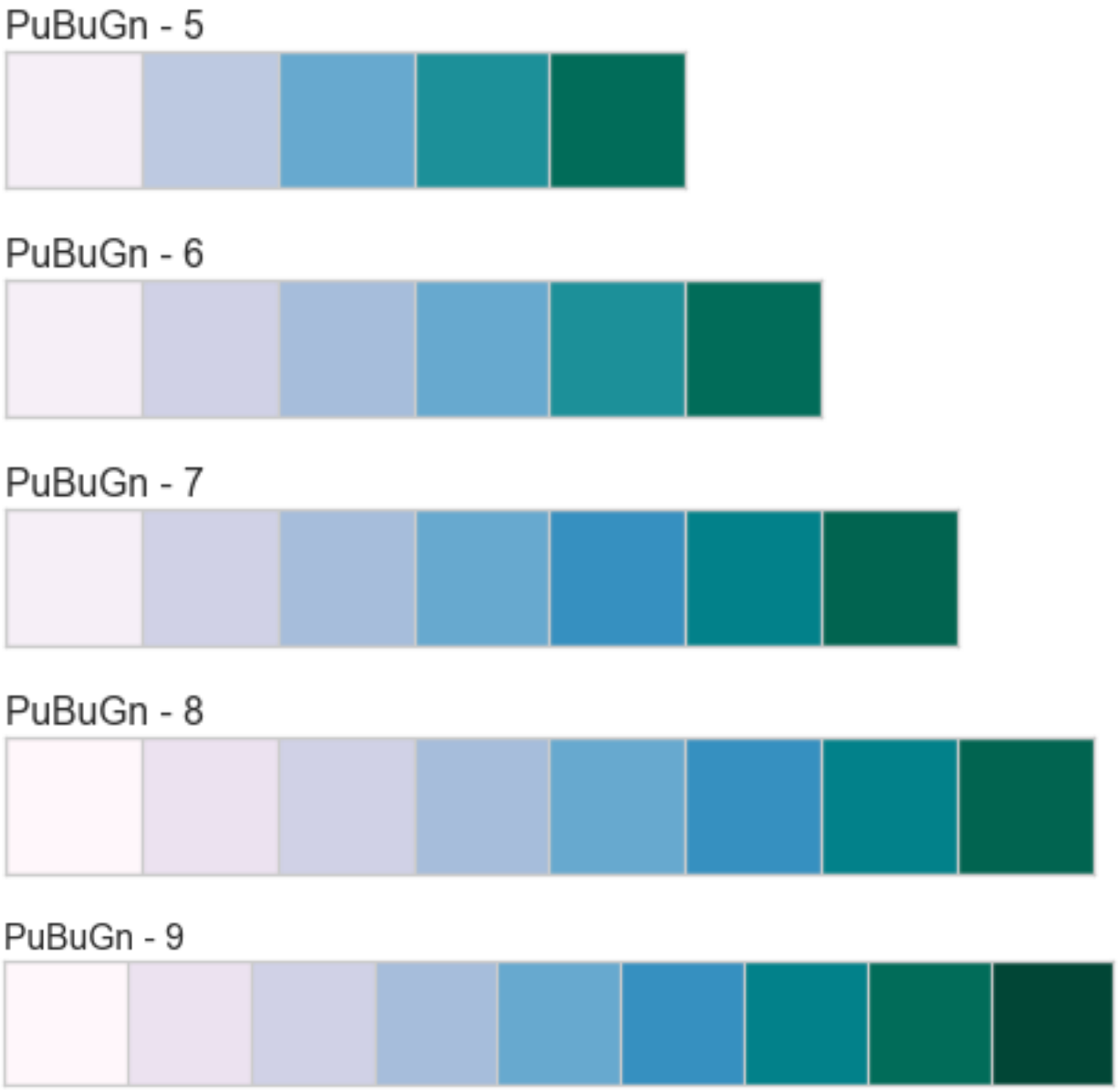
Blues - 7

Blues - 8

Blues - 9

PuBuGn - 3

PuBuGn - 4



API Referansı

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.

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, 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.

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.

.. seealso::

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.

Ayrıca bkz.:

`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.`**`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_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.

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.**reset_defaults**()
Restore all RC params to default settings.

yellowbrick.style.rcmod.**reset_orig**()
Restore all RC params to original settings (respects custom rc).

Not: Many examples utilize data from the UCI Machine Learning repository, in order to run the examples, make sure you follow the instructions in *Örnek Veri Setleri* to download and load 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 like styles, best fit lines, and anscombe's visualization can also be found in the links above.

4.4 Kullanıcı Testi Talimatları

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!

4.4.1 Step One: Questionaire

Please open the quesionaire, in order to familiarize yourself with the feedback that we are looking to receive. We are very interested in identifying any bugs in Yellowbrick. Please include al cells in your jupyter notebook that produce errors so that we may reproduce the problem.

4.4.2 Step Two: Dataset

Select a multivariate dataset of your own; the more (e.g. different) datasets that we can run through Yellowbrick, the more likely we'll discover edge cases and exceptions! Note that your dataset must be well-suited to modeling with Scikit-Learn. In particular we recommend you choose a dataset whose target is suited to 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!

4.4.3 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!

4.4.4 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!

4.4.5 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!

4.4.6 Step Six: Thanks!

Thank you for helping us make Yellowbrick better! We'd love to see pull requests for features you think would be extend 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!

4.5 Katkıda Bulunun

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 issues, mailing list, Stack Overflow, and elsewhere.
- 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 the Python Software Foundation Code of Conduct.

4.5.1 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. **Assign** yourself the task by pulling a card from our Waffle Kanban
4. **Fork** the repository into your own GitHub account.
5. Create a **Pull Request** first thing to connect with us about your task.
6. **Code** the feature, write the tests and documentation, add your contribution.
7. **Review** the code with core contributors who will guide you to a high quality submission.
8. **Merge** your contribution into the Yellowbrick codebase.

Not: Please create a pull request as soon as possible, even before you've started coding. This will allow the core contributors to give you advice about where to add your code or utilities and discuss other style choices and implementation details as you go. Don't wait!

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.

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
   ```

3. Install dependencies.

   Yellowbrick's dependencies are in the `requirements.txt` document at the root of the repository. Open this file and uncomment the dependencies that are for development only. Then install the dependencies with `pip`:

   ```
   $ pip install -r requirements.txt
   ```

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. 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. If you're going to take on a specific task, we'd strongly encourage you to check out the issue on Waffle and create a pull request *before you start coding* to better foster communication with other contributors. More on this in the next section.

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 essenence, 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.

Not: For a great example of a pull request for a new feature visualizer, check out this one by Carlo Morales.

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.

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.

4.5.2 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 *Efektif 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 objects in scikit-learn:

- **Transformers** take input data and return a new data set.
- **Estimators** are fit to training data and can make predictions.

The scikit-learn API is object oriented, and estimators and transformers 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 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 `poof()` method, which in turn calls a `finalize()` method on the visualizer to draw legends, titles, etc. and then `poof()` 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.
- `poof()`: 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 plot

from yellowbrick.base import Visualizer

class MyVisualizer(Visualizer):

    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):
        if self.ax is None:
            self.ax = self.gca()

        self.ax.plot(X)

    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.poof()
```

Score visualizers work on the same principle but accept an additional required `model` argument. Score visualizers wrap the model (which can be either instantiated or uninstantiated) and then pass through all attributes and methods through to the underlying model, drawing where necessary.

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 two primary helpers, `VisualTestCase` and `DatasetMixin`. Create your unittest as follows:

```
import pytest
from tests.base import VisualTestCase
from tests.dataset import DatasetMixin

class MyVisualizerTests(VisualTestCase, DatasetMixin):

    def test_my_visualizer(self):
        """
        Test MyVisualizer on a real dataset
```

(continues on next page)

(önceki sayfadan devam)

```
        """
        # Load the data from the fixture
        dataset = self.load_data('occupancy')

        # Get the data
        X = dataset[[
            "temperature", "relative_humidity", "light", "C02", "humidity"
        ]]
        y = dataset['occupancy'].astype(int)

        try:
            visualizer = MyVisualizer()
            visualizer.fit(X)
            visualizer.poof()
        except Exception as e:
            pytest.fail("my visualizer didn't work")
```

Tests can be run as follows:

```
$ make test
```

The Makefile uses the pytest runner and testing suite as well as the coverage library, so make sure you have those dependencies installed! The `DatasetMixin` also requires requests.py to fetch data from our Amazon S3 account.

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 unittest located in `tests/test_regressor/test_myvisualizer.py` as follows:

```
from tests.base import VisualTestCase
...
    def test_my_visualizer_output(self):
        ...
        visualizer = MyVisualizer()
        visualizer.fit(X)
        visualizer.poof()
        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`.

Documentation

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):
    """
    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
    ----------

    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.

    Examples
    --------

    >>> model = MyVisualizer()
    >>> model.fit(X)
    >>> model.poof()

    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!

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. 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 two primary files that need to be created:

1. **mymodule.rst**: the reStructuredText document
2. **mymodule.py**: a python file that generates images for the rst document

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
=============

Intro to my visualizer

.. code:: python

    # Example to run MyVisualizer
    visualizer = MyVisualizer(LinearRegression())

    visualizer.fit(X, y)
    g = visualizer.poof()


.. image:: images/my_visualizer.png

Discussion about my visualizer


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 `mymodule.py` to generate the images into the local directory's `images` subdirectory). 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.

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.

4.5.3 Advanced Development

In this section we discuss more advanced contributing guidelines including setting up branches for development as well as the release cycle. 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.

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.

Not: All pull requests should be into the `yellowbrick/develop` branch from your forked repository.

You can work directly in your fork and create a pull request from your fork's develop branch into ours. 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)
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. Then, when reviewed and approved, you can merge your fork into our main branch. Make sure to use the "Squash and Merge" option in order to create a Git history that is understandable.

Not: When merging a pull request, use the "squash and merge" option.

Core contributors have write access to the repository. In order to reduce the number of merges (and merge conflicts) we recommend that you utilize a feature branch off of develop to do intermediate work in:

```
$ git checkout -b feature-myfeature develop
```

Once you are done working (and everything is tested) merge your feature into develop.:

```
$ git checkout develop
$ git merge --no-ff feature-myfeature
$ git branch -d feature-myfeature
$ git push origin develop
```

Head back to Waffle and checkout another issue!

Releases

When ready to create a new release we branch off of develop as follows:

```
$ git checkout -b release-x.x
```

This creates a release branch for version x.x. 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. 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.

Once the release is ready for prime-time, merge into master:

```
$ git checkout master
$ git merge --no-ff --no-edit release-x.x
```

Tag the release in GitHub:

```
$ git tag -a vx.x
$ git push origin vx.x
```

You'll have to go to the release page to edit the release with similar information as added to the changelog. Once done, push the release to PyPI:

```
$ make build
$ make deploy
```

Check that the PyPI page is updated with the correct version and that `pip install -U yellowbrick` updates the version and works correctly. Also check the documentation on PyHosted, ReadTheDocs, and on our website to make sure that it was correctly updated. Finally merge the release into develop and clean up:

```
$ git checkout develop
$ git merge --no-ff --no-edit release-x.x
$ git branch -d release-x.x
```

Hotfixes and minor releases also follow a similar pattern; the goal is to effectively get new code to users as soon as possible!

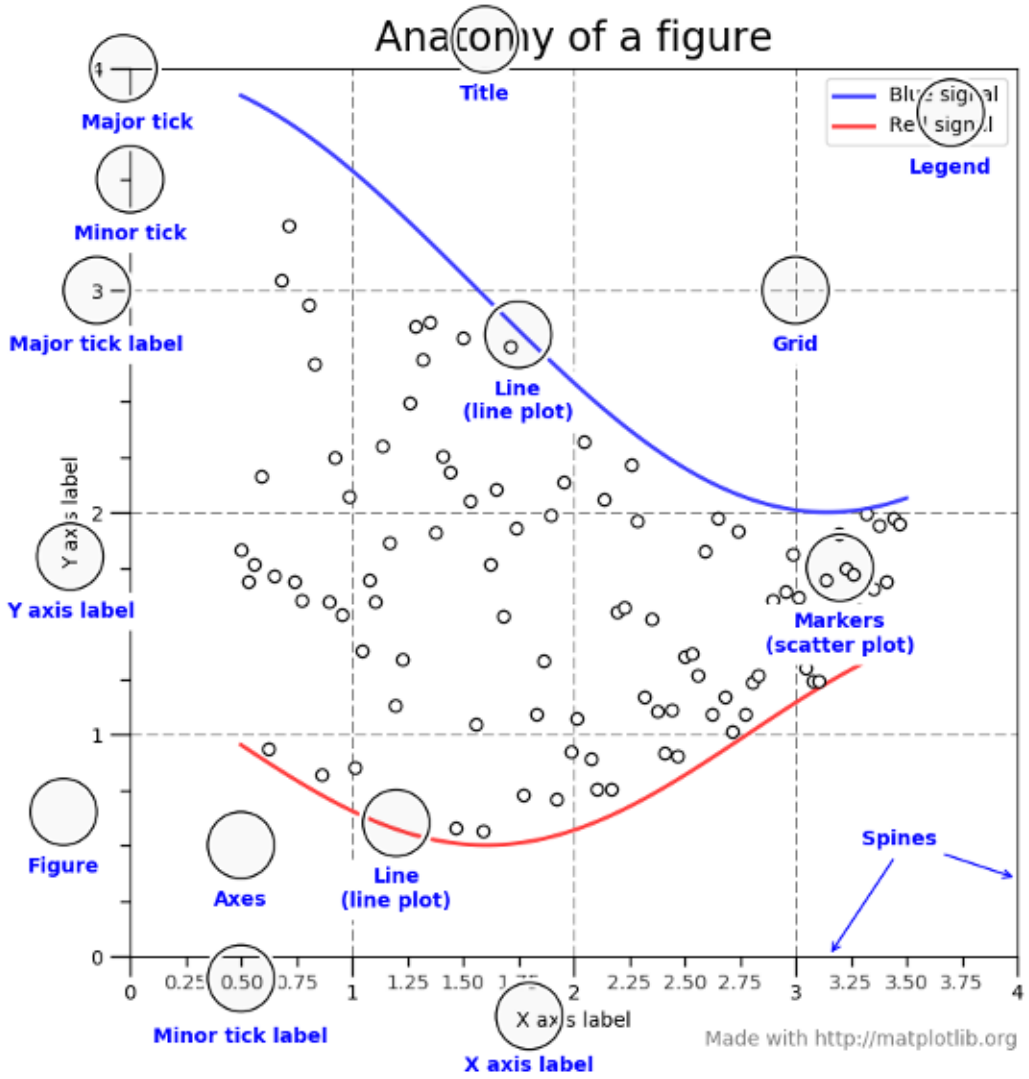
4.6 Efektif Matplotlib

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! Additionally we recommend Nicolas P. Rougier's Matplotlib tutorial for an in-depth dive.

4.6.1 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:

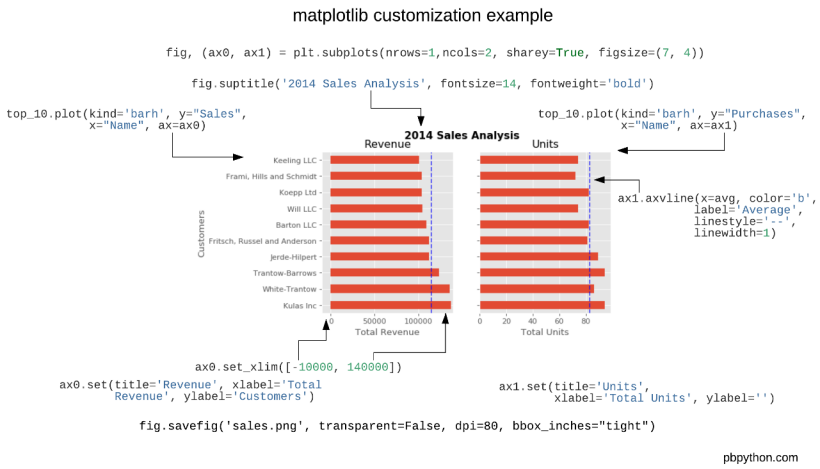
```
from sklearn.linear_model import LinearRegression
from yellowbrick.regressor import PredictionError

# 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.

4.6.2 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])
```

(continues on next page)

(önceki sayfadan devam)

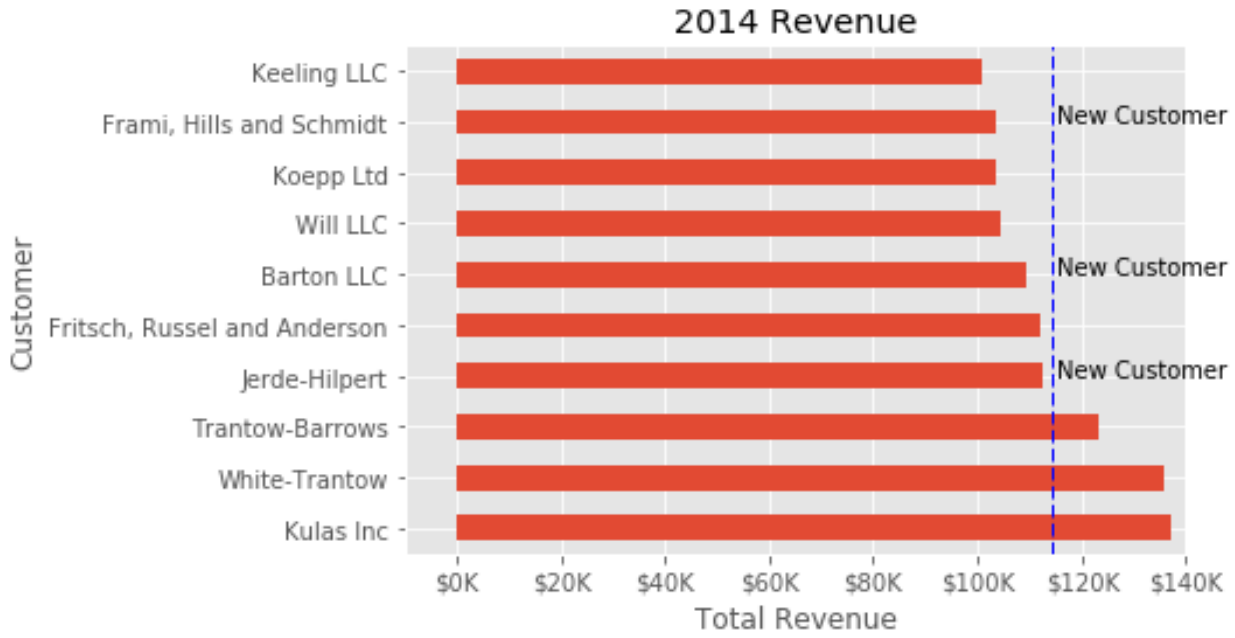
```
ax.set(title='2014 Revenue', xlabel='Total Revenue', ylabel='Customer')
formatter = FuncFormatter(currency)
ax.xaxis.set_major_formatter(formatter)
ax.legend().set_visible(False)
```

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])
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
```

(continues on next page)

(önceki sayfadan devam)

```
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.

4.7 Hakkında

Resim sahibi QuatroCinco, izniyle kullanılmıştır, Flickr Creative Commons.

Yellowbrick, görsel analiz ve tanımlama araçları ile birlikte Scikit-Learn API 'yi genişleten açık kaynak kodlu sade bir Python Projesidir. Yellowbrick API, Matplotlib'i de kapsayarak yayınlamaya hazır figür ve interaktif veri keşfi oluşturmayı sağlarken bir yandan da geliştiricilere ayrıntılı figür kontrolu imkanı sağlamaktadır. Yellowbrick kullanıcılar için; Makine öğrenimi modellerinin performansını geliştirme, güvenirliliğini sağlama, değer tahmininde bulunma ve makine oğrenmesi iş akışında karşılaşılan problemleri teşhis etmede yardımcı olabilmektedir.

Son dönemlerde makine öğrenimi iş akışının büyük bir kısmı; grid search yöntemi, standartlaştırılmıs API ler ve GUI tabanlı uygulamalar yoluyla otomatize edilmiştir. Bununla birlikte, pratikte insan sezgisi ve rehberliği, kaliteli modeller üzerinde detaylı arama yöntemlerine göre daha efektif odaklanma sağlamaktadir. Görsel model seçim işlemi yoluyla, veri bilimcileri; hatalara ve yanılgılara düşmeden finale, açıklanabilir modellere doğru ilerleyiş gösterebilmektedir.

Yellowbrick kütüphanesi, makine öğrenimi için veri bilimcilerine model seçim sürecine yön vermelerine olanak sağlayan bir tanı görselleştirme platformudur. Yellowbrick, Scikit Learn API'sini yeni bir temel obje ile genişletmiştir: Görselleştirici. Görselleştiriciler, çok boyutlu verilerin dönüşümü sırasında görsel tanılar sunarak, Scikit-Learn işlem sürecinin bir parçası olarak görsel modellerin uymasını ve dönüşümünü sağlamaktadır.

4.7.1 Model Seçimi

Makine öğrenimi tartışmaları sık sık model seçimi üzerine tekil odaklanma ile karakterize edilir. Gerek lojistik regresyon, karar ağaçları, Bayesian methodları veya yapay sinir ağları olsun; makine öğrenmesi uygulayıcıları tercihlerini

genellikle hızlı bir şekilde açıklarlar. Bunun nedeni çoğunlukla tarihseldir. Modern üçüncü parti makine öğrenimi kütüphaneleri birçok modelin yayılmasını önemsiz olarak gösterse de, geleneksel olarak bu algoritmalardan birinin bile uygulaması ve ayarlanması yıllar süren çalışma gerektirmiştir. Sonuç olarak makine öğrenmesi uygulayıcıları diğerlerine göre daha belirgin (ve muhtemelen daha yaygın olan) algoritmaları daha çok tercih etmeye yönelmiştir.

Bununla birlikte, model seçimi basit şekilde "doğru" ya da "yanlış" algoritmayı seçmekten biraz daha nüanslıdır. Pratik olarak iş akışı şunları içermektedir:

1. en küçük ve en kestirici tahmin kümesi seçimi ya da oluşturumu
2. bir dizi algoritmaların bir model ailesinden seçimi ve
3. performans optimizesi için algoritma hiperparametlerinin ayarlanması

Kumar et al tarafından **model seçim üçlüsü** 2015 yılı SIGMOD makalesinde ilk defa tanımlanmıştır. Makale içerisinde, tahmin edici modelleme öngörüsü için inşaa edilen yeni nesil veritabanı sistemlerinin gelişimiyle ilgili olarak, makale yazarları pratikte makine öğreniminin büyük ölçüde deneysel yapısı sebebiyle bu tür sistemlere çok fazla ihtiyaç olduğunu ifade ederler. "Model seçimini," şu şekilde açıklarlar, "tekrarlayıcı ve keşifseldir çünkü [model seçim üçlüsü] alanı genellikle sonsuzdur ve analizçiler için yeterli doğruluk ve kavrayış sağlayabilecek bir olası [kombinasyon] bilmek genelde imkansızdır."

4.7.2 İsim Kökeni

Yellowbrick Paketi, Amerikan yazar L. Frank Baum tarafından 1900 yılında yazılan çocuk romanı **The Wonderful Wizard of Oz** (Oz Büyücüsü) içerisinde geçen kurgusal bir elementten adını almıştır. Kitapta, Roman kahramanı Dorothy Gale, Emerald Şehri'ndeki hedefine ulaşabilmesi için sarı tuğlalı yolu (yellowbrick) takip etmesi gerekmektedir.

Wikipedia 'da şu şekilde geçmektedir: "Bu yol ilk olarak Oz Büyücüsü'nün üçüncü bölümünde tanıtılmıştır. Yol, Oz Diyarı'nin doğu çeyreğinin kalbinde bulunan Munchkin ülkesinden başlamaktadır. Bu yol; takip eden herkese,en son varış yeri —kıtanın tam ortasında bulunan Oz emparyel başkenti olan Emerald şehrine kadar rehberlik etmektir. Kitapta, romanın ana karakteri Dorothy, büyücüyü aramaya başlamadan önce bu yolu bulması gerekmektedir. Bunun sebebi çeşitli film adaptasyonlarında gösterilenin aksine, Kansas'ta çıkan hortumun çiftlik evini bu yola yakın biryerde bırakmamasıdır. Munchkins yerlileriyle ve onların sevilen dostu Kuzey'in iyi kalpli cadısı ile yapılan konsey sonrasında, Dorothy bu yolu aramaya başlar ve yakınlarında birçok patikalar ve yolları görür, (Herbiri çeşitli yönlere gitmekte). Neyseki parlak sarı tuğlalarla kaplanmış yolu bulması çok uzun sürmez."

4.7.3 Yellowbrick Ekibi

Yellowbrick açık kaynağa inanan veri bilimcileri tarafından geliştirilmiş ve tüm dünyadan Python geliştiricilerinin projeye katkıları memnuniyet vermiştir. Proje @rebeccabilbro ve @bbengfort tarafından makine öğrenmesi kavramlarını öğrencilerine daha iyi açıklamak amacıyla başlatılmış olup; bununla birlikte görsel yönlendirme potansiyelinin pratik veri bilimi üzerinde büyük bir etkiye sebep olabileceğini ve bunu yüksek düzey Python kütüphanesiyle gerçekleştirebileceklerini çok çabuk farkettiler.

Yellowbrick, iş birliğine ve açık kaynak gelişimine hizmet eden bir organizasyon olan District Data Labs tarafından üretilmiştir. District Data Labs'ın bir parçası olan Yellowbrick, ilk olarak PyCon 2016 da konuşmalarda ve geliştirme sprintlerinde Python topluluğuna tanıştırılmıştır. Daha sonra bu proje, DDL Araştırma Laboratuvarları (DDL topluluğu üyelerinin çeşitli veri projeleri ile katkıda bulunduğu Sömestir - Uzun Sprintler) ile sürdürülmüştür.

4.7.4 Lisans

Yellowbrick açık kaynaklı bir projedir ve lisansı FOSS Apache 2.0 uygulaması olup Apache Software Foundation lisanslı bir uygulamasıdır. Sade bir dil ile ifade etmek istersek Yellowbrick'i ticari amaçlı kullanabilir, kaynak kodu

değiştirilebilir ve paylaşabilirsiniz, hatta alt lisans edinebilirsiniz. Bizler Yellowbrick'i kullanmanızı, yararlanmanızı ve Yellowbrick'le ilgili güzel şeyler yaparsanız geri katkıda bulunmanızı isteriz.

Bununla birlikte sizden talep ettğimiz birkaç gereksinim bulunmakta. Öncelikle Yellowbrick kaynak kodunu dağıtırken veya paylaşırken yazılım depomuzun dizininde bulunan ve içerisinde telif hakkımız ve lisansımızın bulunduğu LICENSE.txt dosyasını lütfen dahil edin. Ek olarak projemiz içerisinde "NOTICE" dosyası oluşturmuşsak ayrıca bu dosyayı da kaynak paylaşım dosyanıza eklemeniz gerekmektedir. "NOTICE" dosyası bu projeye emek vermiş kişilere yönelik atıfların ve teşekkürlerin bulunduğu bir dosya olacaktır. Son olarak yazılımımızı kullanımınıza yönelik hiçbir şekilde District Data Labs veya Yellowbrick'e katkıda bulunan kişileri sorumlu olarak tutamazsınız ve yine aynı şekilde isimlerimizi, logolarımızı, markamızı mesul tutamazsınız.

Bu gereksinimlerin adil olduğuna düşünüyoruz ve açık kaynağa gerçekten inanıyoruz. Eğer yazılımımız üzerinde değişiklik yaparsanız, ticari veya akademide kullanın ya da başka bir ilginiz varsa, bunu duymaktan memnun oluruz.

4.7.5 Sunumlar

Yellowbrick, birkaç konferans ve sergilerde yer almaktan memnun olmustur. Sunduğumuz videolar,konuşmalar ve sunumların Yellowbrick'i daha iyi anlamanıza yardımcı olacağına inanıyoruz.

Videolar:

- Visual Diagnostics for More Informed Machine Learning: Within and Beyond Scikit-Learn (PyCon 2016)
- Visual Diagnostics for More Informed Machine Learning (PyData Carolinas 2016)
- Yellowbrick: Steering Machine Learning with Visual Transformers (PyData London 2017)

Slaytlar:

- Visualizing the Model Selection Process
- Visualizing Model Selection with Scikit-Yellowbrick
- Visual Pipelines for Text Analysis (Data Intelligence 2017)

4.8 Değişiklik Kayıtları

4.8.1 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 Decision Boundary 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 that takes into account only a single feature at a time (e.g. histogram analysis).

- Improved Prediction Error Plot with identity line, shared limits, and r squared.
- Updated FreqDist Visualizer to make word features a hyperparameter.
- Added normalization and scaling to Parallel Coordinates.
- 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 JointPlot Visualizer

4.8.2 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

4.8.3 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 K-Means 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

4.8.4 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

4.8.5 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.

4.8.6 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

4.8.7 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.

4.8.8 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

4.8.9 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

4.8.10 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

BÖLÜM 5

Dizinler ve Tablolar

- genindex
- modindex

Python Modül Dizini

y

Dizin

G

J

K

M

N

P

R

S

T

Y