# Contents

## 1 Visualizers
1.1 Feature Visualization ......................................................... 3
1.2 Classification Visualization .................................................. 3
1.3 Regression Visualization ...................................................... 4
1.4 Clustering Visualization ....................................................... 4
1.5 Model Selection Visualization ............................................... 4
1.6 Target Visualization .......................................................... 4
1.7 Text Visualization .............................................................. 4

## 2 Getting Help

## 3 Open Source

## 4 Table of Contents
4.1 Quick Start ................................................................. 9
4.2 Model Selection Tutorial .................................................... 17
4.3 Visualizers and API .......................................................... 32
4.4 User Testing Instructions ................................................... 211
4.5 Contributing ................................................................. 212
4.6 Effective Matplotlib .......................................................... 222
4.7 Yellowbrick for Teachers .................................................... 228
4.8 Gallery ......................................................................... 229
4.9 About ........................................................................... 231
4.10 Frequently Asked Questions ............................................... 234
4.11 Code of Conduct ............................................................ 237
4.12 Changelog ................................................................. 237

## 5 Indices and tables

Python Module Index ............................................................. 251
Yellowbrick is a suite of visual diagnostic tools called “Visualizers” that extend the Scikit-Learn API to allow human steering of the model selection process. In a nutshell, Yellowbrick combines scikit-learn with matplotlib in the best tradition of the scikit-learn documentation, but to produce visualizations for your models! For more on Yellowbrick, please see the About.

If you’re new to Yellowbrick, checkout the Quick Start or skip ahead to the Model Selection Tutorial. Yellowbrick is a rich library with many Visualizers being added on a regular basis. For details on specific Visualizers and extended usage head over to the Visualizers and API. Interested in contributing to Yellowbrick? Checkout the contributing guide. If you’ve signed up to do user testing, head over to the User Testing Instructions (and thank you!).
Visualizers are estimators (objects that learn from data) whose primary objective is to create visualizations that allow insight into the model selection process. In Scikit-Learn terms, they can be similar to transformers when visualizing the data space or wrap an model estimator similar to how the “ModelCV” (e.g. RidgeCV, LassoCV) methods work. The primary goal of Yellowbrick is to create a sensical API similar to Scikit-Learn. Some of our most popular visualizers include:

### 1.1 Feature Visualization

- **Rank Features**: pairwise ranking of features to detect relationships
- **Parallel Coordinates**: horizontal visualization of instances
- **Radial Visualization**: separation of instances around a circular plot
- **PCA Projection**: projection of instances based on principal components
- **Manifold Visualization**: high dimensional visualization with manifold learning
- **Feature Importances**: rank features by importance or linear coefficients for a specific model
- **Recursive Feature Elimination**: find the best subset of features based on importance
- **Joint Plots**: direct data visualization with feature selection

### 1.2 Classification Visualization

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

### 1.3 Regression Visualization

• **Prediction Error Plot**: find model breakdowns along the domain of the target
• **Residuals Plot**: show the difference in residuals of training and test data
• **Alpha Selection**: show how the choice of alpha influences regularization

### 1.4 Clustering Visualization

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

### 1.5 Model Selection Visualization

• **Validation Curve**: tune a model with respect to a single hyperparameter
• **Learning Curve**: show if a model might benefit from more data or less complexity

### 1.6 Target Visualization

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

### 1.7 Text Visualization

• **Term Frequency**: visualize the frequency distribution of terms in the corpus
• **t-SNE Corpus Visualization**: use stochastic neighbor embedding to project documents
• **Dispersion Plot**: visualize how key terms are dispersed throughout a corpus

... and more! Visualizers are being added all the time; be sure to check the examples (or even the develop branch) and feel free to contribute your ideas for new Visualizers!
Yellowbrick is a welcoming, inclusive project in the tradition of matplotlib and scikit-learn. Similar to those projects, we follow the Python Software Foundation Code of Conduct. Please don’t hesitate to reach out to us for help or if you have any contributions or bugs to report!

The primary way to ask for help with Yellowbrick is to post on our Google Groups Listserv. This is an email list/forum that members of the community can join and respond to each other; you should be able to receive the quickest response here. Please also consider joining the group so you can respond to questions! You can also ask questions on Stack Overflow and tag them with “yellowbrick”. Or you can add issues on GitHub. You can also tweet or direct message us on Twitter @scikit_yb.
The Yellowbrick license is an open source Apache 2.0 license. Yellowbrick enjoys a very active developer community; please consider joining them and contributing!

Yellowbrick is hosted on GitHub. The issues and pull requests are tracked there.
The following is a complete listing of the Yellowbrick documentation for this version of the library:

4.1 Quick Start

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

Yellowbrick has two primary dependencies: scikit-learn and matplotlib. If you do not have these Python packages, they will be installed alongside Yellowbrick. Note that Yellowbrick works best with scikit-learn version 0.18 or later and matplotlib version 2.0 or later. Both of these packages require some C code to be compiled, which can be difficult on some systems, like Windows. If you’re having trouble, try using a distribution of Python that includes these packages like Anaconda.

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

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

4.1.1 Installation

Yellowbrick is compatible with Python 2.7 or later, but it is preferred to use Python 3.5 or later to take full advantage of all functionality. The simplest way to install Yellowbrick is from PyPI with pip, Python’s preferred package installer.

```bash
$ 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.

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

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

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

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

**Warning:** 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 Using Yellowbrick

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

```python
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:

```python
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.

**Note:** 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:

```python
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:

```python
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 Walkthrough

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

**Note:** We have updated the dataset from the UCI ML repository to make it a bit easier to load into Pandas; make sure you download the Yellowbrick version of the dataset.

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

```python
import pandas as pd

data = pd.read_csv('bikeshare.csv')
X = data[
```
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:

```python
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 the features `temp` and `feelslike` have a strong correlation and also that the feature `season` has a strong correlation with the feature `month`. This seems to make sense; the apparent temperature we feel outside depends on the actual temperature and other air quality 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 because there appear to be some outliers in the dataset. These instances may need to be
manually removed in order to improve the quality of the final model because they may represent data input errors, and potentially train the model on a skewed dataset which would return unreliable model predictions. The first instance of outliers occurs in the temp data where the `feelslike` value is approximately equal to 0.25 - showing a horizontal line of data, likely created by input error.

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

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

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

```python
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 (the number of riders), 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 affected by the components of our model (the features). In this case, we can see that the lower predicted number of riders results in lower model error, and conversely that the higher predicted number of riders results in higher model error. This indicates that our model has more noise in certain regions of the target or that two variables are colinear, meaning that they are injecting error as the noise in their relationship changes.

The residuals plot also shows how the model is injecting error, the bold horizontal line at \( \text{residuals} = 0 \) 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 \( \text{actual} - \text{expected} \), this means that the expected value is bigger than the actual value most of the time; e.g., that our model is primarily guessing more than the actual number of riders. Moreover, there is a very interesting boundary along the top right of the residuals graph, indicating an interesting effect in model space; possibly that some feature is strongly weighted in the region of that model.

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

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

```python
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:

```python
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 Selection Tutorial. After that you can get up to speed on specific visualizers detailed in the Visualizers and API.

4.2 Model Selection Tutorial

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

4.2.1 The Model Selection Triple

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

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

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

The model selection triple was first described in a 2015 SIGMOD paper by Kumar et al. In their paper, which concerns the development of next-generation database systems built to anticipate predictive modeling, the authors cogently express that such systems are badly needed due to the highly experimental nature of machine learning in practice. “Model selection,” they explain, “is iterative and exploratory because the space of [model selection triples] is usually infinite, and it is generally impossible for analysts to know a priori which [combination] will yield satisfactory accuracy and/or insights.”

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

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

### 4.2.2 About the Data

This tutorial uses a modified version of the mushroom dataset from the UCI Machine Learning Repository. Our objective is to predict if a mushroom is poisonous or edible based on its characteristics.

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

Our file, “agaricus-lepiota.txt,” contains information for 3 nominally valued attributes and a target value from 8124 instances of mushrooms (4208 edible, 3916 poisonous).

Let’s load the data with Pandas.

```python
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()
```
4.2.3 Feature Extraction

Our data, including the target, is categorical. We will need to change these values to numeric ones for machine learning. In order to extract this from the dataset, we’ll have to use Scikit-Learn transformers to transform our input dataset into something that can be fit to a model. Luckily, Scikit-Learn does provide a transformer for converting categorical labels into numeric integers: sklearn.preprocessing.LabelEncoder. Unfortunately it can only transform a single vector at a time, so we’ll have to adapt it in order to apply it to multiple columns.

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

features = ['cap-shape', 'cap-surface', 'cap-color']
target = ['class']
X = dataset[features]
y = dataset[target]
for column, encoder in self.encoders.items():
    output[column] = encoder.transform(data[column])

return output

4.2.4 Modeling and Evaluation

Common metrics for evaluating classifiers

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

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

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

\[
\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \\
\text{recall} = \frac{\text{true positives}}{\text{false negatives} + \text{true positives}} \\
\text{F1 score} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}
\]

Now we’re ready to make some predictions!

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

```python
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!

```python
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
```

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

0.65846308387744845

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

0.6383842388991346

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

0.66251459711950167

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

0.69944182052382997

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

0.65802139037433149

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

0.65846308387744845

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

0.6582609897010799

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

0.687643484132343

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

0.6873648045448383

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

0.69317131158367451
Preliminary Model Evaluation

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

4.2.5 Visual Model Evaluation

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

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

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

Note: When running in a Jupyter Notebook, be sure to add the following line at the top of the notebook: `%matplotlib notebook`. This will ensure the figures are rendered correctly. For those running this code with a Python script, the figure should appear in a secondary window.

```python
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from yellowbrick.classifier import ClassificationReport

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)
    ])

    # Create a new figure to draw the classification report on
    _, ax = plt.subplots()

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

    # Note that to save the figure to disk, you can specify an outpath
    # argument to the poof method!
    visualizer.poof()

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

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

4.3 Visualizers and API

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

4.3.1 Example Datasets

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

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

This should create a folder named data in your current working directory that contains 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 notebook in Yellowbrick. Please reference this code when trying to load a specific data set:

```python
import os

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"),
    "spam": os.path.join(FIXTURES, "spam", "spam.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(
                f"'{name}' dataset has not been downloaded, use the download.py module to fetch datasets"
            ).format(name)

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

Unless otherwise specified, most of the examples currently use one or more of the listed datasets. 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
• **spam**: suitable for binary classification

### 4.3.2 Anscombe’s Quartet

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

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

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

**API Reference**

Plots Anscombe’s Quartet as an illustration of the importance of visualization.
yellowbrick.anscombe.anscombe()

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

### 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 the following feature analysis visualizers implemented:

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

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

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

#### RadViz Visualizer

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

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

If your data contains rows with missing values (`numpy.nan`), those missing values will not be plotted. In other words, you may not get the entire picture of your data. RadViz will raise a DataWarning to inform you of the percent missing.
If you do receive this warning, you may want to look at imputation strategies. A good starting place is the scikit-learn Imputer.

```python
# 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", "CO2", "humidity"]
classes = ["unoccupied", "occupied"]

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

# 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.
Implement radvz for feature analysis.

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

Bases: yellowbrick.features.base.DataVisualizer

RadViz is a multivariate data visualization algorithm that plots each axis 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.

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.
- **alpha** [float, default: 1.0] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.
- **kwargs** [dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

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

Examples

```python
>>> 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
alias of 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.

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

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

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

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.

```python
from yellowbrick.features import Rank1D

# 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
```
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-occurrence 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.

```python
from yellowbrick.features import Rank2D

# 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.

```python
# 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
```
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.
- `**kwags` [dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.
## Examples

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

### Attributes

- **ranks_** 
  
  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 0x7f3964d1aa60>}

### 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', 'spearman'}, 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

```python
>>> 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 0x7f3964d1ac80>, 'pearson': <function Rank2D.<lambda> at 0x7f3964d1abf8>, 'spearman': <function Rank2D.<lambda> at 0x7f3964d1ad08>}`

Parallel Coordinates

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

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

```python
# 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", "CO2", "humidity"]
classes = ["unoccupied", "occupied"]

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

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

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

The visualization can be drawn with either the `ParallelCoordinates` visualizer or using the `parallel_coordinates` quick method:

```python
from yellowbrick.features import ParallelCoordinates

# Instantiate the visualizer
visualizer = ParallelCoordinates(
    classes=classes, features=features, sample=0.5, shuffle=True
)

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

# Finalize the title and axes then display the visualization
visualizer.poof()
```
By inspecting the visualization closely, we can see that the combination of transparency and overlap gives us the sense of groups of similar instances, sometimes referred to as “braids”. If there are distinct braids of different classes, it suggests that there is enough separability that a classification algorithm might be able to discern between each class.

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

Normalization techniques can be directly applied to the visualizer without pre-transforming the data (though you could also do this) by using the normalize parameter. Several transformers are available; try using minmax, minabs, standard, 11, or 12 normalization to change perspectives in the parallel coordinates as follows:

```python
from yellowbrick.features import ParallelCoordinates

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

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

**Faster Parallel Coordinates**

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

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

The “fast” drawing mode vastly improves the performance of the parallel coordinates drawing algorithm by drawing each line segment by class rather than each instance individually. However, this improved performance comes at a cost, as the visualization produced is subtly different; compare the visualizations in fast and standard drawing modes below:
As you can see the “fast” drawing algorithm does not have the same build up of color density where instances of the same class intersect. Because there is only one line per class, there is only a darkening effect between classes. This can lead to a different interpretation of the plot, though it still may be effective for analytical purposes, particularly when you’re plotting a lot of data. Needless to say, the performance benefits are dramatic:
API Reference

Implementation of parallel coordinates for multi-dimensional feature analysis.

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

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

**Parameters**

- `ax` [matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).
- `features` [list, default: None] a list of feature names to use If a DataFrame is passed to fit and features is None, feature names are selected as the columns of the DataFrame.
- `classes` [list, default: None] a list of class names for the legend If classes is None and a y value is passed to fit then the classes are selected from the target vector.
**normalize** [string or None, default: None] specifies which normalization method to use, if any. Current supported options are ‘minmax’, ‘maxabs’, ‘standard’, ‘l1’, and ‘l2’.

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

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

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

**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.

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

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

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

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

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

### Attributes

- **n_samples** [int] number of samples included in the visualization object

### Notes

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

### Examples

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

NORMALIZERS = {'l1': Normalizer(copy=True, norm='l1'), 'l2': Normalizer(copy=True, norm='l2')}
**draw** *(X, y, **kwargs)*

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

**Parameters**

- **X** [ndarray of shape n x m] A matrix of n instances with m features
- **y** [ndarray of length n] An array or series of target or class values
- **kwargs** [dict] Pass generic arguments to the drawing method

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

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

**Parameters**

- **X** [ndarray of shape n x m] A matrix of n instances with m features
- **y** [ndarray of length n] An array or series of target or class values

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

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

This is the default method of drawing.

**Parameters**

- **X** [ndarray of shape n x m] A matrix of n instances with m features
- **y** [ndarray of length n] An array or series of target or class values

**Notes**

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

**finalize** (**kwargs**)

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 visualization since it has both the X and y data required for the viz and the transform method does not.

**Parameters**

- **X** [ndarray or DataFrame of shape n x m] A matrix of n instances with m features
- **y** [ndarray or Series of length n] An array or series of target or class values
- **kwargs** [dict] Pass generic arguments to the drawing method

**Returns**

- **self** [instance] Returns the instance of the transformer/visualizer
PCA Projection

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

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

# Specify the features of interest and the target
target = "default"
features = [col for col in data.columns if col != target]

# Extract the instance data and the target
X = data[features]
y = data[target]

# Create a list of colors to assign to points in the plot
colors = np.array(['r' if yi else 'b' for yi in y])

from yellowbrick.features.pca import PCADecomposition

visualizer = PCADecomposition(scale=True, color=colors)
visualizer.fit_transform(X, y)
visualizer.poof()
```

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

```python
visualizer = PCA Decomposition(scale=True, color=colors, proj_dim=3)
visualizer.fit_transform(X, y)
visualizer.poof()
```

**Biplot**

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

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

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

# Extract the instance data and the target
X = data[features]
y = data[target]
```
```python
visualizer = PCADecomposition(scale=True, proj_features=True)
visualizer.fit_transform(X, y)
visualizer.poof()

visualizer = PCADecomposition(scale=True, proj_features=True, proj_dim=3)
visualizer.fit_transform(X, y)
visualizer.poof()
```
Decomposition based feature visualization with PCA.

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

Bases: `yellowbrick.features.base.MultiFeatureVisualizer`

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

**Parameters**

- **ax** [matplotlib Axes, default: None] The axes to plot the figure on. If None is passed in the current axes will be used (or generated if required).
- **features** [list, default: None] 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.
- **scale** [bool, default: True] Boolean that indicates if user wants to scale data.
- **proj_dim** [int, default: 2] Dimension of the PCA visualizer.
- **proj_features** [bool, default: False] Boolean that indicates if the user wants to project the features in the projected space. If True the plot will be similar to a biplot.
- **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.

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

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

### Examples

```python
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
>>> y = iris.target
>>> visualizer = PCADecomposition()
>>> visualizer.fit_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 the PCA transformer, transforms the data in X, then draws the decomposition in either 2D or 3D space as a scatter plot.

**Parameters**

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

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

**Returns**

**self**: [visualizer] Returns self for use in Pipelines
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 \(X\).

**Manifold Visualization**

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

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

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

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

Manifolds can be used on many types of problems, and the color used in the scatter plot can describe the target instance. In an unsupervised or clustering problem, a single color is used to show structure and overlap. In a classification problem discrete colors are used for each class. In a regression problem, a color map can be used to describe points as a heat map of their regression values.

#### Discrete Target

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

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

# Specify the features of interest
features = ['temperature', 'relative humidity', 'light', 'CO2', 'humidity']

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

from yellowbrick.features.manifold import Manifold

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

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

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

model = Pipeline([  
    ("selectk", SelectKBest(k=3, score_func=f_classif)),  
    ("viz", Manifold(manifold='isomap', target='discrete')),  
])

# Load the classification dataset
data = load_data("occupancy")

# Specify the features of interest
features = [  
    "temperature", "relative humidity", "light", "CO2", "humidity"  
]

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

(continues on next page)
Continuous Target

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

```python
# Specify the features of interest
feature_names = ['cement', 'slag', 'ash', 'water', 'splat', 'coarse', 'fine', 'age']
target_name = 'strength'

# Get the X and y data from the DataFrame
X = data[feature_names]
y = data[target_name]

visualizer = Manifold(manifold='isomap', target='continuous')
visualizer.fit_transform(X, y)
visualizer.poof()
```
Use manifold algorithms for high dimensional visualization.

```python
class yellowbrick.features.manifold.Manifold(ax=None, manifold='lle', n_neighbors=10, colors=None, target='auto', alpha=0.7, random_state=None, **kwargs)
```

**Bases:** `yellowbrick.features.base.FeatureVisualizer`

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

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

<table>
<thead>
<tr>
<th>Manifold</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;lle&quot;</td>
<td>Locally Linear Embedding</td>
</tr>
<tr>
<td>&quot;ltsa&quot;</td>
<td>LTSA LLE</td>
</tr>
<tr>
<td>&quot;hessian&quot;</td>
<td>Hessian LLE</td>
</tr>
<tr>
<td>&quot;modified&quot;</td>
<td>Modified LLE</td>
</tr>
<tr>
<td>&quot;isomap&quot;</td>
<td>Isomap</td>
</tr>
<tr>
<td>&quot;mds&quot;</td>
<td>Multi-Dimensional Scaling</td>
</tr>
<tr>
<td>&quot;spectral&quot;</td>
<td>Spectral Embedding</td>
</tr>
<tr>
<td>&quot;tsne&quot;</td>
<td>t-SNE</td>
</tr>
</tbody>
</table>
Each of these algorithms embeds non-linear relationships in different ways, allowing for an exploration of various structures in the feature space. Note however, that each of these algorithms has different time, memory and complexity requirements; take special care when using large datasets!

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

**Parameters**

- `ax` [matplotlib Axes, default: None] The axes to plot the figure on. If None, the current axes will be used or generated if required.
- `manifold` [str or Transformer, default: “lle”] Specify the manifold algorithm to perform the embedding. Either one of the strings listed in the table above, or an actual scikit-learn transformer. The constructed manifold is accessible with the manifold property, so as to modify hyperparameters before fit.
- `n_neighbors` [int, default: 10] Many manifold algorithms are nearest neighbors based, for those that are, this parameter specifies the number of neighbors to use in the embedding. If the manifold algorithm doesn’t use nearest neighbors, then this parameter is ignored.
- `colors` [str or list of colors, default: None] Specify the colors used, though note that the specification depends very much on whether the target is continuous or discrete. If continuous, colors must be the name of a colormap. If discrete, then colors can be the name of a palette or a list of colors to use for each class in the target.
- `target` [str, default: “auto”] Specify the type of target as either “discrete” (classes) or “continuous” (real numbers, usually for regression). If “auto”, the Manifold will attempt to determine the type by counting the number of unique values.

If the target is discrete, points will be colored by the target class and a legend will be displayed. If continuous, points will be displayed with a colormap and a color bar will be displayed. In either case, if no target is specified, only a single color will be drawn.
- `alpha` [float, default: 0.7] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.
- `random_state` [int or RandomState, default: None] Fixes the random state for stochastic manifold algorithms.
- `kwargs` [dict] Keyword arguments passed to the base class and may influence the feature visualization properties.

**Notes**

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

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

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

See also:
The Scikit-Learn discussion on Manifold Learning.

Examples

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

Attributes

- **fit_time_[yellowbrick.utils.timer.Timer]** The amount of time in seconds it took to fit the Manifold.
- **classes_[np.ndarray, optional]** If discrete, the classes identified in the target y.
- **range_[tuple of floats, optional]** If continuous, the maximum and minimum values in the target y.

**ALGORITHMS** = {'hessian': LocallyLinearEmbedding(eigen_solver='auto', hessian_tol=0.0001, max_iter=100, method='hessian', ... n_components=2, n_iter=1000, n_iter_without_progress=300, perplexity=30.0, random_state=None, verbose=0)}

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

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

**Parameters**

- **X** [array-like of shape (n, 2)] The matrix produced by the transform() method.
- **y** [array-like of shape (n,), optional] The target, used to specify the colors of the points.

**Returns**

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

**finalize()**

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

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

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

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

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

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

**Parameters**

- **X** [array-like of shape (n, m)] A matrix or data frame with n instances and m features where m > 2.
- **y** [array-like of shape (n,), optional] A vector or series with target values for each instance in X. This vector is used to determine the color of the points in X.

**Returns**

- **self** [Manifold] Returns the visualizer object.
manifold
Property containing the manifold transformer constructed from the supplied hyperparameter. Use this property to modify the manifold before fit with manifold.set_params().

transform(X)
Returns the transformed data points from the manifold embedding.

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

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

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-evaluate 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:

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

# Specify the features of interest
features = [
    "temperature", "relative humidity", "light", "CO2", "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:

```python
import matplotlib.pyplot as plt
from sklearn.ensemble import GradientBoostingClassifier
from yellowbrick.features.importances 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:

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

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

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

```python
import matplotlib.pyplot as plt
from sklearn.linear_model import Lasso

import matplotlib.pyplot as plt
from sklearn.linear_model import Lasso
```

(continues on next page)
from yellowbrick.features.importances import FeatureImportances

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

# 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()

Note: The interpretation of the importance of coefficients 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 oppor-
tunity 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.

```python
class yellowbrick.features.importances.FeatureImportances(model,          
ax=None,  
labels=None,  
relative=True,  
absolute=False,  
xlabel=None,  
stack=False,  
**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.

Note: Some classification models such as LogisticRegression, return coef_ as a multidimensional array of shape (n_classes, n_features). In this case, the FeatureImportances visualizer computes the mean of the coefs_ by class for each feature.

Parameters

- **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.

stack  [bool, default: False] If true and the classifier returns multi-class feature importance, then a stacked bar plot is plotted; otherwise the mean of the feature importance across classes are plotted.

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

Examples

```python
>>> 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
- **classes_**  [np.array] The classees labeled. Is not None only for classifier.

**draw(**kwargs**)**

Draws the feature importances as a bar chart; called from fit.

**finalize(**kwargs**)**

Finalize the drawing setting labels and title.

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

Fits the estimator to discover the feature importances described by the data, then draws those importances as a bar plot.

Parameters

- **X**  [ndarray or DataFrame of shape n x m] A matrix of n instances with m features
- **y**  [ndarray or Series of length n] An array or series of target or class values
- **kwargs**  [dict] Keyword arguments passed to the fit method of the estimator.

**Returns**

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

Recursive Feature Elimination

Recursive feature elimination (RFE) is a feature selection method that fits a model and removes the weakest feature (or features) until the specified number of features is reached. Features are ranked by the model’s `coef_` or `feature_importances_` attributes, and by recursively eliminating a small number of features per loop, RFE attempts to eliminate dependencies and collinearity that may exist in the model.
RFE requires a specified number of features to keep, however it is often not known in advance how many features are valid. To find the optimal number of features cross-validation is used with RFE to score different feature subsets and select the best scoring collection of features. The RFECV visualizer plots the number of features in the model along with their cross-validated test score and variability and visualizes the selected number of features.

To show how this works in practice, we’ll start with a contrived example using a dataset that has only 3 informative features out of 25.

```python
from sklearn.svm import SVC
from sklearn.datasets import make_classification
from yellowbrick.features import RFECV

# Create a dataset with only 3 informative features
X, y = make_classification(
    n_samples=1000, n_features=25, n_informative=3, n_redundant=2,
    n_repeated=0, n_classes=8, n_clusters_per_class=1, random_state=0
)

# Create RFECV visualizer with linear SVM classifier
viz = RFECV(SVC(kernel='linear', C=1))
viz.fit(X, y)
viz.poof()

This figure shows an ideal RFECV curve, the curve jumps to an excellent accuracy when the three informative features are captured, then gradually decreases in accuracy as the non informative features are added into the model. The shaded area represents the variability of cross-validation, one standard deviation above and below the mean accuracy score drawn by the curve.
Exploring a real dataset, we can see the impact of RFECV on a credit default binary classifier.

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold

df = load_data('credit')
target = 'default'
features = [col for col in data.columns if col != target]

X = data[features]
y = data[target]

cv = StratifiedKFold(5)
oz = RFECV(RandomForestClassifier(), cv=cv, scoring='f1_weighted')

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

In this example we can see that 19 features were selected, though there doesn’t appear to be much improvement in the f1 score of the model after around 5 features. Selection of the features to eliminate plays a large role in determining the outcome of each recursion; modifying the step parameter to eliminate more than one feature at each step may help to eliminate the worst features early, strengthening the remaining features (and can also be used to speed up feature elimination for datasets with a large number of features).

See also:

This visualizer is is based off of the visualization in the scikit-learn documentation: recursive feature elimination with cross-validation. However, the Yellowbrick version does not use sklearn.feature_selection.RFECV
but instead wraps sklearn.feature_selection.RFE models. The fitted model can be accessed on the visualizer using the viz.rfe_estimator_ attribute, and in fact the visualizer acts as the fitted model when using predict() or score().

**API Reference**

Visualize the number of features selected using recursive feature elimination

```python
class yellowbrick.features.rfecv.RFECV(model, ax=None, step=1, groups=None, cv=None, scoring=None, **kwargs)
```

Bases: yellowbrick.base.ModelVisualizer

Recursive Feature Elimination, Cross-Validated (RFECV) feature selection.

Selects the best subset of features for the supplied estimator by removing 0 to N features (where N is the number of features) using recursive feature elimination, then selecting the best subset based on the cross-validation score of the model. Recursive feature elimination eliminates n features from a model by fitting the model multiple times and at each step, removing the weakest features, determined by either the coef_ or feature_importances_ attribute of the fitted model.

The visualization plots the score relative to each subset and shows trends in feature elimination. If the feature elimination CV score is flat, then potentially there are not enough features in the model. An ideal curve is when the score jumps from low to high as the number of features removed increases, then slowly decreases again from the optimal number of features.

**Parameters**

- **model** [a scikit-learn estimator] An object that implements fit and provides information about the relative importance of features with either a coef_ or feature_importances_ attribute. Note that the object is cloned for each validation.
- **ax** [matplotlib.Axes object, optional] The axes object to plot the figure on.
- **step** [int or float, optional (default=1)] If greater than or equal to 1, then step corresponds to the (integer) number of features to remove at each iteration. If within (0.0, 1.0), then step corresponds to the percentage (rounded down) of features to remove at each iteration.
- **groups** [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.
- **cv** [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:
  - None, to use the default 3-fold cross-validation,
  - integer, to specify the number of folds.
  - An object to be used as a cross-validation generator.
  - An iterable yielding train/test splits.

see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

- **scoring** [string, callable or None, optional, default: None] A string or scorer callable object / function with signature scorer(estimator, X, y). See scikit-learn model evaluation documentation for names of possible metrics.
- **kwargs** [dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

4.3. Visualizers and API 69
Notes

This model wraps `sklearn.feature_selection.RFE` and not `sklearn.feature_selection.RFECV` because access to the internals of the CV and RFE estimators is required for the visualization. The visualizer does take similar arguments, however it does not expose the same internal attributes.

Additionally, the RFE model can be accessed via the `rfe_estimator_` attribute. Once fitted, the visualizer acts as a wrapper for this estimator and not for the original model passed to the model. This way the visualizer model can be used to make predictions.

Caution: This visualizer requires a model that has either a `coef_` or `feature_importances_` attribute when fitted.

Attributes

- `n_features_` [int] The number of features in the selected subset
- `support_` [array of shape [n_features]] A mask of the selected features
- `ranking_` [array of shape [n_features]] The feature ranking, such that `ranking_[i]` corresponds to the ranked position of feature i. Selected features are assigned rank 1.
- `cv_scores_` [array of shape [n_subsets_of_features, n_splits]] The cross-validation scores for each subset of features and splits in the cross-validation strategy.
- `rfe_estimator_` [sklearn.feature_selection.RFE] A fitted RFE estimator wrapping the original estimator. All estimator functions such as `predict()` and `score()` are passed through to this estimator (it rewraps the original model).

`draw(**kwargs)`
Renders the rfecv curve.

`finalize(**kwargs)`
Add the title, legend, and other visual final touches to the plot.

`fit(X, y=None)`
Fits the RFECV with the wrapped model to the specified data and draws the rfecv curve with the optimal number of features found.

Parameters

- `X` [array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.
- `y` [array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression.

Returns

- `self` [instance] Returns the instance of the RFECV visualizer.

Direct Data Visualization

Sometimes for feature analysis you simply need a scatter plot to determine the distribution of data. Machine learning operates on high dimensional data, so the number of dimensions has to be filtered. As a result these visualizations are typically used as the base for larger visualizers; however you can also use them to quickly plot data during ML analysis.
Joint Plot Visualization

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

```python
# 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]

from yellowbrick.features import JointPlotVisualizer

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.

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

visualizer.fit(X, y)
visualizer.poof()
```
JointPlotVisualizer allows for a simultaneous visualization of the relationship between two variables and the distri-
bution 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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>transparency</td>
</tr>
<tr>
<td>face-color</td>
<td>background color of the joint axis</td>
</tr>
<tr>
<td>aspect</td>
<td>aspect ratio</td>
</tr>
<tr>
<td>fit</td>
<td>used if scatter is selected for joint_plot to draw a best fit line - values can be True or False. Uses Yellowbrick.bestfit</td>
</tr>
<tr>
<td>estimator</td>
<td>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.</td>
</tr>
<tr>
<td>x_bins</td>
<td>used if hex is selected to set the number of bins for the x value</td>
</tr>
<tr>
<td>y_bins</td>
<td>used if hex is selected to set the number of bins for the y value</td>
</tr>
<tr>
<td>cmap</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>transparency</td>
</tr>
<tr>
<td>facecolor_x</td>
<td>background color of the x axis</td>
</tr>
<tr>
<td>facecolor_y</td>
<td>background color of the y axis</td>
</tr>
<tr>
<td>bins</td>
<td>used to set up the number of bins for the hist plot</td>
</tr>
<tr>
<td>histcolor_x</td>
<td>used to set the color for the histogram on the x axis</td>
</tr>
<tr>
<td>histcolor_y</td>
<td>used to set the color for the histogram on the y axis</td>
</tr>
</tbody>
</table>

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

data: 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

kwarg: [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

```python
>>> 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

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.

### 4.3.4 Target Visualizers

Target visualizers specialize in visually describing the dependent variable for supervised modeling, often referred to as $y$ or the target.

The following visualizations are currently implemented:

- **Balanced Binning Reference**: Generate histogram with vertical lines showing the recommended value point to bin data into evenly distributed bins.
- **Class Balance**: Visual inspection of the target to show the support of each class to the final estimator.
- **Feature Correlation**: Plot correlation between features and dependent variables.

```python
# Target Visualizers Imports
from yellowbrick.target import BalancedBinningReference
from yellowbrick.target import ClassBalance
from yellowbrick.target import FeatureCorrelation
```
Balanced Binning Reference

Frequently, machine learning problems in the real world suffer from the curse of dimensionality; you have fewer
training instances than you’d like and the predictive signal is distributed (often unpredictably!) across many different
features.

Sometimes when the your target variable is continuously-valued, there simply aren’t enough instances to predict
these values to the precision of regression. In this case, we can sometimes transform the regression problem into a
classification problem by binning the continuous values into makeshift classes.

To help the user select the optimal number of bins, the BalancedBinningReference visualizer takes the target
variable \( y \) as input and generates a histogram with vertical lines indicating the recommended value points to ensure
that the data is evenly distributed into each bin.

```python
from yellowbrick.target import BalancedBinningReference

data = load_data("concrete")

y = data["strength"]

visualizer = BalancedBinningReference()

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

See also:

To learn more, please read Rebecca Bilbro’s article “Creating Categorical Variables from Continuous Data.”
API Reference

Implements histogram with vertical lines to help with balanced binning.

class yellowbrick.target.binning.BalancedBinningReference(ax=None, target=None, bins=4, **kwargs)

Bases: yellowbrick.target.base.TargetVisualizer

BalancedBinningReference generates a histogram with vertical lines showing the recommended value point to bin your data so they can be evenly distributed in each bin.

Parameters

- ax [matplotlib Axes, default: None] This is inherited from FeatureVisualizer and is defined within BalancedBinningReference.
- target [string, default: “Frequency”] The name of the y variable
- bins [number of bins to generate the histogram, default: 4]
- kwargs [dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

Notes

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

Examples

```python
>>> visualizer = BalancedBinningReference()
>>> visualizer.fit(y)
>>> visualizer.poof()
```

Attributes

- bin_edges [binning reference values]

`draw`(y, **kwargs)

Draws a histogram with the reference value for binning as vertical lines.

Parameters

- y [an array of one dimension or a pandas Series]

finalize(**kwargs)

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

Parameters

- kwargs: generic keyword arguments.

fit(y, **kwargs)

Sets up y for the histogram and checks to ensure that y is of the correct data type. Fit calls draw.

Parameters

- y [an array of one dimension or a pandas Series]
- kwargs [dict] keyword arguments passed to scikit-learn API.
**poof(** \*kwargs\*)

Creates the labels for the feature and target variables.

### Class Balance

One of the biggest challenges for classification models is an imbalance of classes in the training data. Severe class imbalances may be masked by relatively good F1 and accuracy scores – the classifier is simply guessing the majority class and not making any evaluation on the underrepresented class.

There are several techniques for dealing with class imbalance such as stratified sampling, down sampling the majority class, weighting, etc. But before these actions can be taken, it is important to understand what the class balance is in the training data. The `ClassBalance` visualizer supports this by creating a bar chart of the `support` for each class, that is the frequency of the classes’ representation in the dataset.

```python
from yellowbrick.datasets import load_game
from yellowbrick.target import ClassBalance

# Load the classification data set
data = load_game()

# Specify the target
y = data["outcome"]

visualizer = ClassBalance(labels=["draw", "loss", "win"],)
visualizer.fit(y)
visualizer.poof()
```

![Class Balance for 67,557 Instances](image)

**Class Balance for 67,557 Instances**

- **draw**: 0
- **loss**: 10,000
- **win**: 40,000

---

**Chapter 4. Table of Contents**
The resulting figure allows us to diagnose the severity of the balance issue. In this figure we can see that the "win" class dominates the other two classes. One potential solution might be to create a binary classifier: "win" vs "not win" and combining the "loss" and "draw" classes into one class.

**Warning:** The ClassBalance visualizer interface has changed in version 0.9, a classification model is no longer required to instantiate the visualizer, it can operate on data only. Additionally the signature of the fit method has changed from \texttt{fit(X, y=None)} to \texttt{fit(y\_train, y\_test=None)}, passing in \texttt{X} is no longer required.

If a class imbalance must be maintained during evaluation (e.g. the event being classified is actually as rare as the frequency implies) then \textit{stratified sampling} should be used to create train and test splits. This ensures that the test data has roughly the same proportion of classes as the training data. While scikit-learn does this by default in \texttt{train\_test\_split} and other cv methods, it can be useful to compare the support of each class in both splits.

The ClassBalance visualizer has a “compare” mode, where the train and test data can be passed to \texttt{fit()}, creating a side-by-side bar chart instead of a single bar chart as follows:

```python
from sklearn.model_selection import train_test_split
from yellowbrick.model_selection import ClassBalance

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

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

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

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

# Instantiate the classification model and visualizer
visualizer = ClassBalance(labels=classes)

visualizer.fit(y_train, y_test)
return visualizer.poof()
```
This visualization allows us to do a quick check to ensure that the proportion of each class is roughly similar in both splits. This visualization should be a first stop particularly when evaluation metrics are highly variable across different splits.

**API Reference**

Class balance visualizer for showing per-class support.

```python
class yellowbrick.target.class_balance.ClassBalance(ax=None, labels=None, **kwargs)
```

One of the biggest challenges for classification models is an imbalance of classes in the training data. The ClassBalance visualizer shows the relationship of the support for each class in both the training and test data by displaying how frequently each class occurs as a bar graph.

The ClassBalance visualizer can be displayed in two modes:

1. Balance mode: show the frequency of each class in the dataset.
2. Compare mode: show the relationship of support in train and test data.

These modes are determined by what is passed to the `fit()` method.

**Parameters**

- `ax` [matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).
labels: list, optional A list of class names for the x-axis if the target is already encoded. Ensure that the labels are ordered lexicographically with respect to the values in the target. A common use case is to pass LabelEncoder.classes_ as this parameter. If not specified, the labels in the data will be used.

kwags: dict, optional Keyword arguments passed to the super class. Here, used to colorize the bars in the histogram.

Attributes

classes_ [array-like] The actual unique classes discovered in the target.
support_ [array of shape (n_classes,) or (2, n_classes)] A table representing the support of each class in the target. It is a vector when in balance mode, or a table with two rows in compare mode.

draw() Renders the class balance chart on the specified axes from support.

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

Parameters

kwags: generic keyword arguments.

fit(y_train, y_test=None) Fit the visualizer to the the target variables, which must be 1D vectors containing discrete (classification) data. Fit has two modes:

1. Balance mode: if only y_train is specified
2. Compare mode: if both train and test are specified

In balance mode, the bar chart is displayed with each class as its own color. In compare mode, a side-by-side bar chart is displayed colored by train or test respectively.

Parameters

y_train [array-like] Array or list of shape (n,) that contains discrete data.
y_test [array-like, optional] Array or list of shape (m,) that contains discrete data. If specified, the bar chart will be drawn in compare mode.

Feature Correlation

This visualizer calculates Pearson correlation coefficients and mutual information between features and the dependent variable. This visualization can be used in feature selection to identify features with high correlation or large mutual information with the dependent variable.

Pearson Correlation

The default calculation is Pearson correlation, which is perform with scipy.stats.pearsonr.

```python
from sklearn import datasets
from yellowbrick.target import FeatureCorrelation

# Load the regression data set
data = datasets.load_diabetes()
X, y = data['data'], data['target']
```
Mutual information between features and the dependent variable is calculated with sklearn.feature_selection.mutual_info_classif when method='mutual_info-classification' and mutual_info_regression when method='mutual_info-regression'. It is very important to specify discrete features when calculating mutual information because the calculation for continuous and discrete variables are different. See scikit-learn documentation for more details.
visualizer = FeatureCorrelation(method='mutual_info-regression',
    labels=feature_names)
visualizer.fit(X, y, discrete_features=discrete_features, random_state=0)
visualizer.poof()

Mutual Information - Classification

By fitting with a pandas DataFrame, the feature labels are automatically obtained from the column names. This visualizer also allows sorting of the bar plot according to the calculated mutual information (or Pearson correlation coefficients) and selecting features to plot by specifying the names of the features or the feature index.

```python
from sklearn import datasets
from yellowbrick.target import FeatureCorrelation

# Load the regression data set
data = datasets.load_diabetes()
X, y = data['data'], data['target']
feature_names = np.array(data['feature_names'])
X_pd = pd.DataFrame(X, columns=feature_names)

feature_to_plot = ['alcohol', 'ash', 'hue', 'proline', 'total_phenols']

visualizer = FeatureCorrelation(method='mutual_info-classification',
    feature_names=feature_to_plot, sort=True)
```

4.3. Visualizers and API

(continues on next page)
visualizer.fit(X_pd, y, random_state=0)
visualizer.poof()

API Reference

Feature Correlation to Dependent Variable Visualizer.

```
class yellowbrick.target.feature_correlation.FeatureCorrelation(ax=None, method='pearson', labels=None, sort=False, feature_index=None, feature_names=None, **kwargs)
```

Bases: yellowbrick.target.base.TargetVisualizer

Displays the correlation between features and dependent variables.

This visualizer can be used side-by-side with yellowbrick.features.JointPlotVisualizer that plots a feature against the target and shows the distribution of each via a histogram on each axis.

**Parameters**

- **ax** [matplotlib Axes, default: None] The axis to plot the figure on. If None is passed in the current axes will be used (or generated if required).
method [str, default: ‘pearson’] The method to calculate correlation between features and target. Options include:
- ‘pearson’, which uses \texttt{scipy.stats.pearsonr}
- ‘mutual_info-regression’, which uses \texttt{mutual_info-regression} from \texttt{sklearn.feature_selection}
- ‘mutual_info-classification’, which uses \texttt{mutual_info_classif} from \texttt{sklearn.feature_selection}

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

sort [boolean, default: False] If false, the features are not sorted in the plot; otherwise features are sorted in ascending order of correlation.

feature_index [list] A list of feature index to include in the plot.

feature_names [list of feature names] A list of feature names to include in the plot. Must have labels or the fitted data is a DataFrame with column names. If feature_index is provided, feature_names will be ignored.

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

Examples

```python
>>> viz = FeatureCorrelation()
>>> viz.fit(X, y)
>>> viz.poof()
```

Attributes

- \texttt{features_} [np.array] The feature labels
- \texttt{scores_} [np.array] Correlation between features and dependent variable.

draw()
Draws the feature correlation to dependent variable, called from fit.

finalize()
Finalize the drawing setting labels and title.

fit (X, y, **kwarg)
Fits the estimator to calculate feature correlation to dependent variable.

Parameters

- \texttt{X} [ndarray or DataFrame of shape n x m] A matrix of n instances with m features
- \texttt{y} [ndarray or Series of length n] An array or series of target or class values
- \texttt{kwarg} [dict] Keyword arguments passed to the fit method of the estimator.

Returns

- \texttt{self} [visualizer] The fit method must always return self to support pipelines.
### 4.3.5 Regression Visualizers

Regression models attempt to predict a target in a continuous space. Regressor score visualizers display the instances in model space to better understand how the model is making predictions. We currently have implemented three regressor evaluations:

- **Residuals Plot**: plot the difference between the expected and actual values
- **Prediction Error Plot**: plot the expected vs. actual values in model space
- **Alpha Selection**: visual tuning of regularization hyperparameters

Estimator score visualizers *wrap* Scikit-Learn estimators and expose the Estimator API such that they have `fit()`, `predict()`, and `score()` methods that call the appropriate estimator methods under the hood. Score visualizers can wrap an estimator and be passed in as the final step in a `Pipeline` or `VisualPipeline`.

```python
# Regression Evaluation Imports
from sklearn.linear_model import Ridge, Lasso
from sklearn.model_selection import train_test_split
from yellowbrick.regressor import PredictionError, ResidualsPlot
from yellowbrick.regressor.alphas import AlphaSelection
```

**Residuals Plot**

Residuals, in the context of regression models, are the difference between the observed value of the target variable (\(y\)) and the predicted value (\(\hat{y}\)), e.g. the error of the prediction. The residuals plot shows the difference between residuals on the vertical axis and the dependent variable on the horizontal axis, allowing you to detect regions within the target that may be susceptible to more or less error.

```python
from sklearn.model_selection import train_test_split

# Load the data
df = load_data('concrete')

# Identify the feature and target columns
feature_names = ['cement', 'slag', 'ash', 'water', 'splast', 'coarse', 'fine', 'age']
target_name = 'strength'

# Separate the instance data from the target data
X = df[feature_names]
y = df[target_name]

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

from sklearn.linear_model import Ridge
from yellowbrick.regressor import ResidualsPlot

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

visualizer.fit(X_train, y_train)  # Fit the training data to the model
(continues on next page)
A common use of the residuals plot is to analyze the variance of the error of the regressor. If the points are randomly dispersed around the horizontal axis, a linear regression model is usually appropriate for the data; otherwise, a non-linear model is more appropriate. In the case above, we see a fairly random, uniform distribution of the residuals against the target in two dimensions. This seems to indicate that our linear model is performing well. We can also see from the histogram that our error is normally distributed around zero, which also generally indicates a well fitted model.

Note that if the histogram is not desired, it can be turned off with the `hist=False` flag:

```python
visualizer = ResidualsPlot(ridge, hist=False)
visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.poof()
```
Warning: The histogram on the residuals plot requires matplotlib 2.0.2 or greater. If you are using an earlier version of matplotlib, simply set the `hist=False` flag so that the histogram is not drawn.

API Reference

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

```python
class yellowbrick.regressor.residuals.ResidualsPlot(model, ax=None, hist=True, train_color='b', test_color='g', line_color='#111111', alpha=0.75, **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 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).
yellowbrick Documentation, Release 0.9.1

**hist** ([True, False, None, ‘density’, ‘frequency’], default: True] Draw a histogram showing the distribution of the residuals on the right side of the figure. Requires Matplotlib >= 2.0.2. If set to ‘density’, the probability density function will be plotted. If set to True or ‘frequency’ then the frequency will be plotted.

**train_color** [color, default: ‘b’] Residuals for training data are plotted with this color but also given an opacity of 0.5 to ensure that the test data residuals are more visible. Can be any matplotlib color.

**test_color** [color, default: ‘g’] Residuals for test data are plotted with this color. In order to create generalizable models, reserved test data residuals are of the most analytical interest, so these points are highlighted by having full opacity. Can be any matplotlib color.

**line_color** [color, default: dark grey] Defines the color of the zero error line, can be any matplotlib color.

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

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

**Notes**

ResidualsPlot is a ScoreVisualizer, meaning that it wraps a model and its primary entry point is the `score()` method.

The residuals histogram feature requires matplotlib 2.0.2 or greater.

**Examples**

```python
>>> 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**)

Draw the residuals against the predicted value for the specified split. It is best to draw the training split first, then the test split so that the test split (usually smaller) is above the training split; particularly if the histogram is turned on.

**Parameters**

- **y_pred** [ndarray or Series of length n] An array or series of predicted target values
- **residuals** [ndarray or Series of length n] An array or series of the difference between the predicted and the target values
- **train** [boolean, default: False] If False, `draw` assumes that the residual points being plotted are from the test data; if True, `draw` assumes the residuals are the train data.

**Returns**

- **ax** [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, **kwargs)

Parameters
  X [ndarray or DataFrame of shape n x m] A matrix of n instances with m features
  y [ndarray or Series of length n] An array or series of target values
  kwargs: keyword arguments passed to Scikit-Learn API.

Returns
  self [visualizer instance]

hax
Returns the histogram axes, creating it only on demand.

score(X, y=None, train=False, **kwargs)
Generates predicted target values using the Scikit-Learn estimator.

Parameters
  X [array-like] X (also X_test) are the dependent variables of test set to predict
  y [array-like] y (also y_test) is the independent actual variables to score against
  train [boolean] If False, score assumes that the residual points being plotted are from the test data; if True, score assumes the residuals are the train data.

Returns
  score [float] The score of the underlying estimator, usually the R-squared score for regression estimators.

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.

```python
from sklearn.model_selection import train_test_split

# Load the regression data set
data = load_data('concrete')

# Specify the features of interest and the target
features = ['cement', 'slag', 'ash', 'water', 'sp Laurel', 'coarse', 'fine', 'age']
target = 'strength'

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

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

```python
from sklearn.linear_model import Lasso

from yellowbrick.regressor import PredictionError

# 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.

```python
class yellowbrick.regressor.residuals.PredictionError(model, ax=None, shared_limits=True, best-fit=True, identity=True, alpha=0.75, **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 detect noise or heteroscedasticity along a range of the target domain.

Parameters

- **model** [a Scikit-Learn regressor] Should be an instance of a regressor, otherwise 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.
- **bestfit** [bool, default: True] Draw a linear best fit line to estimate the correlation between the predicted and measured value of the target variable. The color of the bestfit line is determined by the line_color argument.
- **identity** [bool, default: True] Draw the 45 degree identity line, y=x in order to better show the relationship or pattern of the residuals. E.g. to estimate if the model is over- or under-estimating the given values. The color of the identity line is a muted version of the line_color argument.
- **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.
- **alpha** [float, default: 0.75] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.
- **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

```python
>>> 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
**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.

```python
# Load the regression data set
df = load_data('concrete')

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

# Extract the instances and target
X = df[features]
y = df[target]

import numpy as np
from sklearn.linear_model import LassoCV
from yellowbrick.regressor import AlphaSelection

# Create a list of alphas to cross-validate against
alphas = np.logspace(-10, 1, 400)

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

(continues on next page)
visualizer.fit(X, y)
g = visualizer.poof()

API Reference

Implements alpha selection visualizers for regularization

class yellowbrick.regressor.alphas.AlphaSelection(model, ax=None, **kwargs)
    Bases: yellowbrick.regressor.base.ReggressionScoreVisualizer

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 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 hooks into the Scikit-Learn API during `fit()`. In order to pass a fitted model to the Visualizer, call the `draw()` method directly after instantiating the visualizer with the fitted model.

Note, each “RegressorCV” module has many different methods for storing alphas and error. This visualizer attempts to get them all and is known to work for RidgeCV, LassoCV, LassoLarsCV, and ElasticNetCV. If your favorite regularization method doesn’t work, please submit a bug report.

For RidgeCV, make sure `store_cv_values=True`.

**Examples**

```python
>>> 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.

score(X, y, **kwargs)  
Simply returns the score of the underlying CV model

**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**

```python
>>> 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.

**score** *(X, y, **kwargs)**

Simply returns the score of the underlying CV model

### 4.3.6 Classification Visualizers

Classification models attempt to predict a target in a discrete space, that is assign an instance of dependent variables one or more categories. Classification score visualizers display the differences between classes as well as a number of classifier-specific visual evaluations. We currently have implemented the following classifier evaluations:

- **Classification Report**: A visual classification report that displays precision, recall, and F1 per-class as a heatmap.
- **Confusion Matrix**: A heatmap view of the confusion matrix of pairs of classes in multi-class classification.
- **ROCAUC**: Graphs the receiver operating characteristics and area under the curve.
- **Precision-Recall Curves**: Plots the precision and recall for different probability thresholds.
- **Class Balance**: Visual inspection of the target to show the support of each class to the final estimator.
- **Class Prediction Error**: An alternative to the confusion matrix that shows both support and the difference between actual and predicted classes.
- **Discrimination Threshold**: Shows precision, recall, f1, and queue rate over all thresholds for binary classifiers that use a discrimination probability or score.

Estimator score visualizers wrap scikit-learn estimators and expose the Estimator API such that they have `fit()`, `predict()`, and `score()` methods that call the appropriate estimator methods under the hood. Score visualizers can wrap an estimator and be passed in as the final step in a `Pipeline` or `VisualPipeline`.

```python
# Classifier Evaluation Imports
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from yellowbrick.target import ClassBalance
from yellowbrick.classifier import ROCAUC
from yellowbrick.classifier import PrecisionRecallCurve
from yellowbrick.classifier import ClassificationReport
from yellowbrick.classifier import ClassPredictionError
from yellowbrick.classifier import DiscriminationThreshold
```

### Classification Report

The classification report visualizer displays the precision, recall, F1, and support scores for the model. In order to support easier interpretation and problem detection, the report integrates numerical scores with a color-coded heatmap. All heatmaps are in the range *(0.0, 1.0)* to facilitate easy comparison of classification models across different classification reports.
```python
from sklearn.model_selection import train_test_split

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 instances and target
X = data[features]
y = data.occupancy

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

from sklearn.naive_bayes import GaussianNB
from yellowbrick.classifier import ClassificationReport

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

visualizer.fit(X_train, y_train)  # Fit the visualizer and the model
visualizer.score(X_test, y_test)  # Evaluate the model on the test data

visualizer.poof()  # Draw/show/poof the data
```
The classification report shows a representation of the main classification metrics on a per-class basis. This gives a deeper intuition of the classifier behavior over global accuracy which can mask functional weaknesses in one class of a multiclass problem. Visual classification reports are used to compare classification models to select models that are “redder”, e.g. have stronger classification metrics or that are more balanced.

The metrics are defined in terms of true and false positives, and true and false negatives. Positive and negative in this case are generic names for the classes of a binary classification problem. In the example above, we would consider true and false occupied and true and false unoccupied. Therefore a true positive is when the actual class is positive as is the estimated class. A false positive is when the actual class is negative but the estimated class is positive. Using this terminology the metrics are defined as follows:

**precision** Precision is the ability of a classifier not to label an instance positive that is actually negative. For each class it is defined as as the ratio of true positives to the sum of true and false positives. Said another way, “for all instances classified positive, what percent was correct?”

**recall** Recall is the ability of a classifier to find all positive instances. For each class it is defined as the ratio of true positives to the sum of true positives and false negatives. Said another way, “for all instances that were actually positive, what percent was classified correctly?”

**f1 score** The F₁ score is a weighted harmonic mean of precision and recall such that the best score is 1.0 and the worst is 0.0. Generally speaking, F₁ scores are lower than accuracy measures as they embed precision and recall into their computation. As a rule of thumb, the weighted average of F₁ should be used to compare classifier models, not global accuracy.

**support** Support is the number of actual occurrences of the class in the specified dataset. Imbalanced support in the training data may indicate structural weaknesses in the reported scores of the classifier and could indicate the need for stratified sampling or rebalancing. Support doesn’t change between models but instead diagnoses the evaluation process.
Visual classification report for classifier scoring.

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

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

Parameters

- **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.
- **cmap** [string, default: 'YlOrRd'] Specify a colormap to define the heatmap of the predicted class against the actual class in the classification report.
- **support**: {True, False, None, 'percent', 'count'}, default: None Specify if support will be displayed. It can be further defined by whether support should be reported as a raw count or percentage.
- **kwargs** [keyword arguments passed to the super class.]

Examples

```python
>>> 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()
```

Attributes

- **score_** [float] Global accuracy score
- **scores_** [dict of dicts] Outer dictionary composed of precision, recall, f1, and support scores with inner dictionaries specifying the values for each class listed.

```python
draw()
draw()
```

Renders the classification report across each axis.

```python
finalize(**kwargs)
```

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

Parameters
kwags: 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

**Returns**

- **score_** [float] Global accuracy score

**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.

```python
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
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()
```
Plotting with Class Names

Class names can be added to a ConfusionMatrix plot using the label_encoder argument. The label_encoder can be a sklearn.preprocessing.LabelEncoder (or anything with an inverse_transform method that performs the mapping), or a dict with the encoding-to-string mapping as in the example below:

```python
iris = load_iris()
X = iris.data
y = iris.target
classes = iris.target_names

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

model = LogisticRegression()

iris_cm = ConfusionMatrix(
    model, classes=classes,
    label_encoder={0: 'setosa', 1: 'versicolor', 2: 'virginica'}
)

iris_cm.fit(X_train, y_train)
iris_cm.score(X_test, y_test)
iris_cm.poof()
```
Visual confusion matrix for classifier scoring.

```python
class yellowbrick.classifier.confusion_matrix.ConfusionMatrix(model, ax=None, classes=None, sample_weight=None, percent=False, label_encoder=None, cmap='YlOrRd', fontsize=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** [estimator] Must be a classifier, otherwise raises `YellowbrickTypeError`
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).

sample_weight: array-like of shape = [n_samples], optional Passed to confusion_matrix to weight the samples.

percent: bool, default: False Determines whether or not the confusion_matrix is displayed as counts or as a percent of true predictions. Note, if specifying a subset of classes, percent should be set to False or inaccurate figures will be displayed.

classes [list, 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, classes 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.

cmap [string, default: 'YlOrRd'] Specify a colormap to define the heatmap of the predicted class against the actual class in the confusion matrix.

fontsize [int, default: None] Specify the fontsize of the text in the grid and labels to make the matrix a bit easier to read. Uses rcParams font size by default.

Examples

```python
>>> 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()
```

Attributes

score_ [float] Global accuracy score

confusion_matrix_ [array, shape = [n_classes, n_classes]] The numeric scores of the confusion matrix

class_counts_ [array, shape = [n_classes,]] The total number of each class supporting the confusion matrix

draw()
Renders the classification report; must be called after score.

finalize(**kwargs)
Finalize executes any subclass-specific axes finalization steps.

Parameters

kwargs: dict generic keyword arguments.
Notes

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

\texttt{score}(X, y)

Draws a confusion matrix based on the test data supplied by comparing predictions on instances X with the true values specified by the target vector y.

**Parameters**

\begin{itemize}
  \item \texttt{X} [ndarray or DataFrame of shape n x m] A matrix of n instances with m features
  \item \texttt{y} [ndarray or Series of length n] An array or series of target or class values
\end{itemize}

**Returns**

\texttt{score} [float] Global accuracy score

\textbf{ROCAUC}

A \texttt{ROCAUC} (Receiver Operating Characteristic/Area Under the Curve) plot allows the user to visualize the tradeoff between the classifier’s sensitivity and specificity.

The Receiver Operating Characteristic (ROC) is a measure of a classifier’s predictive quality that compares and visualizes the tradeoff between the model’s sensitivity and specificity. When plotted, a ROC curve displays the true positive rate on the Y axis and the false positive rate on the X axis on both a global average and per-class basis. The ideal point is therefore the top-left corner of the plot: false positives are zero and true positives are one.

This leads to another metric, area under the curve (AUC), which is a computation of the relationship between false positives and true positives. The higher the AUC, the better the model generally is. However, it is also important to inspect the “steepness” of the curve, as this describes the maximization of the true positive rate while minimizing the false positive rate.

```python
from sklearn.model_selection import train_test_split

# 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", "CO2", "humidity"]
classes = ["unoccupied", "occupied"]

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

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

from yellowbrick.classifier import ROCAUC
from sklearn.linear_model import LogisticRegression

# Instantiate the visualizer with the classification model
visualizer = ROCAUC(LogisticRegression(), classes=classes)

visualizer.fit(X_train, y_train) # Fit the training data to the visualizer
```

(continues on next page)
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
g = visualizer.poof()  # Draw/show/poof the data

Warning: Binary classification using a Scikit-learn-style estimator with only a decision_function, triggers an IndexError because the predictions will be a 1D array, meaning there is only sufficient information to plot a single curve. More on this bug can be found in this notebook. The bug was addressed in a July 2018 PR and will be fixed in v0.9, where the solution will be to set the micro, macro, and per-class parameters of ROCAUC to False.

Multi-class ROCAUC Curves

Yellowbrick’s ROCAUC Visualizer does allow for plotting multiclass classification curves. ROC curves are typically used in binary classification, and in fact the Scikit-Learn roc_curve metric is only able to perform metrics for binary classifiers. Yellowbrick addresses this by binarizing the output (per-class) or to use one-vs-rest (micro score) or one-vs-all (macro score) strategies of classification.

# Load multi-class classification dataset
game = load_game()

classes = ['win', 'loss', 'draw']

# Encode the non-numeric columns
game.replace({'loss':-1, 'draw':0, 'win':1, 'x':2, 'o':3, 'b':4}, inplace=True)
# Extract the instances and target
X = game.iloc[:, game.columns != 'outcome']
y = game['outcome']

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

```python
from sklearn.linear_model import RidgeClassifier
visualizer = ROCAUC(RidgeClassifier(), 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
```

By default with multi-class ROCAUC visualizations, a curve for each class is plotted, in addition to the micro- and macro-average curves for each class. This enables the user to inspect the tradeoff between sensitivity and specificity on a per-class basis. Note that for multi-class ROCAUC, at least one of the micro, macro, or per_class parameters must be set to True (by default, all are set to True).

### API Reference

**Implements visual ROC/AUC curves for classification evaluation.**

4.3. Visualizers and API 107
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

- **model** [estimator] Must be a classifier, otherwise raises YellowbrickTypeError
- **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).
- **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. Micro is not defined for binary classification problems with estimators with only a decision_function method.
- **macro** [bool, default = True] Plot the macro-averages ROC curve, which simply takes the average of curves across all classes. Macro is not defined for binary classification problems with estimators with only a decision_function method.
- **per_class** [bool, default = True] Plot the ROC curves for each individual class. This should be set to false if only the macro or micro average curves are required. Per-class classification is not defined for binary classification problems with estimators with only a decision_function method.
- **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.

See also:

Examples

```python
>>> from yellowbrick.classifier import ROCAUC
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.model_selection import train_test_split

>>> data = load_data("occupancy")
>>> features = ["temp", "relative humidity", "light", "CO2", "humidity"]
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)

>>> oz = ROCAUC(LogisticRegression())
>>> oz.fit(X_train, y_train)
>>> oz.score(X_test, y_test)
>>> oz.poof()
```

Attributes

- **score_** [float] Global accuracy score, unless micro or macro scores are requested

**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] Global accuracy unless micro or macro scores are requested.

**Precision-Recall Curves**

Precision-Recall curves are a metric used to evaluate a classifier’s quality, particularly when classes are very imbalanced. The precision-recall curve shows the tradeoff between precision, a measure of result relevancy, and recall, a measure of how many relevant results are returned. A large area under the curve represents both high recall and precision, the best case scenario for a classifier, showing a model that returns accurate results for the majority of classes it selects.

**Binary Classification**

```python
from sklearn.linear_model import RidgeClassifier
from sklearn.model_selection import train_test_split as tts
from yellowbrick.classifier import PrecisionRecallCurve
```
# Load the dataset and split into train/test splits

data = load_spam()
X = data[[col for col in data.columns if col != "is_spam"]]
y = data["is_spam"]

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

# Create the visualizer, fit, score, and poof it

ıziz = PrecisionRecallCurve(RidgeClassifier())

viz.fit(X_train, y_train)

viz.score(X_test, y_test)

viz.poof()

The base case for precision-recall curves is the binary classification case, and this case is also the most visually interpretable. In the figure above we can see the precision plotted on the y-axis against the recall on the x-axis. The larger the filled in area, the stronger the classifier is. The red line annotates the average precision, a summary of the entire plot computed as the weighted average of precision achieved at each threshold such that the weight is the difference in recall from the previous threshold.

**Multi-Label Classification**

To support multi-label classification, the estimator is wrapped in a `OneVsRestClassifier` to produce binary comparisons for each class (e.g. the positive case is the class and the negative case is any other class). The Precision-Recall curve is then computed as the micro-average of the precision and recall for all classes:
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder

# Load dataset and encode categorical variables
data = load_game()
data.replace({'x':0, 'o':1, 'b':2}, inplace=True)

# Create train/test splits
X = data.iloc[:, data.columns != 'outcome']
y = LabelEncoder().fit_transform(data['outcome'])

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

# Create the visualizer, fit, score, and poof it
viz = PrecisionRecallCurve(RandomForestClassifier())
viz.fit(X_train, y_train)
viz.score(X_test, y_test)
viz.poof()

A more complex Precision-Recall curve can be computed, however, displaying the each curve individually, along with F1-score ISO curves (e.g. that show the relationship between precision and recall for various F1 scores).

from sklearn.naive_bayes import MultinomialNB

oz = PrecisionRecallCurve(
    MultinomialNB(), per_class=True, iso_f1_curves=True,
    fill_area=False, micro=False
)
viz.fit(X_train, y_train)
viz.score(X_test, y_test)
viz.poof()

**Precision-Recall Curve for MultinomialNB**

See also:
Scikit-Learn: Model Selection with Precision Recall Curves

**API Reference**

Implements Precision-Recall curves for classification models.

```python
class yellowbrick.classifier.prcurve.PrecisionRecallCurve(model, ax=None, classes=None, fill_area=True, ap_score=True, micro=True, iso_fl_curves=False, per_class=False, fill_opacity=0.2, line_opacity=0.8, **kwargs)
```

**Bases:** yellowbrick.classifier.base.ClassificationScoreVisualizer

Precision-Recall curves are a metric used to evaluate a classifier’s quality, particularly when classes are very imbalanced. The precision-recall curve shows the tradeoff between precision, a measure of result relevancy, and recall, a measure of how many relevant results are returned. A large area under the curve represents both high
recall and precision, the best case scenario for a classifier, showing a model that returns accurate results for the majority of classes it selects.

**Parameters**

- **model** [the Scikit-Learn estimator] A classification model to score the precision-recall curve on.
- **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).
- **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.
- **fill_area** [bool, default=True] Fill the area under the curve (or curves) with the curve color.
- **ap_score** [bool, default=True] Annotate the graph with the average precision score, a summary of the plot that is computed as the weighted mean of precisions at each threshold, with the increase in recall from the previous threshold used as the weight.
- **micro** [bool, default=True] If multi-class classification, draw the precision-recall curve for the micro-average of all classes. In the multi-class case, either micro or per-class must be set to True. Ignored in the binary case.
- **iso_f1_curves** [bool, default=False] Draw ISO F1-Curves on the plot to show how close the precision-recall curves are to different F1 scores.
- **per_class** [bool, default=False] If multi-class classification, draw the precision-recall curve for each class using a OneVsRestClassifier to compute the recall on a per-class basis. In the multi-class case, either micro or per-class must be set to True. Ignored in the binary case.
- **fill_opacity** [float, default=0.2] Specify the alpha or opacity of the fill area (0 being transparent, and 1.0 being completely opaque).
- **line_opacity** [float, default=0.8] Specify the alpha or opacity of the lines (0 being transparent, and 1.0 being completely opaque).
- **kwargs** [dict] Keyword arguments passed to the visualization base class.

**Notes**

See also:


**Attributes**

- **target_type_** [str] Either "binary" or "multiclass" depending on the type of target fit to the visualizer. If "multiclass" then the estimator is wrapped in a OneVsRestClassifier classification strategy.
- **score_** [float or dict of floats] Average precision, a summary of the plot as a weighted mean of precision at each threshold, weighted by the increase in recall from the previous threshold. In the multiclass case, a mapping of class/metric to the average precision score.
- **precision_** [array or dict of array with shape=[n_thresholds + 1]] Precision values such that element i is the precision of predictions with score >= thresholds[i] and the last element is 1. In the multiclass case, a mapping of class/metric to precision array.
**recall_** [array or dict of array with shape=[n_thresholds + 1]] Decreasing recall values such that element i is the recall of predictions with score >= thresholds[i] and the last element is 0. In the multiclass case, a mapping of class/metric to recall array.

**draw()**
Draws the precision-recall curves computed in score on the axes.

**finalize()**
Finalize the figure by adding titles, labels, and limits.

**fit(X, y=None)**
Fit the classification model; if y is multi-class, then the estimator is adapted with a OneVsRestClassifier strategy, otherwise the estimator is fit directly.

**score(X, y=None)**
Generates the Precision-Recall curve on the specified test data.

**Returns**

**score_** [float] Average precision, a summary of the plot as a weighted mean of precision at each threshold, weighted by the increase in recall from the previous threshold.

### Class Prediction Error

The class prediction error chart provides a way to quickly understand how good your classifier is at predicting the right classes.

```python
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Create classification dataset
X, y = make_classification(
    n_samples=1000, n_classes=5, n_informative=3, n_clusters_per_class=1
)
classes = ["apple", "kiwi", "pear", "banana", "orange"]

# Perform 80/20 training/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
                                                    random_state=42)

from sklearn.ensemble import RandomForestClassifier
from yellowbrick.classifier import ClassPredictionError

# Instantiate the classification model and visualizer
visualizer = ClassPredictionError(RandomForestClassifier(), classes=classes)

# Fit the training data to the visualizer
visualizer.fit(X_train, y_train)

# Evaluate the model on the test data
visualizer.score(X_test, y_test)

# Draw visualization
g = visualizer.poof()
```
Shows the balance of classes and their associated predictions.

```python
class yellowbrick.classifier.class_prediction_error.ClassPredictionError (model,
    ax=None, classes=None, **kwargs)
```

Class Prediction Error chart that shows the support for each class in the fitted classification model displayed as a stacked bar. Each bar is segmented to show the distribution of predicted classes for each class. It is initialized with a fitted model and generates a class prediction error chart on draw.

**Parameters**

- `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.

Attributes

- **score_** [float] Global accuracy score
- **predictions_** [ndarray] An ndarray of predictions whose rows are the true classes and whose columns are the predicted classes

**draw()**
Renders the class prediction error across the axis.

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

**score**(X, y, **kwargs)**
Generates a 2D array where each row is the count of the predicted classes and each column is the true class

Parameters

- **X** [ndarray or DataFrame of shape n x m] A matrix of n instances with m features
- **y** [ndarray or Series of length n] An array or series of target or class values

Returns

- **score_** [float] Global accuracy score

**Discrimination Threshold**

**Caution:** This visualizer only works for binary classification.

A visualization of precision, recall, f1 score, and queue rate with respect to the discrimination threshold of a binary classifier. The **discrimination threshold** is the probability or score at which the positive class is chosen over the negative class. Generally, this is set to 50% but the threshold can be adjusted to increase or decrease the sensitivity to false positives or to other application factors.

```python
# Load a binary classification dataset
data = load_data("spam")
target = "is_spam"
features = [col for col in data.columns if col != target]

# Extract the instances and target from the dataset
X = data[features]
y = data[target]

# Instantiate the classification model and visualizer
from sklearn.linear_model import LogisticRegression
from yellowbrick.classifier import DiscriminationThreshold
logistic = LogisticRegression()
visualizer = DiscriminationThreshold(logistic)
```
One common use of binary classification algorithms is to use the score or probability they produce to determine cases that require special treatment. For example, a fraud prevention application might use a classification algorithm to determine if a transaction is likely fraudulent and needs to be investigated in detail. In the figure above, we present an example where a binary classifier determines if an email is “spam” (the positive case) or “not spam” (the negative case). Emails that are detected as spam are moved to a hidden folder and eventually deleted.

Many classifiers use either a decision_function to score the positive class or a predict_proba function to compute the probability of the positive class. If the score or probability is greater than some discrimination threshold then the positive class is selected, otherwise, the negative class is.

Generally speaking, the threshold is balanced between cases and set to 0.5 or 50% probability. However, this threshold may not be the optimal threshold: often there is an inverse relationship between precision and recall with respect to a discrimination threshold. By adjusting the threshold of the classifier, it is possible to tune the F1 score (the harmonic mean of precision and recall) to the best possible fit or to adjust the classifier to behave optimally for the specific application. Classifiers are tuned by considering the following metrics:

- **Precision**: An increase in precision is a reduction in the number of false positives; this metric should be optimized when the cost of special treatment is high (e.g. wasted time in fraud preventing or missing an important email).

- **Recall**: An increase in recall decrease the likelihood that the positive class is missed; this metric should be optimized when it is vital to catch the case even at the cost of more false positives.

- **F1 Score**: The F1 score is the harmonic mean between precision and recall. The $fbeta$ parameter determines the relative weight of precision and recall when computing this metric, by default set to 1 or F1. Optimizing this
metric produces the best balance between precision and recall.

- **Queue Rate**: The “queue” is the spam folder or the inbox of the fraud investigation desk. This metric describes the percentage of instances that must be reviewed. If review has a high cost (e.g. fraud prevention) then this must be minimized with respect to business requirements; if it doesn’t (e.g. spam filter), this could be optimized to ensure the inbox stays clean.

In the figure above we see the visualizer tuned to look for the optimal F1 score, which is annotated as a threshold of 0.43. The model is run multiple times over multiple train/test splits in order to account for the variability of the model with respect to the metrics (shown as the fill area around the median curve).

**API Reference**

DiscriminationThreshold visualizer for probabilistic classifiers.

```python
class yellowbrick.classifier.threshold.DiscriminationThreshold(model, ax=None, n_trials=50, cv=0.1, fbeta=1.0, argmax='fScore', exclude=None, quantiles=array([0.1, 0.5, 0.9]), random_state=None, **kwargs)
```

*Bases*: yellowbrick.base.ModelVisualizer

Visualizes how precision, recall, F1 score, and queue rate change as the discrimination threshold increases. For probabilistic, binary classifiers, the discrimination threshold is the probability at which you choose the positive class over the negative. Generally this is set to 50%, but adjusting the discrimination threshold will adjust sensitivity to false positives which is described by the inverse relationship of precision and recall with respect to the threshold.

The visualizer also accounts for variability in the model by running multiple trials with different train and test splits of the data. The variability is visualized using a band such that the curve is drawn as the median score of each trial and the band is from the 10th to 90th percentile.

The visualizer is intended to help users determine an appropriate threshold for decision making (e.g. at what threshold do we have a human review the data), given a tolerance for precision and recall or limiting the number of records to check (the queue rate).

**Caution**: This method only works for binary, probabilistic classifiers.

**Parameters**

- **model** [Classification Estimator] A binary classification estimator that implements `predict_proba` or `decision_function` methods. Will raise `TypeError` if the model cannot be used with the visualizer.

- **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).
n_trials [integer, default: 50] Number of times to shuffle and split the dataset to account for noise in the threshold metrics curves. Note if cv provides > 1 splits, the number of trials will be n_trials * cv.get_n_splits()

cv [float or cross-validation generator, default: 0.1] Determines the splitting strategy for each trial. Possible inputs are:

- float, to specify the percent of the test split
- object to be used as cross-validation generator

This attribute is meant to give flexibility with stratified splitting but if a splitter is provided, it should only return one split and have shuffle set to True.

fbeta [float, 1.0 by default] The strength of recall versus precision in the F-score.

argmax [str, default: ‘fscore’] Annotate the threshold maximized by the supplied metric (see exclude for the possible metrics to use). If None, will not annotate the graph.

exclude [str or list, optional] Specify metrics to omit from the graph, can include:

- "precision"
- "recall"
- "queue_rate"
- "fscore"

All metrics not excluded will be displayed in the graph, nor will they be available in thresholds_; however, they will be computed on fit.

quantiles [sequence, default: np.array([0.1, 0.5, 0.9])] Specify the quantiles to view model variability across a number of trials. Must be monotonic and have three elements such that the first element is the lower bound, the second is the drawn curve, and the third is the upper bound. By default the curve is drawn at the median, and the bounds from the 10th percentile to the 90th percentile.

random_state [int, optional] Used to seed the random state for shuffling the data while composing different train and test splits. If supplied, the random state is incremented in a deterministic fashion for each split.

Note that if a splitter is provided, it's random state will also be updated with this random state, even if it was previously set.

kwargs [dict] Keyword arguments that are passed to the base visualizer class.

Notes

The term “discrimination threshold” is rare in the literature. Here, we use it to mean the probability at which the positive class is selected over the negative class in binary classification.

Classification models must implement either a decision_function or predict_proba method in order to be used with this class. A YellowbrickTypeError is raised otherwise.

See also:

For a thorough explanation of discrimination thresholds, see: Visualizing Machine Learning Thresholds to Make Better Business Decisions by Insight Data.

Attributes

thresholds_ [array] The uniform thresholds identified by each of the trial runs.
cv_scores_ [dict of arrays of len(thresholds_)] The values for all included metrics including the upper and lower bounds of the metrics defined by quantiles.

draw()
Draws the cv scores as a line chart on the current axes.

finalize(**kwargs)
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)
Fit is the entry point for the visualizer. Given instances described by X and binary classes described in the target y, fit performs n trials by shuffling and splitting the dataset then computing the precision, recall, f1, and queue rate scores for each trial. The scores are aggregated by the quantiles expressed then drawn.

Parameters

X [ndarray or DataFrame of shape n x m] A matrix of n instances with m features
y [ndarray or Series of length n] An array or series of target or class values. The target y must be a binary classification target.

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

Returns

self [instance] Returns the instance of the visualizer

raises: YellowbrickValueError If the target y is not a binary classification target.

4.3.7 Clustering Visualizers

Clustering models are unsupervised methods that attempt to detect patterns in unlabeled data. There are two primary classes of clustering algorithm: agglomerative clustering links similar data points together, whereas centroidal clustering attempts to find centers or partitions in the data. Yellowbrick provides the yellowbrick.cluster module to visualize and evaluate clustering behavior. Currently we provide several visualizers to evaluate centroidal mechanisms, particularly K-Means clustering, that help us to discover an optimal $K$ parameter in the clustering metric:

- Elbow Method: visualize the clusters according to some scoring function, look for an “elbow” in the curve.
- Silhouette Visualizer: visualize the silhouette scores of each cluster in a single model.
- Intercluster Distance Maps: visualize the relative distance and size of clusters.

Because it is very difficult to score a clustering model, Yellowbrick visualizers wrap scikit-learn clusterer estimators via their fit() method. Once the clustering model is trained, then the visualizer can call poof() to display the clustering evaluation metric.

Elbow Method

The KElbowVisualizer implements the “elbow” method to help data scientists select the optimal number of clusters by fitting the model with a range of values for $K$. If the line chart resembles an arm, then the “elbow” (the point of inflection on the curve) is a good indication that the underlying model fits best at that point.

To demonstrate, in the following example the KElbowVisualizer fits the KMeans model for a range of $K$ values from 4 to 11 on a sample two-dimensional dataset with 8 random clusters of points. When the model is fit with 8 clusters, we can see an “elbow” in the graph, which in this case we know to be the optimal number.
from sklearn.datasets import make_blobs

# Create synthetic dataset with 8 random clusters
X, y = make_blobs(centers=8, n_features=12, shuffle=True, random_state=42)

from sklearn.cluster import KMeans
from yellowbrick.cluster import KElbowVisualizer

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

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

Distortion Score Elbow for KMeans Clustering

By default, the scoring parameter metric is set to distortion, which computes the sum of squared distances from each point to its assigned center. However, two other metrics can also be used with the KElbowVisualizer — silhouette and calinski_harabaz. The silhouette score calculates the mean Silhouette Coefficient of all samples, while the calinski_harabaz score computes the ratio of dispersion between and within clusters.

The KElbowVisualizer also displays the amount of time to train the clustering model per $K$ as a dashed green line, but is can be hidden by setting timings=False. In the following example, we'll use the calinski_harabaz score and hide the time to fit the model.

from sklearn.cluster import KMeans
from yellowbrick.cluster import KElbowVisualizer

(continues on next page)
It is important to remember that the “elbow” method does not work well if the data is not very clustered. In this case, you might see a smooth curve and the optimal value of $K$ will be unclear.

### API Reference

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

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

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, tuple, or iterable] The k values to compute silhouette scores for. If a single integer is specified, then will compute the range (2,k). If a tuple of 2 integers is specified, then k will be in np.arange(k[0], k[1]). Otherwise, specify an iterable of integers to use as values for k.
- **metric** [string, default: "distortion"] Select the scoring metric to evaluate the clusters. The default is the mean distortion, defined by the sum of squared distances between each observation and its closest centroid. Other metrics include:
  - **distortion**: mean sum of squared distances to centers
  - **silhouette**: mean ratio of intra-cluster and nearest-cluster distance
  - **calinski_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.

**See also:**

The scikit-learn documentation for the silhouette_score and calinski_harabaz_score. The default, distortion_score, is implemented in 'yellowbrick.cluster.elbow'.

**Examples**

```python
>>> from yellowbrick.cluster import KEllbowVisualizer
>>> from sklearn.cluster import KMeans
>>> model = KEllbowVisualizer(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 silhouette 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.

```
from sklearn.datasets import make_blobs
X, y = make_blobs(centers=8)

from sklearn.cluster import MiniBatchKMeans
from yellowbrick.cluster import SilhouetteVisualizer

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.

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

Bases: yellowbrick.cluster.base.ClusteringScoreVisualizer

The Silhouette Visualizer displays the silhouette coefficient for each sample on a per-cluster basis, visually evaluating the density and separation between clusters. The score is calculated by averaging the silhouette coefficient for each sample, computed as the difference between the average intra-cluster distance and the mean nearest-cluster distance for each sample, normalized by the maximum value. This produces a score between -1 and +1, where scores near +1 indicate high separation and scores near -1 indicate that the samples may have been assigned to the wrong cluster.

In SilhouetteVisualizer plots, clusters with higher scores have wider silhouettes, but clusters that are less cohesive will fall short of the average score across all clusters, which is plotted as a vertical dotted red line.

This is particularly useful for determining cluster imbalance, or for selecting a value for K by comparing multiple visualizers.

Parameters

- **model** [a Scikit-Learn clusterer] Should be an instance of a centroidal clustering algorithm (KMeans or MiniBatchKMeans).

- **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**

```python
guideline from yellowbrick.cluster import SilhouetteVisualizer
guideline from sklearn.cluster import KMeans
guideline model = SilhouetteVisualizer(KMeans(10))
guideline model.fit(X)
guideline model.poof()
```

**Attributes**

- **silhouette_score_** [float] Mean Silhouette Coefficient for all samples. Computed via scikit-learn `sklearn.metrics.silhouette_score`.
- **silhouette_samples_** [array, shape = [n_samples]] Silhouette Coefficient for each samples. Computed via scikit-learn `sklearn.metrics.silhouette_samples`.
- **n_samples_** [integer] Number of total samples in the dataset (X.shape[0])
- **n_clusters_** [integer] Number of clusters (e.g. n_clusters or k value) passed to internal scikit-learn model.

**draw** *(labels)*

Draw the silhouettes for each sample and the average score.

**Parameters**

- **labels** [array-like] An array with the cluster label for each silhouette sample, usually computed with `predict()`. Labels are not stored on the visualizer so that the figure can be redrawn with new data.

**finalize()**

Prepare the figure for rendering by setting the title and adjusting the limits on the axes, adding labels and a legend.

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

Fits the model and generates the silhouette visualization.

**Intercluster Distance Maps**

Intercluster distance maps display an embedding of the cluster centers in 2 dimensions with the distance to other centers preserved. E.g. the closer to centers are in the visualization, the closer they are in the original feature space. The clusters are sized according to a scoring metric. By default, they are sized by membership, e.g. the number of instances that belong to each center. This gives a sense of the relative importance of clusters. Note however, that because two clusters overlap in the 2D space, it does not imply that they overlap in the original feature space.

```python
from sklearn.datasets import make_blobs
# Make 12 blobs dataset
X, y = make_blobs(centers=12, n_samples=1000, n_features=16, shuffle=True)
```
from sklearn.cluster import KMeans
from yellowbrick.cluster import InterclusterDistance

# Instantiate the clustering model and visualizer
visualizer = InterclusterDistance(KMeans(9))

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

API Reference

Implements Intercluster Distance Map visualizations.

class yellowbrick.cluster.icdm.InterclusterDistance(model=None, ax=None, min_size=400, max_size=25000, embedding='mds', scoring='membership', legend=True, legend_loc='lower left', legend_size=1.5, random_state=None, **kwargs)

Bases: yellowbrick.cluster.base.ClusteringScoreVisualizer

Intercluster distance maps display an embedding of the cluster centers in 2 dimensions with the distance to other centers preserved. E.g. the closer to centers are in the visualization, the closer they are in the original feature space. The clusters are sized according to a scoring metric. By default, they are sized by membership, e.g. the number of instances that belong to each center. This gives a sense of the relative importance of clusters. Note however, that because two clusters overlap in the 2D space, it does not imply that they overlap in the original feature space.

Parameters

model [a Scikit-Learn clusterer] Should be an instance of a centroidal clustering algorithm (or a
hierarchical algorithm with a specified number of clusters). Also accepts some other models like LDA for text clustering. If it is not a clusterer, an exception is raised.

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

**min_size** [int, default: 400] The size, in points, of the smallest cluster drawn on the graph. Cluster sizes will be scaled between the min and max sizes.

**max_size** [int, default: 25000] The size, in points, of the largest cluster drawn on the graph. Cluster sizes will be scaled between the min and max sizes.

**embedding** [default: ‘mds’] The algorithm used to embed the cluster centers in 2 dimensional space so that the distance between clusters is represented equivalently to their relationship in feature spaceself. Embedding algorithm options include:

- *mds*: multidimensional scaling
- *tsne*: stochastic neighbor embedding

**scoring** [default: ‘membership’] The scoring method used to determine the size of the clusters drawn on the graph so that the relative importance of clusters can be viewed. Scoring method options include:

- *membership*: number of instances belonging to each cluster

**legend** [bool, default: True] Whether or not to draw the size legend onto the graph, omit the legend to more easily see clusters that overlap.

**legend_loc** [str, default: “lower left”] The location of the legend on the graph, used to move the legend out of the way of clusters into open space. The same legend location options for matplotlib are used here.

See also:
https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.legend

**legend_size** [float, default: 1.5] The size, in inches, of the size legend to inset into the graph.

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

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

Notes

Currently the only two embeddings supported are MDS and TSNE. Soon to follow will be PCoA and a customized version of PCoA for LDA. The only supported scoring metric is membership, but in the future, silhouette scores and cluster diameter will be added.

In terms of algorithm support, right now any clustering algorithm that has a learned `cluster_centers_` and `labels_` attribute will work with the visualizer. In the future, we will update this to work with hierarchical clusterers that have `n_components` and LDA.

Attributes

- **cluster_centers_** [array of shape (n_clusters, n_features)] Searches for or creates cluster centers for the specified clustering algorithm.

- **embedded_centers_** [array of shape (n_clusters, 2)] The positions of all the cluster centers on the graph.
scores_. [array of shape (n_clusters,)] The scores of each cluster that determine its size on the graph.

fit_time_. [Timer] The time it took to fit the clustering model and perform the embedding.

cluster_centers_
Searches for or creates cluster centers for the specified clustering algorithm. This algorithm ensures that the centers are appropriately drawn and scaled so that distance between clusters are maintained.

draw()
Draw the embedded centers with their sizes on the visualization.

finalize()
Finalize the visualization to create an “origin grid” feel instead of the default matplotlib feel. Set the title, remove spines, and label the grid with components. This function also adds a legend from the sizes if required.

fit(X, y=None)
Fit the clustering model, computing the centers then embeds the centers into 2D space using the embedding method specified.

lax
Returns the legend axes, creating it only on demand by creating a 2” by 2” inset axes that has no grid, ticks, spines or face frame (e.g is mostly invisible). The legend can then be drawn on this axes.

transformer
Creates the internal transformer that maps the cluster center’s high dimensional space to its two dimensional space.

4.3.8 Model Selection Visualizers

Yellowbrick visualizers are intended to steer the model selection process. Generally, model selection is a search problem defined as follows: given N instances described by numeric properties and (optionally) a target for estimation, find a model described by a triple composed of features, an algorithm and hyperparameters that best fits the data. For most purposes the “best” triple refers to the triple that receives the best cross-validated score for the model type.

The yellowbrick.model_selection package provides visualizers for inspecting the performance of cross validation and hyper parameter tuning. Many visualizers wrap functionality found in sklearn.model_selection and others build upon it for performing multi-model comparisons.

The currently implemented model selection visualizers are as follows:

- **Validation Curve**: visualizes how the adjustment of a hyperparameter influences training and test scores to tune the bias/variance trade-off.

- **Learning Curve**: shows how the size of training data influences the model to diagnose if a model suffers more from variance error vs. bias error.

- **Cross Validation Scores**: displays cross-validated scores as a bar chart with average as a horizontal line.

Model selection makes heavy use of cross validation to measure the performance of an estimator. Cross validation splits a dataset into a training data set and a test data set; the model is fit on the training data and evaluated on the test data. This helps avoid a common pitfall, overfitting, where the model simply memorizes the training data and does not generalize well to new or unknown input.

There are many ways to define how to split a dataset for cross validation. For more information on how scikit-learn implements these mechanisms, please review Cross-validation: evaluating estimator performance in the scikit-learn documentation.
Validation Curve

Model validation is used to determine how effective an estimator is on data that it has been trained on as well as how generalizable it is to new input. To measure a model’s performance we first split the dataset into training and test splits, fitting the model on the training data and scoring it on the reserved test data.

In order to maximize the score, the hyperparameters of the model must be selected which best allow the model to operate in the specified feature space. Most models have multiple hyperparameters and the best way to choose a combination of those parameters is with a grid search. However, it is sometimes useful to plot the influence of a single hyperparameter on the training and test data to determine if the estimator is underfitting or overfitting for some hyperparameter values.

In our first example, we’ll explore using the `ValidationCurve` visualizer with a regression dataset and in the second, a classification dataset. Note that any estimator that implements `fit()` and `predict()` and has an appropriate scoring mechanism can be used with this visualizer.

```python
import numpy as np
from sklearn.tree import DecisionTreeRegressor
from yellowbrick.model_selection import ValidationCurve

# Load a regression dataset
data = load_data('energy')

# Specify features of interest and the target
targets = ['heating load', 'cooling load']
features = [col for col in data.columns if col not in targets]

# Extract the instances and target
X = data[features]
y = data[targets[0]]

viz = ValidationCurve(
    DecisionTreeRegressor(), param_name='max_depth',
    param_range=np.arange(1, 11), cv=10, scoring='r2'
)

# Fit and poof the visualizer
viz.fit(X, y)
viz.poof()
```
After loading and wrangling the data, we initialize the ValidationCurve with a DecisionTreeRegressor. Decision trees become more overfit the deeper they are because at each level of the tree the partitions are dealing with a smaller subset of data. One way to deal with this overfitting process is to limit the depth of the tree. The validation curve explores the relationship of the "max_depth" parameter to the R2 score with 10 shuffle split cross-validation. The param_range argument specifies the values of max_depth, here from 1 to 10 inclusive.

We can see in the resulting visualization that a depth limit of less than 5 levels severely underfits the model on this data set because the training score and testing score climb together in this parameter range, and because of the high variability of cross validation on the test scores. After a depth of 7, the training and test scores diverge, this is because deeper trees are beginning to overfit the training data, providing no generalizability to the model. However, because the cross validation score does not necessarily decrease, the model is not suffering from high error due to variance.

In the next visualizer, we will see an example that more dramatically visualizes the bias/variance tradeoff.

```python
from sklearn.svm import SVCrom sklearn.model_selection import StratifiedKFold

data = load_data('game')

target = "outcome"
features = [col for col in data.columns if col != target]

X = pd.get_dummies(data[features])
y = data[target]
```

(continues on next page)
After loading data and one-hot encoding it using the Pandas `get_dummies` function, we create a stratified k-folds cross-validation strategy. The hyperparameter of interest is the gamma of a support vector classifier, the coefficient of the RBF kernel. Gamma controls how much influence a single example has, the larger gamma is, the tighter the support vector is around single points (overfitting the model).

In this visualization we see a definite inflection point around $\gamma = 0.1$. At this point the training score climbs rapidly as the SVC memorizes the data, while the cross-validation score begins to decrease as the model cannot generalize to unseen data.

**Warning:** Note that running this example may take a long time. Even with parallelism using `n_jobs=8`, this can take several hours.

Validation curves can be performance intensive since they are training $n_{\text{params}} \times n_{\text{splits}}$ models and scoring
them. It is critically important to ensure that the specified hyperparameter range is correct, as we will see in the next example.

```python
from sklearn.neighbors import KNeighborsClassifier
cv = StratifiedKFold(4)
param_range = np.arange(3, 20, 2)

oz = ValidationCurve(
    KNeighborsClassifier(), param_name="n_neighbors",
    param_range=param_range, cv=cv, scoring="f1_weighted", n_jobs=4,
)

# Using the same game dataset as in the SVC example
oz.fit(X, y)
oz.poof()
```

The k nearest neighbors (kNN) model is commonly used when similarity is important to the interpretation of the model. Choosing k is difficult, the higher k is the more data is included in a classification, creating more complex decision topologies, whereas the lower k is, the simpler the model is and the less it may generalize. Using a validation curve seems like an excellent strategy for choosing k, and often it is. However in the example above, all we can see is a decreasing variability in the cross-validated scores.

This validation curve poses two possibilities: first, that we do not have the correct param_range to find the best k and need to expand our search to larger values. The second is that other hyperparameters (such as uniform or distance based weighting, or even the distance metric) may have more influence on the default model than k by itself does. Although validation curves can give us some intuition about the performance of a model to a single hyperparameter, grid search is required to understand the performance of a model with respect to multiple hyperparameters.
See also:

This visualizer is based on the validation curve described in the scikit-learn documentation: Validation Curves. The visualizer wraps the validation_curve function and most of the arguments are passed directly to it.

API Reference

Implements a visual validation curve for a hyperparameter.

```python
class yellowbrick.model_selection.validation_curve.ValidationCurve(model, param_name, param_range, ax=None, logx=False, groups=None, cv=None, scoring=None, n_jobs=1, pre_dispatch='all', **kwargs)
```

Bases: yellowbrick.base.ModelVisualizer

Visualizes the validation curve for both test and training data for a range of values for a single hyperparameter of the model. Adjusting the value of a hyperparameter adjusts the complexity of a model. Less complex models suffer from increased error due to bias, while more complex models suffer from increased error due to variance. By inspecting the training and cross-validated test score error, it is possible to estimate a good value for a hyperparameter that balances the bias/variance trade-off.

The visualizer evaluates cross-validated training and test scores for the different hyperparameters supplied. The curve is plotted so that the x-axis is the value of the hyperparameter and the y-axis is the model score. This is similar to a grid search with a single hyperparameter.

The cross-validation generator splits the dataset k times, and scores are averaged over all k runs for the training and test subsets. The curve plots the mean score, and the filled in area suggests the variability of cross-validation by plotting one standard deviation above and below the mean for each split.

Parameters

- **model** [a scikit-learn estimator] An object that implements fit and predict, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric.
  
  Note that the object is cloned for each validation.

- **param_name** [string] Name of the parameter that will be varied.

- **param_range** [array-like, shape (n_values,)] The values of the parameter that will be evaluated.

- **ax** [matplotlib.Axes object, optional] The axes object to plot the figure on.

- **logx** [boolean, optional] If True, plots the x-axis with a logarithmic scale.

- **groups** [array-like, with shape (n_samples,)] Optional group labels for the samples used while splitting the dataset into train/test sets.

- **cv** [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:
  
  - None, to use the default 3-fold cross-validation,
  
  - integer, to specify the number of folds.
• An object to be used as a cross-validation generator.
• An iterable yielding train/test splits.

see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.

**scoring** [string, callable or None, optional, default: None] A string or scorer callable object / function with signature scorer(estimator, X, y). See scikit-learn model evaluation documentation for names of possible metrics.

**n_jobs** [integer, optional] Number of jobs to run in parallel (default 1).

**pre_dispatch** [integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like ‘2*n_jobs’.

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

**Notes**

This visualizer is essentially a wrapper for the sklearn.model_selection.validation_curve utility, discussed in the validation curves documentation.

See also:

The documentation for the validation_curve function, which this visualizer wraps.

**Examples**

```python
>>> import numpy as np
>>> from yellowbrick.model_selection import ValidationCurve
>>> from sklearn.svm import SVC
>>> pr = np.logspace(-6,-1,5)
>>> model = ValidationCurve(SVC(), param_name="gamma", param_range=pr)
>>> model.fit(X, y)
>>> model.poof()
```

**Attributes**

- **train_scores_** [array, shape (n_ticks, n_cv_folds)] Scores on training sets.
- **train_scores_mean_** [array, shape (n_ticks,)] Mean training data scores for each training split
- **train_scores_std_** [array, shape (n_ticks,)] Standard deviation of training data scores for each training split
- **test_scores_** [array, shape (n_ticks, n_cv_folds)] Scores on test set.
- **test_scores_mean_** [array, shape (n_ticks,)] Mean test data scores for each test split
- **test_scores_std_** [array, shape (n_ticks,)] Standard deviation of test data scores for each test split

**draw** (**kwargs**)

Renders the training and test curves.
finalize(**kwargs)
Add the title, legend, and other visual final touches to the plot.

fit(X, y=None)
Fits the validation curve with the wrapped estimator and parameter array to the specified data. Draws training and test score curves and saves the scores to the visualizer.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

self [instance] Returns the instance of the validation curve visualizer for use in pipelines and other sequential transformers.

Learning Curve

A learning curve shows the relationship of the training score vs the cross validated test score for an estimator with a varying number of training samples. This visualization is typically used to show two things:

1. How much the estimator benefits from more data (e.g. do we have “enough data” or will the estimator get better if used in an online fashion).
2. If the estimator is more sensitive to error due to variance vs. error due to bias.

Consider the following learning curves (generated with Yellowbrick, but from Plotting Learning Curves in the scikit-learn documentation):

If the training and cross validation scores converge together as more data is added (shown in the left figure), then the model will probably not benefit from more data. If the training score is much greater than the validation score (as shown in the right figure) then the model probably requires more training examples in order to generalize more effectively.

The curves are plotted with the mean scores, however variability during cross-validation is shown with the shaded areas that represent a standard deviation above and below the mean for all cross-validations. If the model suffers from
error due to bias, then there will likely be more variability around the training score curve. If the model suffers from error due to variance, then there will be more variability around the cross validated score.

**Note:** Learning curves can be generated for all estimators that have `fit()` and `predict()` methods as well as a single scoring metric. This includes classifiers, regressors, and clustering as we will see in the following examples.

### Classification

In the following example we show how to visualize the learning curve of a classification model. After loading a `DataFrame` and performing categorical encoding, we create a `StratifiedKFold` cross-validation strategy to ensure all of our classes in each split are represented with the same proportion. We then fit the visualizer using the `f1_weighted` scoring metric as opposed to the default metric, accuracy, to get a better sense of the relationship of precision and recall in our classifier.

```python
import numpy as np
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import StratifiedKFold
from yellowbrick.model_selection import LearningCurve

# Load a classification data set
data = load_data('game')

# Specify the features of interest and the target
target = "outcome"
features = [col for col in data.columns if col != target]

# Encode the categorical data with one-hot encoding
X = pd.get_dummies(data[features])
y = data[target]

# Create the learning curve visualizer
cv = StratifiedKFold(12)
sizes = np.linspace(0.3, 1.0, 10)

viz = LearningCurve(MultinomialNB(), cv=cv, train_sizes=sizes,
                     scoring='f1_weighted', n_jobs=4)

# Fit and poof the visualizer
datafit(X, y)
viz.poof()
```
This learning curve shows high test variability and a low score up to around 30,000 instances, however after this level the model begins to converge on an F1 score of around 0.6. We can see that the training and test scores have not yet converged, so potentially this model would benefit from more training data. Finally, this model suffers primarily from error due to variance (the CV scores for the test data are more variable than for training data) so it is possible that the model is overfitting.

**Regression**

Building a learning curve for a regression is straightforward and very similar. In the below example, after loading our data and selecting our target, we explore the learning curve score according to the coefficient of determination or R2 score.

```python
from sklearn.linear_model import RidgeCV

# Load a regression dataset
data = load_data('energy')

# Specify features of interest and the target
targets = ['heating load', 'cooling load']
features = [col for col in data.columns if col not in targets]

# Extract the instances and target
X = data[features]
y = data[targets[0]]

# Create the learning curve visualizer, fit and poof
(continues on next page)
This learning curve shows a very high variability and much lower score until about 350 instances. It is clear that this model could benefit from more data because it is converging at a very high score. Potentially, with more data and a larger alpha for regularization, this model would become far less variable in the test data.

**Clustering**

Learning curves also work for clustering models and can use metrics that specify the shape or organization of clusters such as silhouette scores or density scores. If the membership is known in advance, then rand scores can be used to compare clustering performance as shown below:

```python
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Create a dataset of blobs
X, y = make_blobs(n_samples=1000, centers=5)

viz = LearningCurve(KMeans(), train_sizes=sizes, scoring="adjusted_rand_score")
```
Unfortunately, with random data these curves are highly variable, but serve to point out some clustering-specific items. First, note the y-axis is very narrow, roughly speaking these curves are converged and actually the clustering algorithm is performing very well. Second, for clustering, convergence for data points is not necessarily a bad thing; in fact we want to ensure as more data is added, the training and cross-validation scores do not diverge.

**See also:**

This visualizer is based on the validation curve described in the scikit-learn documentation: Learning Curves. The visualizer wraps the `learning_curve` function and most of the arguments are passed directly to it.

**API Reference**

Implements a learning curve visualization for model selection.
class yellowbrick.model_selection.learning_curve.LearningCurve(model,
ax=None,
groups=None,
train_sizes=array([0.1, 0.325, 0.55, 0.775, 1.]),
cv=None,
score=None,
explot_incremental_learning=False,
n_jobs=1,
pre_dispatch='all',
shuffle=False,
random_state=None,
**kwargs)

Bases: yellowbrick.base.ModelVisualizer

Visualizes the learning curve for both test and training data for different training set sizes. These curves can act as a proxy to demonstrate the implied learning rate with experience (e.g. how much data is required to make an adequate model). They also demonstrate if the model is more sensitive to error due to bias vs. error due to variance and can be used to quickly check if a model is overfitting.

The visualizer evaluates cross-validated training and test scores for different training set sizes. These curves are plotted so that the x-axis is the training set size and the y-axis is the score.

The cross-validation generator splits the whole dataset k times, scores are averaged over all k runs for the training subset. The curve plots the mean score for the k splits, and the filled in area suggests the variability of the cross-validation by plotting one standard deviation above and below the mean for each split.

Parameters

- **model** [a scikit-learn estimator] An object that implements fit and predict, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric.
  
  Note that the object is cloned for each validation.

- **ax** [matplotlib.Axes object, optional] The axes object to plot the figure on.

- **groups** [array-like, with shape (n_samples,)] Optional group labels for the samples used while splitting the dataset into train/test sets.

- **train_sizes** [array-like, shape (n_ticks,)] default: np.linspace(0.1,1.0,5)
  
  Relative or absolute numbers of training examples that will be used to generate the learning curve. If the dtype is float, it is regarded as a fraction of the maximum size of the training set, otherwise it is interpreted as absolute sizes of the training sets.

- **cv** [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:
  
  - None, to use the default 3-fold cross-validation,
  
  - integer, to specify the number of folds.
  
  - An object to be used as a cross-validation generator.
  
  - An iterable yielding train/test splits.

  see the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.
scoring [string, callable or None, optional, default: None] A string or scorer callable object / function with signature scorer(estimator, X, y). See scikit-learn model evaluation documentation for names of possible metrics.

exploit_incremental_learning [boolean, default: False] If the estimator supports incremental learning, this will be used to speed up fitting for different training set sizes.

n_jobs [integer, optional] Number of jobs to run in parallel (default 1).

pre_dispatch [integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like `2*n_jobs`.

shuffle [boolean, optional] Whether to shuffle training data before taking prefixes of it based on `train_sizes`.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Used when shuffle is True.

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

Notes

This visualizer is essentially a wrapper for the sklearn.model_selection.learning_curve utility, discussed in the validation curves documentation.

See also:

The documentation for the learning_curve function, which this visualizer wraps.

Examples

```python
>>> from yellowbrick.model_selection import LearningCurve
>>> from sklearn.naive_bayes import GaussianNB
>>> model = LearningCurve(GaussianNB())
>>> model.fit(X, y)
>>> model.poof()
```

Attributes

- `train_sizes_` [array, shape = (n_unique_ticks,), dtype int] Numbers of training examples that has been used to generate the learning curve. Note that the number of ticks might be less than n_ticks because duplicate entries will be removed.

- `train_scores_` [array, shape (n_ticks, n_cv_folds)] Scores on training sets.

- `train_scores_mean_` [array, shape (n_ticks,)] Mean training data scores for each training split

- `train_scores_std_` [array, shape (n_ticks,)] Standard deviation of training data scores for each training split

- `test_scores_` [array, shape (n_ticks, n_cv_folds)] Scores on test set.

- `test_scores_mean_` [array, shape (n_ticks,)] Mean test data scores for each test split
test_scores_std_ [array, shape (n_ticks,)] Standard deviation of test data scores for each test split

draw (**kwargs)
Renders the training and test learning curves.

finalize (**kwargs)
Add the title, legend, and other visual final touches to the plot.

fit (X, y=None)
Fits the learning curve with the wrapped model to the specified data. Draws training and test score curves and saves the scores to the estimator.

Parameters

X [array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

self [instance] Returns the instance of the learning curve visualizer for use in pipelines and other sequential transformers.

Cross Validation Scores

Generally we determine whether a given model is optimal by looking at it’s F1, precision, recall, and accuracy (for classification), or it’s coefficient of determination (R2) and error (for regression). However, real world data is often distributed somewhat unevenly, meaning that the fitted model is likely to perform better on some sections of the data than on others. Yellowbrick’s CVScores visualizer enables us to visually explore these variations in performance using different cross validation strategies.

Cross Validation

Cross-validation starts by shuffling the data (to prevent any unintentional ordering errors) and splitting it into $k$ folds. Then $k$ models are fit on $\frac{k-1}{k}$ of the data (called the training split) and evaluated on $\frac{1}{k}$ of the data (called the test split). The results from each evaluation are averaged together for a final score, then the final model is fit on the entire dataset for operationalization.
In Yellowbrick, the CVScores visualizer displays cross-validated scores as a bar chart (one bar for each fold) with the average score across all folds plotted as a horizontal dotted line.

**Classification**

In the following example we show how to visualize cross-validated scores for a classification model. After loading a DataFrame, we create a StratifiedKFold cross-validation strategy to ensure all of our classes in each split are represented with the same proportion. We then fit the CVScores visualizer using the f1_weighted scoring metric as opposed to the default metric, accuracy, to get a better sense of the relationship of precision and recall in our classifier across all of our folds.

```python
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import StratifiedKFold
from yellowbrick.model_selection import CVScores

# 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

# Create a new figure and axes
_, ax = plt.subplots()

# Create a cross-validation strategy
cv = StratifiedKFold(12)

# Create the cv score visualizer
oz = CVScores(
    MultinomialNB(), ax=ax, cv=cv, scoring='f1_weighted'
)

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

Our resulting visualization shows that while our average cross-validation score is quite high, there are some splits for which our fitted MultinomialNB classifier performs significantly less well.
Regression

In this next example we show how to visualize cross-validated scores for a regression model. After loading our energy data into a DataFrame, we instantiate a simple KFold cross-validation strategy. We then fit the CVScores visualizer using the r2 scoring metric, to get a sense of the coefficient of determination for our regressor across all of our folds.

```python
from sklearn.linear_model import Ridge
from sklearn.model_selection import KFold

# Load the regression data set
data = load_data("energy")

# Specify the features of interest and the target
targets = ["heating load", "cooling load"]
features = [col for col in data.columns if col not in targets]

# Extract the instances and target
X = data[features]
y = data[targets[1]]

# Create a new figure and axes
_, ax = plt.subplots()
```

(continues on next page)
As with our classification CVScores visualization, our regression visualization suggests that our Ridge regressor performs very well (e.g. produces a high coefficient of determination) across nearly every fold, resulting in another fairly high overall R² score.
CVScores displays cross-validated scores as a bar chart, with the average of the scores plotted as a horizontal line.

**Parameters**

- **model** [a scikit-learn estimator] An object that implements fit and predict, can be a classifier, regressor, or clusterer so long as there is also a valid associated scoring metric. Note that the object is cloned for each validation.
- **ax** [matplotlib.Axes object, optional] The axes object to plot the figure on.
- **cv** [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:
  - None, to use the default 3-fold cross-validation,
  - integer, to specify the number of folds.
  - An object to be used as a cross-validation generator.
  - An iterable yielding train/test splits.
  See the scikit-learn cross-validation guide for more information on the possible strategies that can be used here.
- **scoring** [string, callable or None, optional, default: None] A string or scorer callable object / function with signature scorer(estimator, X, y).
  See scikit-learn cross-validation guide for more information on the possible metrics that can be used.
- **kwargs** [dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

**Notes**

This visualizer is a wrapper for sklearn.model_selection.cross_val_score.
Refer to the scikit-learn cross-validation guide for more details.

**Examples**

```python
>>> from sklearn import datasets, svm
>>> iris = datasets.load_iris()
>>> clf = svm.SVC(kernel='linear', C=1)
>>> X = iris.data
>>> y = iris.target
>>> visualizer = CVScores(model=clf, cv=5, scoring='f1_macro')
>>> visualizer.fit(X, y)
>>> visualizer.poof()
```

- **draw(** **kwargs**)
  Creates the bar chart of the cross-validated scores generated from the fit method and places a dashed horizontal line that represents the average value of the scores.

- **finalize(** **kwargs**)
  Add the title, legend, and other visual final touches to the plot.
**fit**(X, y, **kwargs)
Fits the learning curve with the wrapped model to the specified data. Draws training and test score curves and saves the scores to the estimator.

**Parameters**

- X [array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.
- y [array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

**Returns**

self [instance]

### 4.3.9 Text Modeling Visualizers

Yellowbrick provides the `yellowbrick.text` module for text-specific visualizers. The `TextVisualizer` class specifically deals with datasets that are corpora and not simple numeric arrays or DataFrames, providing utilities for analyzing word dispersion and distribution, showing document similarity, or simply wrapping some of the other standard visualizers with text-specific display properties.

We currently have three 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
- **Dispersion Plot**: plot the dispersion of target words throughout a corpus

Note that the examples in this section require a corpus of text data, see [loading a text corpus](#) for more information.

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

**Note:** 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.
```python
class 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 bit 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/
    - 56d62a53c1808113ff8b7f1f.txt
    - 5745a9c7c180810be6ef70b.txt
  - cinema/
    - 56d629b5c1808113ff8b7d8f.txt
    - 57408e54c180810be6e574c8.txt
  - cooking/
    - 56d62b25c1808113ff8813b.txt
    - 573f0728c180810be6e2575c.txt

(continues on next page)
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:

```python
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.

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

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

visualizer = FreqDistVisualizer(features=features)
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:

```python
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”.

```python
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()
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.

```python
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

```python
# 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.

```python
# 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.

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

![TSNE Projection of 448 Documents](image)

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

![TSNE Projection of 448 Documents](image)
This means we don’t have to use class labels at all. Instead we can use cluster membership from K-Means to label each document. This will allow us to look for clusters of related text by their contents:

```python
# 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.

```python
class yellowbrick.text.tsne.TSNEVisualizer(ax=None, decompose='svd', decompose_by=50, labels=None, classes=None, colors=None, colormap=None, random_state=None, alpha=0.7, **kwargs)
```

**Bases:** `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/](https://lvdmaaten.github.io/tsne/)

**Parameters**
ax [matplotlib axes] The axes to plot the figure on.

de 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.

de decompose_by [int, default: 50] Specify the number of components for preliminary decomposition, by default this is 50; the more components, the slower TSNE will be.

labels [list of strings] The names of the classes in the target, used to create a legend. Labels must match names of classes in sorted order.

colors [list or tuple of colors] Specify the colors for each individual class

colormap [string or matplotlib cmap] Sequential colormap for continuous target

random_state [int, RandomState instance or None, optional, default: None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. The random state is applied to the preliminary decomposition as well as tSNE.

alpha [float, default: 0.7] Specify a transparency where 1 is completely opaque and 0 is completely transparent. This property makes densely clustered points more visible.

kwargs [dict] Pass any additional keyword arguments to the TSNE transformer.

NULL_CLASS = None

draw (points, target=None, **kwargs)
   Called from the fit method, this method draws the TSNE scatter plot, from a set of decomposed points in 2 dimensions. This method also accepts a third dimension, target, which is used to specify the colors of each of the points. If the target is not specified, then the points are plotted as a single cloud to show similar documents.

finalize (**kwargs)
   Finalize the drawing by adding a title and legend, and removing the axes objects that do not convey information about TNSE.

fit (X, y=None, **kwargs)
   The fit method is the primary drawing input for the TSNE projection since the visualization requires both X and an optional y value. The fit method expects an array of numeric vectors, so text documents must be vectorized before passing them to this method.

Parameters

   X [ndarray or DataFrame of shape n x m] A matrix of n instances with m features representing the corpus of vectorized documents to visualize with tSNE.

   y [ndarray or Series of length n] An optional array or series of target or class values for instances. If this is specified, then the points will be colored according to their class. Often cluster labels are passed in to color the documents in cluster space, so this method is used both for classification and clustering methods.

   kwargs [dict] Pass generic arguments to the drawing method

Returns

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

make_transformer (decompose='svd', decompose_by=50, tsne_kwargs={})
   Creates an internal transformer pipeline to project the data set into 2D space using TSNE, applying an
pre-decomposition technique ahead of embedding if necessary. This method will reset the transformer on the class, and can be used to explore different decompositions.

**Parameters**

- **decompose** [string or None, default: 'svd'] A preliminary decomposition is often used prior to TSNE to make the projection faster. Specify "svd" for sparse data or "pca" for dense data. If decompose is None, the original data set will be used.

- **decompose_by** [int, default: 50] Specify the number of components for preliminary decomposition, by default this is 50; the more components, the slower TSNE will be.

**Returns**

- **transformer** [Pipeline] Pipelined transformer for TSNE projections

**Dispersion Plot**

A word’s importance can be weighed by its dispersion in a corpus. Lexical dispersion is a measure of a word’s homogeneity across the parts of a corpus. This plot notes the occurrences of a word and how many words from the beginning of the corpus it appears.

```python
from yellowbrick.text import DispersionPlot

After importing the visualizer, we can load the corpus

```
API Reference

Implementation of lexical dispersion for text visualization

class yellowbrick.text.dispersion.DispersionPlot (target_words, ax=None, colors=None, ignore_case=False, annotate_docs=False, labels=None, colormap=None, **kwargs)

Bases: yellowbrick.text.base.TextVisualizer

DispersionPlotVisualizer allows for visualization of the lexical dispersion of words in a corpus. Lexical dispersion is a measure of a word’s homogeneity across the parts of a corpus. This plot notes the occurrences of a word and how many words from the beginning it appears.

Parameters

- **target_words** [list] A list of target words whose dispersion across a corpus passed at fit will be visualized.
- **ax** [matplotlib axes, default: None] The axes to plot the figure on.
- **labels** [list of strings] The names of the classes in the target, used to create a legend. Labels must match names of classes in sorted order.
- **colors** [list or tuple of colors] Specify the colors for each individual class
- **colormap** [string or matplotlib cmap] Qualitative colormap for discrete target
- **ignore_case** [boolean, default: False] Specify whether input will be case-sensitive.
annotate_docs [boolean, default: False] Specify whether document boundaries will be displayed. Vertical lines are positioned at the end of each document.

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.

NULL_CLASS = None
draw (points, target=None, **kwargs)
   Called from the fit method, this method creates the canvas and draws the plot on it. Parameters ——— kwags: generic keyword arguments.

finalize (**kwargs)
   The finalize method executes any subclass-specific axes finalization steps. The user calls poof & poof calls finalize. Parameters ———- kwags: generic keyword arguments.

fit (X, y=None, **kwargs)
   The fit method is the primary drawing input for the dispersion visualization.

Parameters

   X [list or generator] Should be provided as a list of documents or a generator that yields a list of documents that contain a list of words in the order they appear in the document.

   y [ndarray or Series of length n] An optional array or series of target or class values for instances. If this is specified, then the points will be colored according to their class.

   kwargs [dict] Pass generic arguments to the drawing method

Returns

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

4.3.10 Yellowbrick Contrib

The yellowbrick.contrib package contains a variety of extra tools and experimental visualizers that are outside of core support or are still in development. Here is a listing of the contrib modules currently available:

DecisionBoundaries Vizualizer

The DecisionBoundariesVisualizer is a bivariate data visualization algorithm that plots the decision boundaries of each class.

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_classification

# Create dummy data
X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=1, n_clusters_per_class=1)

rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

data_set = make_moons(noise=0.3, random_state=0)
```

(continues on next page)
X, y = data_set
X = StandardScaler().fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.4, random_state=42)

from sklearn.neighbors import KNeighborsClassifier
from yellowbrick.contrib.classifier import DecisionViz

viz = DecisionViz(KNeighborsClassifier(3), title="Nearest Neighbors", features=['Feature One', 'Feature Two'], classes=['A', 'B'])
viz.fit(X_train, y_train)
viz.draw(X_test, y_test)
viz.poof(outpath="images/knn_decisionviz.png")

---

**API Reference**

class yellowbrick.contrib.classifier.boundaries.DecisionBoundariesVisualizer(**kwargs)

Bases: yellowbrick.classifier.base.ClassificationScoreVisualizer

DecisionBoundariesVisualizer is a bivariate data visualization algorithm that plots the decision boundaries of each class.

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

x [string, default: None] The feature name that corresponds to a column name or index position 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 position in the matrix that will be plotted against the y-axis

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.

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

show_scatter [boolean, default: True] If boolean is True, then a scatter plot with points will be drawn on top of the decision boundary graph

step_size [float percentage, default: 0.0025] Determines the step size for creating the numpy meshgrid that will later become the foundation of the decision boundary graph. The default value of 0.0025 means that the step size for constructing the meshgrid will be 0.25% of differences of the max and min of x and y for each feature.

markers [iterable of strings, default: ,od*vh+] Matplotlib style markers for points on the scatter plot points

pcolormesh_alpha [float, default: 0.8] Sets the alpha transparency for the meshgrid of model boundaries

scatter_alpha [float, default: 1.0] Sets the alpha transparency for the scatter plot points

title [string, default: stringified feature_one and feature_two] Sets the title of the visualization

kwarg [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=None, **kwargs)
Called from the fit method, this method creates a decision boundary plot, and if self.scatter is True, it will scatter plot that draws each instance as a class or target colored point, whose location is determined by the feature data set.

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

Parameters

kwarg: generic keyword arguments.

fit (X, y=None, **kwargs)
The fit method is the primary drawing input for the decision boundaries visualization since it has both the X and y data required for the viz and the transform method does not.

Parameters

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

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

kwarg [dict] Pass generic arguments to the drawing method

Returns

self [instance] Returns the instance of the visualizer
**fit_draw** (*X*, *y=None, **kwargs*)
Fits a transformer to *X* and *y* then returns visualization of features or fitted model.

**fit_draw_poof** (*X*, *y=None, **kwargs*)
Fits a transformer to *X* and *y* then returns visualization of features or fitted model. Then calls poof to finalize.

### StatsModels Visualizers

A basic wrapper for statsmodels that emulates a scikit-learn estimator.

```python
class yellowbrick.contrib.statsmodels.base.StatsModelsWrapper(glm_partial,
    stated_estimator_type='regressor',
    scorer=<function r2_score>)
```

**Bases:** `sklearn.base.BaseEstimator`

Wrap a statsmodels GLM as a sklearn (fake) BaseEstimator for YellowBrick.

### Examples

First import the external libraries and helper utilities:

```python
>>> import statsmodels.api as sm
>>> from functools import partial
```

Instantiate a partial with the statsmodels API:

```python
>>> glm_gaussian_partial = partial(sm.GLM, family=sm.families.Gaussian())
>>> sm_est = StatsModelsWrapper(glm_gaussian_partial)
```

Create a Yellowbrick visualizer to visualize prediction error:

```python
>>> visualizer = PredictionError(sm_est)
>>> visualizer.fit(X_train, y_train)
>>> visualizer.score(X_test, y_test)
```

For statsmodels usage, calling `.summary()` etc:

```python
>>> gaussian_model = glm_gaussian_partial(y_train, X_train)
```

#### fit (*X*, *y*)
Pretend to be a sklearn estimator, fit is called on creation

#### predict (*X*)

#### score (*X*, *y*)

### Scatter Plot Visualizer

Sometimes for feature analysis you simply need a scatter plot to determine the distribution of data. Machine learning operates on high dimensional data, so the number of dimensions has to be filtered. As a result these visualizations are typically used as the base for larger visualizers; however you can also use them to quickly plot data during ML analysis.

A scatter visualizer simply plots two features against each other and colors the points according to the target. This can be useful in assessing the relationship of pairs of features to an individual target.
# Load the classification data set
```python
data = load_data("occupancy")
```

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

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

```python
from yellowbrick.contrib.scatter import ScatterVisualizer

visualizer = ScatterVisualizer(x="light", y="CO2", classes=classes)
visualizer.fit(X, y)
visualizer.transform(X)
visualizer.poof()
```

## API Reference

Implements a 2D scatter plot for feature analysis.
class yellowbrick.contrib.scatter.ScatterVisualizer(ax=None, x=None, y=None, features=None, classes=None, color=None, colormap=None, markers=None, alpha=1.0, **kwargs)

Bases: 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 position 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 position in the matrix that will be plotted against the y-axis

features [a list of two feature names to use, default: None] List of two features that correspond to the columns in the array. The order of the two features correspond to X and Y axes on the graph. More than two feature names or columns will raise an error. If a DataFrame is passed to fit and features is None, feature names are selected that are the columns of the DataFrame.

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

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

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

markers [iterable of strings, default: ,+o*vhd] Matplotlib style markers for points on the scatter plot points

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

**kwargs : keyword arguments passed to the super class.

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

draw (X, y, **kwargs)
Called from the fit method, this method creates a scatter plot that draws each instance as a class or target colored point, whose location is determined by the feature data set.

finalize (**kwargs)
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

**Missing Values**

MissingValues visualizers are a variant of feature visualizers that specifically show places in a dataset that have missing values (numpy NaN).

- **MissingValues Bar**: visualize the count of missing values by feature.
- **MissingValues Dispersion**: visualize the position of missing values by position in the index.

**MissingValues Bar**

The MissingValues Bar visualizer creates a bar graph that counts the number of missing values per feature column. If the target y is supplied to fit, then produces a stacked bar chart.

Setup

```python
import numpy as np
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=400, n_features=10, n_informative=2, n_redundant=3,
    n_classes=2, n_clusters_per_class=2, random_state=854
)
# assign some NaN values
X[X > 1.5] = np.nan
features = ["Feature {0}".format(str(n)) for n in range(10)]
```

**Without Targets Supplied**

```python
from yellowbrick.contrib.missing import MissingValuesBar

viz = MissingValuesBar(features=features)
viz.fit(X)
viz.poof()
```
With Targets (y) Supplied

```python
from yellowbrick.contrib.missing import MissingValuesBar

viz = MissingValuesBar(features=features)
viz.fit(X, y=y)  # supply the targets via y
viz.poof()
```
Bar visualizer of missing values by column.

class yellowbrick.contrib.missing.bar.MissingValuesBar

Bases: yellowbrick.contrib.missing.base.MissingDataVisualizer

The MissingValues Bar visualizer creates a bar graph that lists the total count of missing values for each selected feature column.

When y targets are supplied to fit, the output is a stacked bar chart where each color corresponds to the total NaNs for the feature in that column.

Parameters

alpha [float, default: 0.5] A value for bending elements with the background.

marker [matplotlib marker, default: '|'] The marker used for each element coordinate in the plot.

color [str, default: 'black'] The color for drawing the bar chart when the y targets are not passed to fit.

colors [list, default: None] The color palette for drawing a stack bar chart when the y targets are passed to fit.

classes [list, default: None] A list of class names for the legend. If classes is None and a y value is passed to fit then the classes are selected from the target vector.
**kwargs [dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.

**Examples**

```python
>>> from yellowbrick.contrib.missing import MissingValuesBar
>>> visualizer = MissingValuesBar()
>>> visualizer.fit(X, y=y)
>>> visualizer.poof()
```

**Attributes**

- `features_` [np.array] The feature labels ranked according to their importance
- `classes_` [np.array] The class labels for each of the target values

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

Called from the fit method, this method generated a horizontal bar plot.

If y is none, then draws a simple horizontal bar chart. If y is not none, then draws a stacked horizontal bar chart for each nan count per target values.

**draw_stacked_bar** *(nan_col_counts)*

Draws a horizontal stacked bar chart with different colors for each count of nan values per label.

**finalize** *(**kwargs)*

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

**Parameters**

- **kwargs: generic keyword arguments.**

**get_nan_col_counts** *(**kwargs)*

---

**MissingValues Dispersion**

The MissingValues Dispersion visualizer creates a chart that maps the position of missing values by the order of the index.

**Setup**

```python
import numpy as np
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=400, n_features=10, n_informative=2, n_redundant=3,
    n_classes=2, n_clusters_per_class=2, random_state=854
)

# assign some NaN values
X[X > 1.5] = np.nan
features = ["Feature {}".format(str(n)) for n in range(10)]
```
Without Targets Supplied

```python
from yellowbrick.contrib.missing import MissingValuesDispersion

viz = MissingValuesDispersion(features=features)
viz.fit(X)
viz.poof()
```

With Targets (y) Supplied

```python
from yellowbrick.contrib.missing import MissingValuesDispersion

viz = MissingValuesDispersion(features=features)
viz.fit(X, y=y)  # supply the targets via y
viz.poof()
```
**API Reference**

Dispersion visualizer for locations of missing values by column against index position.

```python
class yellowbrick.contrib.missing.dispersion.MissingValuesDispersion(alpha=0.5, marker='|', classes=None, **kwargs)
```

**Bases:** yellowbrick.contrib.missing.base.MissingDataVisualizer

The Missing Values Dispersion visualizer shows the locations of missing (nan) values in the feature dataset by the order of the index.

When y targets are supplied to fit, the output dispersion plot is color coded according to the target y that the element refers to.

**Parameters**

- **alpha** [float, default: 0.5] A value for bending elements with the background.
- **marker** [matplotlib marker, default: |] The marker used for each element coordinate in the plot
- **classes** [list, default: None] A list of class names for the legend. If classes is None and a y value is passed to fit then the classes are selected from the target vector.
- **kwargs** [dict] Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.
Examples

```python
>>> from yellowbrick.contrib.missing import MissingValuesDispersion
>>> visualizer = MissingValuesDispersion()
>>> visualizer.fit(X, y=y)
>>> visualizer.poof()
```

Attributes

- `features_` [np.array] The feature labels ranked according to their importance
- `classes_` [np.array] The class labels for each of the target values

```python
def draw(X, y, **kwargs)
.. _draw:
    Called from the fit method, this method creates a scatter plot that draws each instance as a class or target colored point, whose location is determined by the feature data set.
    If y is not None, then it draws a scatter plot where each class is in a different color.

```python
def draw_multi_dispersion_chart(nan_locs)
.. _draw_multi:
    Draws a multi dimensional dispersion chart, each color corresponds to a different target variable.

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

```python
def get_nan_locs(**kwargs)
.. _get:
    Gets the locations of nans in feature data and returns the coordinates in the matrix.
```

4.3.11 Colors and Style

Yellowbrick believes that visual diagnostics are more effective if visualizations are appealing. As a result, we have borrowed familiar styles from Seaborn and use the new matplotlib 2.0 styles. We hope that these out-of-the-box styles will make your visualizations publication ready, though you can also still customize your own look and feel by directly modifying the visualizations with matplotlib.

For most visualizers, Yellowbrick prioritizes color in its visualizations. There are two types of color sets that can be provided to a visualizer: a palette and a sequence. Palettes are discrete color values usually of a fixed length and are typically used for classification or clustering by showing each class, cluster, or topic. Sequences are continuous color values that do not have a fixed length but rather a range and are typically used for regression or clustering, showing all possible values in the target or distances between items in clusters.

In order to make the distinction easy, most matplotlib colors (both palettes and sequences) can be referred to by name. A complete listing can be imported as follows:

```python
import matplotlib.pyplot as plt
from yellowbrick.style.palettes import PALETTES, SEQUENCES, color_palette
```

Palettes and sequences can be passed to visualizers as follows:

```python
visualizer = Visualizer(color="bold")
```

Refer to the API listing of each visualizer for specifications on how the color argument is handled. In the next two sections, we will show every possible color palette and sequence currently available in Yellowbrick.
Color Palettes

Color palettes are discrete color lists that have a fixed length. The most common palettes are ordered as “blue”, “green”, “red”, “maroon”, “yellow”, “cyan”, and an optional “key”. This allows you to specify these named colors by the first character, e.g. ‘bgrmyck’ for matplotlib visualizations.

To change the global color palette, use the \texttt{set\_palette} function as follows:

\begin{verbatim}
from yellowbrick.style import set_palette
set_palette('flatui')
\end{verbatim}

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:

\begin{verbatim}
# ['blue', 'green', 'red', 'maroon', 'yellow', 'cyan']
for palette in PALETTES.keys():
    color_palette(palette).plot()
    plt.title(palette, loc='left')
\end{verbatim}
Color Sequences

Color sequences are continuous representations of color and are usually defined as a fixed number of steps between a minimum and maximal value. Sequences must be created with a total number of bins (or length) before plotting to ensure that the values are assigned correctly. In the listing below, each sequence is shown with varying lengths to describe the range of colors in detail.

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

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

```python
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](image)
![Spectral - 4](image)
![Spectral - 5](image)
![Spectral - 6](image)
![Spectral - 7](image)
4.3. Visualizers and API
PiYG - 9

PiYG - 10

PiYG - 11

YlOrBr - 3

YlOrBr - 4

YlOrBr - 5

YlOrBr - 6
4.3. Visualizers and API
API Reference

yellowbrick.style.colors module

Colors and color helpers brought in from an alternate library. See https://bl.ocks.org/mbostock/5577023

class yellowbrick.style.colors.ColorMap (colors='flatui', shuffle=False)

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 multiply 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 rcmod.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.

```python
yellowbrick.style.palettes.set_color_codes(palette='accent')
```

Change how matplotlib color shorthands are interpreted.

Calling this will change how shorthand codes like “b” or “g” are interpreted by matplotlib in subsequent plots.

**Parameters**

  **palette** [str] Named yellowbrick palette to use as the source of colors.

**See also:**

  * **set_palette** Color codes can also be set through the function that sets the matplotlib color cycle.

**yellowbrick.style.rcmod module**

Modifies the matplotlib rcParams in order to make yellowbrick more appealing. This has been modified from Seaborn’s rcmod.py: github.com/mwaskom/seaborn in order to alter the matplotlib rc dictionary on the fly.

**NOTE:** matplotlib 2.0 styles mean we can simply convert this to a stylesheet!

```python
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_prepend)] 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).

**Note:** Many examples utilize data from the UCI Machine Learning repository. In order to run the accompanying code, make sure to follow the instructions in *Example Datasets* to download and load the required data.

A guide to finding the visualizer you’re looking for: generally speaking, visualizers can be data visualizers which visualize instances relative to the model space; score visualizers which visualize model performance; model selection visualizers which compare multiple model forms against each other; and application specific-visualizers. This can be a bit confusing, so we’ve grouped visualizers according to the type of analysis they are well suited for.

Feature analysis visualizers are where you’ll find the primary implementation of data visualizers. Regression, classification, and clustering analysis visualizers can be found in their respective libraries. Finally, visualizers for text analysis are also available in Yellowbrick! Other utilities, such as styles, best fit lines, and Anscombe’s visualization, can also be found in the links above.
4.4 User Testing Instructions

We are looking for people to help us Alpha test the Yellowbrick project! Helping is simple: simply create a notebook that applies the concepts in this Getting Started guide to a small-to-medium size dataset of your choice. Run through the examples with the dataset, and try to change options and customize as much as possible. After you’ve exercised the code with your examples, respond to our alpha testing survey!

4.4.1 Step One: Questionnaire

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

4.4.2 Step Two: Dataset

Select a multivariate dataset of your own. The greater the variety of datasets that we can run through Yellowbrick, the more likely we’ll discover edge cases and exceptions! Please note that your dataset must be well-suited to modeling with scikit-learn. In particular, we recommend choosing a dataset whose target is suited to one of the following supervised learning tasks:

- Regression (target is a continuous variable)
- Classification (target is a discrete variable)

There are datasets that are well suited to both types of analysis; either way, you can use the testing methodology from this notebook for either type of task (or both). In order to find a dataset, we recommend you try the following places:

- UCI Machine Learning Repository
- MLData.org
- Awesome Public Datasets

You’re more than welcome to choose a dataset of your own, but we do ask that you make at least the notebook containing your testing results publicly available for us to review. If the data is also public (or you’re willing to share it with the primary contributors) that will help us figure out bugs and required features much more easily!

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 should be added to the library. We’ll also be doing a user study that we would love for you to participate in. Stay tuned for more great things from Yellowbrick!

4.5 Contributing

Yellowbrick is an open source project that is supported by a community who will gratefully and humbly accept any contributions you might make to the project. Large or small, any contribution makes a big difference; and if you’ve never contributed to an open source project before, we hope you will start with Yellowbrick!

Principally, Yellowbrick development is about the addition and creation of visualizers — objects that learn from data and create a visual representation of the data or model. Visualizers integrate with scikit-learn estimators, transformers, and pipelines for specific purposes and as a result, can be simple to build and deploy. The most common contribution is a new visualizer for a specific model or model family. We’ll discuss in detail how to build visualizers later.

Beyond creating visualizers, there are many ways to contribute:
• Submit a bug report or feature request on GitHub issues.
• Contribute an Jupyter notebook to our examples gallery.
• Assist us with user testing.
• Add to the documentation or help with our website, scikit-yb.org
• Write unit or integration tests for our project.
• Answer questions on our GitHub issues, mailing list, Stack Overflow, and Twitter.
• Translate our documentation into another language.
• Write a blog post, tweet, or share our project with others.
• Teach someone how to use Yellowbrick.

As you can see, there are lots of ways to get involved and we would be very happy for you to join us! The only thing we ask is that you abide by the principles of openness, respect, and consideration of others as described in our Code of Conduct.

**Note:** If you’re unsure where to start, perhaps the best place is to drop the maintainers a note via our mailing list: http://bit.ly/yb-listserv.

### 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.

**Note:** 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:
   
   ```bash
   $ 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 essence, you are requesting that the maintainers merge code from your forked repository into the develop branch of the primary Yellowbrick repository. Once completed, your code will be part of Yellowbrick!

When starting a Yellowbrick contribution, *open the pull request as soon as possible*. We use your PR issue page to discuss your intentions and to give guidance and direction. Every time you push a commit into your forked repository, the commit is automatically included with your pull request, therefore we can review as you code. The earlier you open a PR, the more easily we can incorporate your updates, we’d hate for you to do a ton of work only to discover someone else already did it or that you went in the wrong direction and need to refactor.

**Note:** For a great example of a pull request for a new feature visualizer, check out this one by Carlo Morales.

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 Effective Matplotlib. Additional resources include Nicolas P. Rougier’s Matplotlib tutorial and Chris Moffitt’s Effectively Using Matplotlib.

Visualizer API

There are two basic types of Visualizers:

- **Feature Visualizers** are high dimensional data visualizations that are essentially transformers.
- **Score Visualizers** wrap a scikit-learn regressor, classifier, or clusterer and visualize the behavior or performance of the model on test data.

These two basic types of visualizers map well to the two basic 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:

```python
import matplotlib.pyplot as plt
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.plt(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:

```python
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:

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

class MyVisualizerTests(VisualTestCase, DatasetMixin):
    def test_my_visualizer(self):
        
        # Load the data from the fixture
        dataset = self.load_data('occupancy')
        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:

```bash
$ 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:

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

(continues on next page)
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:

```bash
$ 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:

```bash
$ 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:

```python
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).

    kwags : dict
        Keyword arguments that are passed to the base class and may influence the visualization as defined in other Visualizers.
```

(continues on next page)
Examples
--------

```python
>>> 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:

```rst
.. -*- 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
```

(continues on next page)
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.

**Note:** 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:

```bash
$ 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.

**Note:** 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 Effective Matplotlib

Yellowbrick generates visualizations by wrapping matplotlib, the most prominent Python scientific visualization library. Because of this, Yellowbrick is able to generate publication-ready images for a variety of GUI backends, image formats, and Jupyter notebooks. Yellowbrick strives to provide well-styled visual diagnostic tools and complete information. However, to customize figures or roll your own visualizers, a strong background in using matplotlib is required.

With permission, we have included part of Chris Moffitt’s Effectively Using Matplotlib as a crash course into Matplotlib terminology and usage. For a complete example, please visit his excellent post on creating a visual sales analysis! 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 matplotliblib as follows:

```python
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:

```python
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:

```python
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:

```python
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`):

```python
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:
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:

```python
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)
```

```python
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:

```python
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)`:

```python
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:

```python
fig, ax = plt.subplots()
top_10.plot(kind='barh', y="Sales", x="Name", ax=ax)
ax.set_xlim([-10000, 140000])
ax.legend().set_visible(False)
```

```python
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:

```python
fig, ax = plt.subplots()
top_10.plot(kind='barh', y="Sales", x="Name", ax=ax)
ax.set_xlim([-10000, 140000])
```
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.

```python
# 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()`:

```python
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`.

```python
# 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='')
ax0.axvline(x=avg, color='b', label='Average', linestyle='--', linewidth=1)

# 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)
```

(continues on next page)
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:

```python
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:

```python
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 Yellowbrick for Teachers

For teachers and students of machine learning, Yellowbrick can be used as a framework for teaching and understanding a large variety of algorithms and methods. In fact, Yellowbrick grew out of teaching data science courses at Georgetown’s School of Continuing Studies!

The following slide deck presents an approach to teaching students about the machine learning workflow (the model selection triple), including:

- feature analysis
- feature importances
- feature engineering
- algorithm selection
- model evaluation for classification and regression
- cross-validation
- hyperparameter tuning
Teachers are welcome to download the slides via SlideShare as a PowerPoint deck, and to add them to their course materials to assist in teaching these important concepts.

4.8 Gallery

4.8.1 Feature Analysis

4.8.2 Regression Visualizers

4.8.3 Classification Visualizers

4.8.4 Clustering Visualizers
4.8.5 Model Selection Visualizers

4.8.6 Text Modeling Visualizers

4.8.7 Decision Boundaries Visualizer

4.8.8 Target Visualizers
4.9 About

Yellowbrick is an open source, pure Python project that extends the scikit-learn API with visual analysis and diagnostic tools. The Yellowbrick API also wraps matplotlib to create publication-ready figures and interactive data explorations while still allowing developers fine-grain control of figures. For users, Yellowbrick can help evaluate the performance, stability, and predictive value of machine learning models and assist in diagnosing problems throughout the machine learning workflow.

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

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

4.9.1 Model Selection

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

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

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

The model selection triple was first described in a 2015 SIGMOD paper by Kumar et al. In their paper, which concerns the development of next-generation database systems built to anticipate predictive modeling, the authors cogently express that such systems are badly needed due to the highly experimental nature of machine learning in practice. “Model selection,” they explain, “is iterative and exploratory because the space of [model selection triples] is usually infinite, and it is generally impossible for analysts to know a priori which [combination] will yield satisfactory accuracy and/or insights.”

4.9.2 Who is Yellowbrick for?

Yellowbrick Visualizers have multiple use cases:

- For data scientists, they can help evaluate the stability and predictive value of machine learning models and improve the speed of the experimental workflow.
- For data engineers, Yellowbrick provides visual tools for monitoring model performance in real world applications.
- For users of models, Yellowbrick provides visual interpretation of the behavior of the model in high dimensional feature space.
- For teachers and students, Yellowbrick is a framework for teaching and understanding a large variety of algorithms and methods.

4.9.3 Name Origin

The Yellowbrick package gets its name from the fictional element in the 1900 children’s novel The Wonderful Wizard of Oz by American author L. Frank Baum. In the book, the yellow brick road is the path that the protagonist, Dorothy Gale, must travel in order to reach her destination in the Emerald City.

From Wikipedia: “The road is first introduced in the third chapter of The Wonderful Wizard of Oz. The road begins in the heart of the eastern quadrant called Munchkin Country in the Land of Oz. It functions as a guideline that leads all who follow it, to the road’s ultimate destination—the imperial capital of Oz called Emerald City that is located in the exact center of the entire continent. In the book, the novel’s main protagonist, Dorothy, is forced to search for the road before she can begin her quest to seek the Wizard. This is because the cyclone from Kansas did not release her farmhouse closely near it as it did in the various film adaptations. After the council with the native Munchkins and their dear friend the Good Witch of the North, Dorothy begins looking for it and sees many pathways and roads nearby, (all of which lead in various directions). Thankfully it doesn’t take her too long to spot the one paved with bright yellow bricks.”

4.9.4 Team

Yellowbrick is developed by data scientists who believe in open source and the project enjoys contributions from Python developers all over the world. The project was started by @rebeccabilbro and @bbengfort as an attempt to
better explain machine learning concepts to their students; they quickly realized, however, that the potential for visual steering could have a large impact on practical data science and developed it into a high-level Python library.

Yellowbrick is incubated by District Data Labs, an organization that is dedicated to collaboration and open source development. As part of District Data Labs, Yellowbrick was first introduced to the Python Community at PyCon 2016 in both talks and during the development sprints. The project was then carried on through DDL Research Labs (semester-long sprints where members of the DDL community contribute to various data-related projects).

For a full list of current maintainers and core contributors, please see MAINTAINERS.md in the root of our GitHub repository. Thank you so much to everyone who has contributed to Yellowbrick!

4.9.5 License

Yellowbrick is an open source project and its license is an implementation of the FOSS Apache 2.0 license by the Apache Software Foundation. In plain English this means that you can use Yellowbrick for commercial purposes, modify and distribute the source code, and even sublicense it. We want you to use Yellowbrick, profit from it, and contribute back if you do cool things with it.

There are, however, a couple of requirements that we ask from you. First, when you copy or distribute Yellowbrick source code, please include our copyright and license found in the LICENSE.txt at the root of our software repository. In addition, if we create a file called “NOTICE” in our project you must also include that in your source distribution. The “NOTICE” file will include attribution and thanks to those who have worked so hard on the project! Finally you can’t hold District Data Labs or any Yellowbrick contributor liable for your use of our software, nor use any of our names, trademarks, or logos.

We think that’s a pretty fair deal, and we’re big believers in open source. If you make any changes to our software, use it commercially or academically, or have any other interest, we’d love to hear about it.

4.9.6 Presentations

Yellowbrick has enjoyed the spotlight in several presentations at recent conferences. We hope that these notebooks, talks, and slides will help you understand Yellowbrick a bit better.

Conference Presentations (videos):


Jupyter Notebooks:

- Data Science Delivered: ML Regression Predications

Slides:

- Machine Learning Libraries You’d Wish You’d Known About (PyData Budapest 2017)
- Visualizing the Model Selection Process
- Visualizing Model Selection with Scikit-Yellowbrick
- Visual Pipelines for Text Analysis (Data Intelligence 2017)

4.9.7 Citing Yellowbrick

We hope that Yellowbrick facilitates machine learning of all kinds and we’re particularly fond of academic work and research. If you’re writing a scientific publication that uses Yellowbrick you can cite Bengfort et al. (2018) with the following BibTex:
You can also find DOI (digital object identifiers) for every version of Yellowbrick on zenodo.org; use the BibTeX on this site to reference specific versions or changes made to the software.

We’re also currently working on a scientific paper that describes Yellowbrick in the context of steering the model selection process. Stay tuned for a pre-release of this paper on arXiv.

### 4.9.8 Contacting Us

The best way to contact the Yellowbrick team is to send us a note on one of the following platforms:

- Send an email via our mailing list.
- Direct message us on Twitter.
- Ask a question on Stack Overflow.
- Report an issue on our GitHub Repo.

### 4.10 Frequently Asked Questions

Welcome to our frequently asked questions page. We’re glad that you’re using Yellowbrick! If your question is not captured here, please submit it to our Google Groups Listserv. This is an email list/forum that you, as a Yellowbrick user, can join and interact with other users to address and troubleshoot Yellowbrick issues. The Google Groups Listserv is where you should be able to receive the quickest response. We would welcome and encourage you to join the group so that you can respond to others’ questions! You can also ask questions on Stack Overflow and tag them with “yellowbrick”. Finally, you can add issues on GitHub and you can tweet or direct message us on Twitter @scikit_yb.

#### 4.10.1 How can I change the size of a Yellowbrick plot?

You can change the size of a plot by passing in the desired dimensions in pixels on instantiation of the visualizer:
# Import the visualizer
from yellowbrick.features import RadViz

# Instantiate the visualizer using the 'size' param
visualizer = RadViz(
    classes=classes, features=features, size=(1080, 720)
)

...

Note: we are considering adding support for passing in size in inches in a future Yellowbrick release. For a convenient inch-to-pixel converter, check out www.unitconversion.org.

## 4.10.2 How can I change the title of a Yellowbrick plot?

You can change the title of a plot by passing in the desired title as a string on instantiation:

```python
from yellowbrick.classifier import ROCAUC
from sklearn.linear_model import RidgeClassifier

# Create your custom title
my_title = "ROCAUC Curves for Multiclass RidgeClassifier"

# Instantiate the visualizer passing the custom title
visualizer = ROCAUC(
    RidgeClassifier(), classes=classes, title=my_title
)

...
```

## 4.10.3 How can I change the color of a Yellowbrick plot?

To customize coloring in your plot, use the colors or cmap (or colormap) arguments. Note that different visualizers may require slightly different arguments depending on how they construct the plots.

For instance, the Manifold Visualization accepts a colors argument, for which discrete targets should be the name of one of the Colors and Style or a list of matplotlib colors represented as strings: For instance, the Manifold Visualization accepts a colors argument, for which discrete targets should be the name of a palette from the Yellowbrick Colors and Style or a list of matplotlib colors represented as strings:

```python
from yellowbrick.features.manifold import Manifold

visualizer = Manifold(
    manifold="tsne", target="discrete", colors=["teal", "orchid"]
)

...
```

... whereas for continuous targets, colors should be a colormap:

```python
from yellowbrick.features.manifold import Manifold

visualizer = Manifold(
    manifold="isomap", target="continuous", colors="YlOrRd"
)
```

(continues on next page)
Other visualizers accept a `cmap` argument:

```python
from sklearn.linear_model import LogisticRegression
from yellowbrick.classifier import ConfusionMatrix

visualizer = ConfusionMatrix(
    LogisticRegression(), cmap="YlGnBu"
)
```

Or a `colormap` argument:

```python
from yellowbrick.features import ParallelCoordinates

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

The `Residuals Plot` accepts color argument for the training and test points, `train_color` and `test_color`, respectively:

```python
from yellowbrick.regressor import ResidualsPlot
from sklearn.linear_model import ElasticNet

visualizer = ResidualsPlot(
    model=ElasticNet(),
    train_color=train_color,  # color of points model was trained on
    test_color=test_color,    # color of points model was tested on
    line_color=line_color    # color of zero-error line
)
```

### 4.10.4 How can I save a Yellowbrick plot?

Save your Yellowbrick plot by passing an `outpath` into `poof()`:

```python
from sklearn.cluster import MiniBatchKMeans
from yellowbrick.cluster import KElbowVisualizer

visualizer = KElbowVisualizer(MiniBatchKMeans(), k=(4,12))

visualizer.fit(X)
visualizer.poof(outpath="kelbow_minibatchkmeans.png")
```

Most backends support png, pdf, ps, eps and svg to save your work!
4.10.5 How can I make overlapping points show up better?

You can use the `alpha` param to change the opacity of plotted points (where `alpha=1` is complete opacity, and `alpha=0` is complete transparency):

```python
from yellowbrick.contrib.scatter import ScatterVisualizer

visualizer = ScatterVisualizer(
    x="light", y="CO2", classes=classes, alpha=0.5
)
```

4.10.6 How can I access the sample datasets used in the examples?

Visit the *Example Datasets* page.

4.11 Code of Conduct

The Yellowbrick project is an open source, Python affiliated project. As a result, all interactions that occur with Yellowbrick must meet the guidelines described by the Python Software Foundation Code of Conduct. This includes interactions on all websites, tools, and resources used by Yellowbrick members including (but not limited to) mailing lists, issue trackers, GitHub, StackOverflow, etc.

In general this means everyone is expected to be open, considerate, and respectful of others no matter what their position is within the project.

Beyond this code of conduct, Yellowbrick is striving to set a very particular tone for contributors to the project. We show gratitude for any contribution, no matter how small. We don’t only point out constructive criticism, we always identify positive feedback. When we communicate via text, we write as though we are speaking to each other and our mothers are in the room with us. Our goal is to make Yellowbrick the best possible place to do your first open source contribution, no matter who you are.

4.12 Changelog

4.12.1 Version 0.9

- Tag: v0.9
- Deployed: Wednesday, November 14, 2018
- Contributors: Rebecca Bilbro, Benjamin Bengfort, Zijie (ZJ) Poh, Kristen McIntyre, Nathan Danielsen, David Waterman, Larry Gray, Prema Roman, Juan Kehoe, Alyssa Batula, Peter Espinosa, Joanne Lin, @rlshuhart, @archaeocharlie, @dschoenleber, Tim Black, @iguk1987, Mohammed Fadhil, Jonathan Lakanlale, Andrew Godbehere, Sivasurya Santhanam, Gopal Krishna

**Major Changes:**

- Target module added for visualizing dependent variable in supervised models.
- Prototype missing values visualizer in contrib module.
- `BalancedBinningReference` visualizer for thresholding unbalanced data (undocumented).
- `CVScores` visualizer to instrument cross-validation.
FeatureCorrelation visualizer to compare relationship between a single independent variable and the target.

ICDM visualizer, intercluster distance mapping using projections similar to those used in pyLDAVis.

PrecisionRecallCurve visualizer showing the relationship of precision and recall in a threshold-based classifier.

Enhanced FeatureImportance for multi-target and multi-coefficient models (e.g. probabilistic models) and allows stacked bar chart.

Adds option to plot PDF to ResidualsPlot histogram.

Adds document boundaries option to DispersionPlot and uses colored markers to depict class.

Added alpha parameter for opacity to the scatter plot visualizer.

Modify KElbowVisualizer to accept a list of k values.

ROCAUC bugfix to allow binary classifiers that only have a decision function.

TSNE bugfix so that title and size params are respected.

ConfusionMatrix bugfix to correct percentage displays adding to 100.

ResidualsPlot bugfix to ensure specified colors are both in histogram and scatterplot.

Fixed unicode decode error on Py2 compatible Windows using Hobbies corpus.

Require matplotlib 1.5.1 or matplotlib 2.0 (matplotlib 3.0 not supported yet).

Deprecated percent and sample_weight arguments to ConfusionMatrix fit method.

Yellowbrick now depends on SciPy 1.0 and scikit-learn 0.20.

Minor Changes:

Removed hardcoding of SilhouetteVisualizer axes dimensions.

Audit classifiers to ensure they conform to score API.

Fix for Manifold fit_transform bug.

Fixed Manifold import bug.

Started reworking datasets API for easier loading of examples.

Added Timer utility for keeping track of fit times.

Added slides to documentation for teachers teaching ML/Yellowbrick.

Added an FAQ to the documentation.

Manual legend drawing utility.

New examples notebooks for regression and clustering.

Example of interactive classification visualization using ipywidgets.

Example of using Yellowbrick with PyTorch.

Repairs to ROCAUC tests and binary/multiclass ROCAUC construction.

Rename tests/random.py to tests/rand.py to prevent NumPy errors.

Improves ROCAUC, KElbowVisualizer, and SilhouetteVisualizer documentation.

Fixed visual display bug in JointPlotVisualizer.

Fixed image in JointPlotVisualizer documentation.
• Clear figure option to poof.
• Fix color plotting error in residuals plot quick method.
• Fixed bugs in KElbowVisualizer, FeatureImportance, Index, and Datasets documentation.
• Use LGTM for code quality analysis (replacing Landscape).
• Updated contributing docs for better PR workflow.
• Submitted JOSS paper.

4.12.2 Version 0.8

• Tag: v0.8
• Deployed: Thursday, July 12, 2018
• Contributors: Rebecca Bilbro, Benjamin Bengfort, Nathan Danielsen, Larry Gray, Prema Roman, Adam Morris, Kristen McIntyre, Raul Peralta, Sayali Sonawane, Alyssa Riley, Petr Mitev, Chris Stehlík, @thekylesaurus, Luis Carlos Mejía García, Raul Samayoa, Carlo Mazzaferro

Major Changes:
• Added Support to ClassificationReport - @ariley1472
• We have an updated Image Gallery - @ralle123
• Improved performance of ParallelCoordinates Visualizer @thekylesaurus
• Added Alpha Transparency to RadViz Visualizer @lumega
• CVScores Visualizer - @pdamodaran
• Added fast and alpha parameters to ParallelCoordinates visualizer @bbengfort
• Make support an optional parameter for ClassificationReport @lwgray
• Bug Fix for Usage of multidimensional arrays in FeatureImportance visualizer @rebeccabilbro
• Deprecate ScatterVisualizer to contrib @bbengfort
• Implements histogram alongside ResidualsPlot @bbengfort
• Adds biplot to the PCADecomposition visualizer @RaulPL
• Adds Datasaurus Dataset to show importance of visualizing data @lwgray
• Add DispersionPlot Plot @lwgray

Minor Changes:
• Fix grammar in tutorial.rst - @chrisfs
• Added Note to tutorial indicating subtle differences when working in Jupyter notebook - @chrisfs
• Update Issue template @bbengfort
• Added Test to check for NLTK postag data availability - @Sayali
• Clarify quick start documentation @mitevpi
• Deprecated DecisionBoundary
• Threshold Visualization aliases deprecated

4.12. Changelog
4.12.3 Version 0.7

- Tag: v0.7
- Deployed: Thursday, May 17, 2018
- Contributors: Benjamin Bengfort, Nathan Danielsen, Rebecca Bilbro, Larry Gray, Ian Ozsvald, Jeremy Tuloup, Abhishek Bharani, Raúl Peralta Lozada, Tabishsada, Kristen McIntyre, Neal Humphrey

Changes:

- **New Feature!** Manifold visualizers implement high-dimensional visualization for non-linear structural feature analysis.
- **New Feature!** There is now a model_selection module with LearningCurve and ValidationCurve visualizers.
- **New Feature!** The RFECV (recursive feature elimination) visualizer with cross-validation visualizes how removing the least performing features improves the overall model.
- **New Feature!** The VisualizerGrid is an implementation of the MultipleVisualizer that creates axes for each visualizer using plt.subplots, laying the visualizers out as a grid.
- **New Feature!** Added yellowbrick.datasets to load example datasets.
- New Experimental Feature! An experimental StatsModelsWrapper was added to yellowbrick.contrib.statsmodels that will allow user to use StatsModels estimators with visualizers.
- **Enhancement!** ClassificationReport documentation to include more details about how to interpret each of the metrics and compare the reports against each other.
- **Enhancement!** Modifies scoring mechanism for regressor visualizers to include the R2 value in the plot itself with the legend.
- **Enhancement!** Updated and renamed the ThreshViz to be defined as DiscriminationThreshold, implements a few more discrimination features such as F1 score, maximizing arguments and annotations.
- **Enhancement!** Update clustering visualizers and corresponding distortion_score to handle sparse matrices.
- Added code of conduct to meet the GitHub community guidelines as part of our contributing documentation.
- Added is_probabilistic type checker and converted the type checking tests to pytest.
- Added a contrib module and DecisionBoundaries visualizer has been moved to it until further work is completed.
- Numerous fixes and improvements to documentation and tests. Add academic citation example and Zenodo DOI to the Readme.

Bug Fixes:

- Adds RandomVisualizer for testing and add it to the VisualizerGrid test cases.
- Fix / update tests in tests.test_classifier.test_class_prediction_error.py to remove hardcoded data.

Deprecation Warnings:

- ScatterPlotVisualizer is being moved to contrib in 0.8
- DecisionBoundaryVisualizer is being moved to contrib in 0.8
- ThreshViz is renamed to DiscriminationThreshold.
NOTE: These deprecation warnings originally mentioned deprecation in 0.7, but their life was extended by an additional version.

4.12.4 Version 0.6

- Tag: v0.6
- Deployed: Saturday, March 17, 2018
- Contributors: Benjamin Bengfort, Nathan Danielsen, Rebecca Bilbro, Larry Gray, Kristen McIntyre, George Richardson, Taylor Miller, Gary Mayfield, Phillip Schafer, Jason Keung

Changes:

- **New Feature!** The `FeatureImportances` Visualizer enables the user to visualize the most informative (relative and absolute) features in their model, plotting a bar graph of `feature_importances_` or `coef_` attributes.
- **New Feature!** The `ExplainedVariance` Visualizer produces a plot of the explained variance resulting from a dimensionality reduction to help identify the best tradeoff between number of dimensions and amount of information retained from the data.
- **New Feature!** The `GridSearchVisualizer` creates a color plot showing the best grid search scores across two parameters.
- **New Feature!** The `ClassPredictionError` Visualizer is a heatmap implementation of the class balance visualizer, which provides a way to quickly understand how successfully your classifier is predicting the correct classes.
- **New Feature!** The `ThresholdVisualizer` allows the user to visualize the bounds of precision, recall and queue rate at different thresholds for binary targets after a given number of trials.
- New `MultiFeatureVisualizer` helper class to provide base functionality for getting the names of features for use in plot annotation.
- Adds font size param to the confusion matrix to adjust its visibility.
- Add quick method for the confusion matrix
- Tests: In this version, we’ve switched from using nose to pytest. Image comparison tests have been added and the visual tests are updated to matplotlib 2.2.0. Test coverage has also been improved for a number of visualizers, including `JointPlot`, `AlphaPlot`, `FreqDist`, `RadViz`, `ElbowPlot`, `SilhouettePlot`, `ConfusionMatrix`, `Rank1D`, and `Rank2D`.
- Documentation updates, including discussion of Image Comparison Tests for contributors.

Bug Fixes:

- Fixes the `resolve_colors` function. You can now pass in a number of colors and a colormap and get back the correct number of colors.
- Fixes `TSNEVisualizer` Value Error when no classes are specified.
- Adds the circle back to `RadViz`! This visualizer has also been updated to ensure there’s a visualization even when there are missing values.
- Updated `RocAuc` to correctly check the number of classes.
- Switch from converting structured arrays to ndarrays using `np.copy` instead of `np.tolist` to avoid NumPy deprecation warning.
- `DataVisualizer` updated to remove `np.nan` values and warn the user that nans are not plotted.
- `ClassificationReport` no longer has lines that run through the numbers, is more grid-like
Deprecation Warnings:

- ScatterPlotVisualizer is being moved to contrib in 0.7
- DecisionBoundaryVisualizer is being moved to contrib in 0.7

4.12.5 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 PCA Decomposition Visualizer, which decomposes high-dimensional data into two or three dimensions so that each instance can be plotted in a scatter plot.
- New and improved ROCAUC Visualizer, which now supports multiclass classification.
- Prototype DecisionBoundary Visualizer, which is a bivariate data visualization algorithm that plots the decision boundaries of each class.
- Added Rank1D Visualizer, which is a one-dimensional ranking of features that utilizes the Shapiro-Wilks ranking by taking into account only a single feature at a time (e.g. histogram analysis).
- Improved PredictionErrorPlot with identity line, shared limits, and R-squared.
- Updated FreqDist Visualizer to make word features a hyperparameter.
- Added normalization and scaling to ParallelCoordinates.
- Added Learning Curve Visualizer, which displays a learning curve based on the number of samples versus the training and cross validation scores to show how a model learns and improves with experience.
- Added data downloader module to the Yellowbrick library.
- Complete overhaul of the Yellowbrick documentation; categories of methods are located in separate pages to make it easier to read and contribute to the documentation.
- Added a new color palette inspired by ANN-generated colors

Bug Fixes:

- Repairs to PCA, RadViz, FreqDist unit tests
- Repair to matplotlib version check in JointPlotVisualizer

4.12.6 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
4.12.7 Version 0.4.1

This release is an intermediate version bump in anticipation of the PyCon 2017 sprints.

The primary goals of this version were to (1) update the Yellowbrick dependencies (2) enhance the Yellowbrick documentation to help orient new users and contributors, and (3) make several small additions and upgrades (e.g. pulling the Yellowbrick utils into a standalone module).

We have updated the scikit-learn and SciPy dependencies from version 0.17.1 or later to 0.18 or later. This primarily entails moving from `from sklearn.cross_validation import train_test_split` to `from sklearn.model_selection import train_test_split`.

The updates to the documentation include new Quickstart and Installation guides, as well as updates to the Contributors documentation, which is modeled on the scikit-learn contributing documentation.

This version also included upgrades to the KMeans visualizer, which now supports not only silhouette_score but also distortion_score and calinski_harabaz_score. The distortion_score computes the mean distortion of all samples as the sum of the squared distances between each observation and its closest centroid. This is the metric that KMeans attempts to minimize as it is fitting the model. The calinski_harabaz_score is defined as ratio between the within-cluster dispersion and the between-cluster dispersion.

Finally, this release includes a prototype of the VisualPipeline, which extends scikit-learn's Pipeline class, allowing multiple Visualizers to be chained or sequenced together.

- Tag: v0.4.1
- Deployed: Monday, May 22, 2017
- Contributors: Benjamin Bengfort, Rebecca Bilbro, Nathan Danielsen

Changes:

- Score and model visualizers now wrap estimators as proxies so that all methods on the estimator can be directly accessed from the visualizer
- Updated scikit-learn dependency from >=0.17.1 to >=0.18
- Replaced `sklearn.cross_validation` with `model_selection`
- Updated SciPy dependency from >=0.17.1 to >=0.18
- ScoreVisualizer now subclasses ModelVisualizer; towards allowing both fitted and unfitted models passed to Visualizers
- Added CI tests for Python 3.6 compatibility
- Added new quickstart guide and install instructions
- Updates to the contributors documentation
- Added distortion_score and calinski_harabaz_score computations and visualizations to KMeans visualizer.
- Replaced the `self.ax` property on all of the individual draw methods with a new property on the Visualizer class that ensures all visualizers automatically have axes.
- Refactored the utils module into a package
- Continuing to update the docstrings to conform to Sphinx
- Added a prototype visual pipeline class that extends the scikit-learn pipeline class to ensure that visualizers get called correctly.

Bug Fixes:

- Fixed title bug in Rank2D FeatureVisualizer
4.12.8 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

4.12.9 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.

4.12.10 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

4.12.11 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.

4.12.12 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 ($y - \hat{y}$) 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

4.12.13 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, autocdoc, 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

4.12.14 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
CHAPTER 5

Indices and tables

- genindex
- modindex
yellowbrick.anscombe, 34
yellowbrick.classifier.classification_report, 115
yellowbrick.classifier.confusion_matrix, 103
yellowbrick.classifier.prcurve, 112
yellowbrick.classifier.rocauc, 107
yellowbrick.classifier.threshold, 118
yellowbrick.cluster.elbow, 122
yellowbrick.cluster.icdm, 127
yellowbrick.cluster.silhouette, 125
yellowbrick.contrib.classifier.boundaries, 163
yellowbrick.contrib.missing.bar, 170
yellowbrick.contrib.missing.dispersion, 173
yellowbrick.contrib.scatter, 166
yellowbrick.contrib.statsmodels.base, 165
yellowbrick.features.importances, 65
yellowbrick.features.jointplot, 73
yellowbrick.features.manifold, 59
yellowbrick.features.pca, 53
yellowbrick.features.pcoords, 47
yellowbrick.features.radviz, 37
yellowbrick.features.rankd, 41
yellowbrick.features.rfecv, 69
yellowbrick.model_selection.cross_validation, 146
yellowbrick.model_selection.learning_curve, 140
yellowbrick.model_selection.validation_curve, 134
yellowbrick.regressor.alphas, 94
yellowbrick.regressor.residuals, 88
yellowbrick.style.colors, 208
yellowbrick.style.palettes, 208
yellowbrick.style.rcmod, 209
yellowbrick.target.binning, 77
yellowbrick.target.class_balance, 80
yellowbrick.target.feature_correlation, 84
yellowbrick.text.dispersion, 161
yellowbrick.text.freqdist, 154
yellowbrick.text.tsne, 158
Index

A
ALGORITHMS
   brick.features.manifold.Manifold attribute), 61
AlphaSelection (class in yellowbrick.regressor.alphas), 94
anscombe() (in module yellowbrick.anscombe), 34
B
BalancedBinningReference (class in yellowbrick.target.binning), 77
C
ClassBalance (class in yellowbrick.target.class_balance), 80
ClassificationReport (class in yellowbrick.classifier.classification_report), 100
ClassPredictionError (class in yellowbrick.classifier.class_prediction_error), 115
cluster_centers_ (yellowbrick.cluster.icdm.InterclusterDistance attribute), 129
color_palette() (in module yellowbrick.style.palettes), 208
ColorMap (class in yellowbrick.style.colors), 208
colors (yellowbrick.style.colors.ColorMap attribute), 208
ConfusionMatrix (class in yellowbrick.classifier.confusion_matrix), 103
count() (yellowbrick.text.freqdist.FrequencyVisualizer method), 155
CVScores (class in yellowbrick.model_selection.cross_validation), 146
D
DecisionBoundariesVisualizer (class in yellowbrick.contrib.classifier.boundaries), 163
DiscriminationThreshold (class in yellowbrick.classifier.threshold), 118
DispersionPlot (class in yellowbrick.text.dispersion), 161
draw() (yellowbrick.classifier.class_prediction_error.ClassPredictionError method), 116
draw() (yellowbrick.classifier.classification_report.ClassificationReport method), 100
draw() (yellowbrick.classifier.confusion_matrix.ConfusionMatrix method), 104
draw() (yellowbrick.classifier.prcurve.PrecisionRecallCurve method), 114
draw() (yellowbrick.classifier.rocauc.ROCAUC method), 109
draw() (yellowbrick.classifier.threshold.DiscriminationThreshold method), 120
draw() (yellowbrick.cluster.elbow.KElbowVisualizer method), 124
draw() (yellowbrick.cluster.icdm.InterclusterDistance method), 129
draw() (yellowbrick.cluster.silhouette.SilhouetteVisualizer method), 126
draw() (yellowbrick.contrib.classifier.boundaries.DecisionBoundariesVisualizer method), 164
draw() (yellowbrick.contrib.missing.bar.MissingValuesBar method), 171
draw() (yellowbrick.contrib.missing.dispersion.MissingValuesDispersion method), 174
draw() (yellowbrick.contrib.scatter.ScatterVisualizer method), 167
draw() (yellowbrick.features.importances.FeatureImportances method), 66
draw() (yellowbrick.features.jointplot.JointPlotVisualizer method), 75
draw() (yellowbrick.features.manifold.Manifold method), 61
draw() (yellowbrick.features.pca.PCADecomposition method), 54
draw() (yellowbrick.features.pcoords.ParallelCoordinates method), 48
draw() (yellowbrick.features.radviz.RadialVisualizer method), 37
draw() (yellowbrick.features.rankd.Rank1D method), 42
draw() (yellowbrick.features.rankd.Rank2D method), 43
draw() (yellowbrick.features.rfecv.RFECV method), 70
finaliz() (yellowbrick.classifier.confusion_matrix.ConfusionMatrix
method), 104
finaliz() (yellowbrick.classifier.pcurve.PrecisionRecallCurve
method), 114
finaliz() (yellowbrick.classifier.rocauc.ROCAUC
method), 109

draw() (yellowbrick.model_selection.cross_validation.CVScores
method), 147
finaliz() (yellowbrick.classifier.threshold.DiscriminationThreshold
method), 120
finaliz() (yellowbrick.cluster.elbow.KElbowVisualizer
method), 124
finaliz() (yellowbrick.cluster.icdm.InterclusterDistance
method), 129
finaliz() (yellowbrick.cluster.silhouette.SilhouetteVisualizer
method), 126

draw() (yellowbrick.model_selection.learning_curve.LearningCurve
method), 143
finaliz() (yellowbrick.contrib.classifier.boundaries.DecisionBoundariesVisualizer
method), 164
finaliz() (yellowbrick.contrib.missing.bar.MissingValuesBar
method), 171
finaliz() (yellowbrick.contrib.missing.dispersion.MissingValuesDispersion
method), 174
finaliz() (yellowbrick.contrib.scatter.ScatterVisualizer
method), 167
finaliz() (yellowbrick.features.importances.FeatureImportances
method), 66
finaliz() (yellowbrick.features.jointplot.JointPlotVisualizer
method), 75
finaliz() (yellowbrick.features.manifold.Manifold
method), 61
finaliz() (yellowbrick.features.pca.PCADecomposition
method), 54
finaliz() (yellowbrick.features.pcoords.ParallelCoordinates
method), 49
finaliz() (yellowbrick.models.regressor.alphas.AlphaSelection
method), 95
finaliz() (yellowbrick.models.regressor.alphas.ManualAlphaSelection
method), 96
finaliz() (yellowbrick.models.regressor.residuals.PredictionError
method), 93
finaliz() (yellowbrick.models.regressor.residuals.ResidualsPlot
method), 89
finaliz() (yellowbrick.target.binning.BalancedBinningReference
method), 77
finaliz() (yellowbrick.target.class_balance.ClassBalance
method), 81
finaliz() (yellowbrick.target.feature_correlation.FeatureCorrelation
method), 85
finaliz() (yellowbrick.text.dispersion.DispersionPlot
method), 162

draw_classes() (yellowbrick.features.pcoords.ParallelCoordinates
method), 49

draw_instances() (yellowbrick.features.pcoords.ParallelCoordinates
method), 49

draw_joint() (yellowbrick.features.jointplot.JointPlotVisualizer
method), 75

draw_multi_dispersion_chart() (yellowbrick.contrib.missing.dispersion.MissingValuesDispersion
method), 174
finaliz() (yellowbrick.contrib.missing.dispersion.MissingValuesDispersion
method), 174
finaliz() (yellowbrick.features.pcoords.ParallelCoordinates
method), 49
finaliz() (yellowbrick.features.radviz.RadialVisualizer
method), 37
finaliz() (yellowbrick.models.rfecv.RFECV method), 70
finaliz() (yellowbrick.models.selection.cross_validation.CVScores
method), 147
finaliz() (yellowbrick.models.selection.learning_curve.LearningCurve
method), 143
finaliz() (yellowbrick.models.selection.validation_curve.ValidationCurve
method), 135
finaliz() (yellowbrick.models.regressor.alphas.AlphaSelection
method), 95
finaliz() (yellowbrick.models.regressor.residuals.PredictionError
method), 93
finaliz() (yellowbrick.models.regressor.residuals.ResidualsPlot
method), 89
finaliz() (yellowbrick.target.binning.BalancedBinningReference
method), 77
finaliz() (yellowbrick.target.class_balance.ClassBalance
method), 81
finaliz() (yellowbrick.target.feature_correlation.FeatureCorrelation
method), 85
finaliz() (yellowbrick.text.dispersion.DispersionPlot
method), 162

\textbf{F}

FeatureCorrelation (class in yellowbrick.target.feature_correlation), 84
FeatureImportances (class in yellowbrick.features.importances), 65
finalize() (yellowbrick.classifier.class_prediction_error.ClassPredictionError
method), 116
finalize() (yellowbrick.classifier.classification_report.ClassificationReport
method), 100
Index
MissingValuesDispersion (class in yellowbrick.contrib.missing.dispersion), 173

N
normalize() (yellowbrick.features.radviz.RadialVisualizer static method), 38
NORMALIZERS (yellowbrick.features.pcoords.ParallelCoordinates attribute), 48
NULL_CLASS (yellowbrick.text.dispersion_DISPERSION), 162
NULL_CLASS (yellowbrick.text.tsne.TSNEVisualizer attribute), 159

P
ParallelCoordinates (class in yellowbrick.features.pcoords), 47
PCADecomposition (class in yellowbrick.features.pca), 53
poof() (yellowbrick.target.binning.BalancedBinningReference method), 77
PrecisionRecallCurve (class in yellowbrick.classifier.prcurve), 112
predict() (yellowbrick.contrib.statsmodels.base.StatsModelsWrapper method), 165
PredictionError (class in yellowbrick.regressor.residuals), 91

R
RadialVisualizer (class in yellowbrick.features.radviz), 37
RadViz (in module yellowbrick.features.radviz), 38
Rank1D (class in yellowbrick.features.rankd), 41
Rank2D (class in yellowbrick.features.rankd), 42
ranking_methods (yellowbrick.features.rankd.Rank1D attribute), 42
ranking_methods (yellowbrick.features.rankd.Rank2D attribute), 43
reset_defaults() (in module yellowbrick.style.rcmod), 210
reset_orig() (in module yellowbrick.style.rcmod), 210
ResidualsPlot (class in yellowbrick.regressor.residuals), 88
resolve_colors() (in module yellowbrick.style.colors), 208
RFECV (class in yellowbrick.features.rfecv), 69
ROCAUC (class in yellowbrick.classifier.rocauc), 107

S
ScatterVisualizer (class in yellowbrick.contrib.scatter), 166
score() (yellowbrick.classifier.class_prediction_error.ClassPredictionError method), 116
score() (yellowbrick.classifier.classification_report.ClassificationReport method), 101

T
transform() (yellowbrick.features.manifold.Manifold method), 62
transform() (yellowbrick.features.pca.PCADecomposition method), 54
transformer (yellowbrick.cluster.icdm.InterclusterDistance attribute), 129
TSNEVisualizer (class in yellowbrick.text.tsne), 158

V
ValidationCurve (class in yellowbrick.model_selection.validation_curve), 134

Y
yellowbrick.cluster.silhouette (module), 125
yellowbrick.classifier.class_prediction_error (module), 115
yellowbrick.classifier.classification_report (module), 100
yellowbrick.classifier.confusion_matrix (module), 103
yellowbrick.classifier.pcurve (module), 112
yellowbrick.classifier.rocauc (module), 107
yellowbrick.classifier.threshold (module), 118
yellowbrick.cluster.elbow (module), 122
yellowbrick.cluster.icdm (module), 127
yellowbrick.cluster.silhouette (module), 125
yellowbrick.contrib.classifier.boundaries (module), 163
yellowbrick.contrib.missing.bar (module), 170
yellowbrick Documentation, Release 0.9.1

yellowbrick.contrib.missing.dispersion (module), 173
yellowbrick.contrib.scatter (module), 166
yellowbrick.contrib.statsmodels.base (module), 165
yellowbrick.features.importances (module), 65
yellowbrick.features.jointplot (module), 73
yellowbrick.features.manifold (module), 59
yellowbrick.features.pca (module), 53
yellowbrick.features.pcoords (module), 47
yellowbrick.features.radviz (module), 37
yellowbrick.features.rankd (module), 41
yellowbrick.features.rfe (module), 69
yellowbrick.model_selection.cross_validation (module), 146
yellowbrick.model_selection.learning_curve (module), 140
yellowbrick.model_selection.validation_curve (module), 134
yellowbrick.regressor.alphas (module), 94
yellowbrick.regressor.residuals (module), 88, 91
yellowbrick.style.colors (module), 208
yellowbrick.style.palettes (module), 208
yellowbrick.style.rcmod (module), 209
yellowbrick.target.binning (module), 77
yellowbrick.target.class_balance (module), 80
yellowbrick.target.feature_correlation (module), 84
yellowbrick.text.dispersion (module), 161
yellowbrick.text.freqdist (module), 154
yellowbrick.text.tsne (module), 158

Index 257