# Contributing to Yellowbrick

\*\*NOTE: This document is a "getting started" summary for contributing to the Yellowbrick project.\*\* To read the full contributor's guide, please visit the [contributing page](http://www.scikit-yb.org/en/latest/contributing/index.html) in the documentation. Please make sure to read this page carefully to ensure the review process is as smooth as possible and to ensure the greatest likelihood of having your contribution be merged.

For more on the development path, goals, and motivations behind Yellowbrick, check out our developer presentation: [Visualizing Model Selection with Scikit-Yellowbrick: An Introduction to Developing Visualizers](http://www.slideshare.net/BenjaminBengfort/visualizing-model-selection-with-scikityellowbrick-an-introduction-to-developing-visualizers).

## How to Contribute

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\* &mdash; 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 therefore 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](https://github.com/DistrictDataLabs/yellowbrick/issues).

- Contribute a Jupyter notebook to our examples[ gallery](https://github.com/DistrictDataLabs/yellowbrick/tree/develop/examples).

- Assist us with [user testing](http://www.scikit-yb.org/en/latest/evaluation.html).

- Add to the documentation or help with our website, [scikit-yb.org](http://www.scikit-yb.org).

- Write [unit or integration tests](https://www.scikit-yb.org/en/latest/contributing/developing\_visualizers.html#integration-tests) for our project.

- Answer questions on our issues, mailing list, Stack Overflow, and elsewhere.

- Translate our documentation into another language.

- Write a blog post, tweet, or share our project with others.

- [Teach](https://www.scikit-yb.org/en/latest/teaching.html) someone how to use Yellowbrick.

As you can see, there are lots of ways to get involved and we would be very happy for you to join us! The only thing we ask is that you abide by the principles of openness, respect, and consideration of others as described in the [Python Software Foundation Code of Conduct](https://www.python.org/psf/codeofconduct/).

## 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 contributes by [adding an issue](https://github.com/DistrictDataLabs/yellowbrick/issues).

3. \*\*Fork\*\* the repository into your own GitHub account.

4. Create a \*\*Pull Request\*\* first thing to [connect with us](https://github.com/DistrictDataLabs/yellowbrick/pulls) about your task.

5. \*\*Code\*\* the feature, write the documentation, add your contribution.

6. \*\*Review\*\* the code with core contributors who will guide you to a high quality submission.

7. \*\*Merge\*\* your contribution into the Yellowbrick codebase.

We believe that \*contribution is collaboration\* and therefore emphasize \*communication\* throughout the open source process. We rely heavily on GitHub's social coding tools to allow us to do this. For instance, we use GitHub's [milestone](https://help.github.com/en/articles/about-milestones) feature to focus our development efforts for each Yellowbrick semester, so be sure to check out the issues associated with our [current milestone](https://github.com/districtdatalabs/yellowbrick/milestones)!

Once you have a good sense of how you are going to implement the new feature (or fix the bug!), you can reach out for feedback from the maintainers by creating a [pull request](https://github.com/DistrictDataLabs/yellowbrick/pulls). Please note that if we feel your solution has not been thought out in earnest, or if the PR is not aligned with our [current milestone](https://github.com/districtdatalabs/yellowbrick/milestones) goals, we may reach out to ask that you close the PR so that we can prioritize reviewing the most critical feature requests and bug fixes.

Ideally, any pull request should be capable of resolution within 6 weeks of being opened. This timeline helps to keep our pull request queue small and allows Yellowbrick to maintain a robust release schedule to give our users the best experience possible. However, the most important thing is to keep the dialogue going! And if you're unsure whether you can complete your idea within 6 weeks, you should still go ahead and open a PR and we will be happy to help you scope it down as needed.

If we have comments or questions when we evaluate your pull request and receive no response, we will also close the PR after this period of time. Please know that this does not mean we don't value your contribution, just that things go stale. If in the future you want to pick it back up, feel free to address our original feedback and to reference the original PR in a new pull request.

### Forking the Repository

The first step is to fork the repository into your own account. This will create a copy of the codebase that you can edit and write to. Do so by clicking the \*\*"fork"\*\* button in the upper right corner of the Yellowbrick GitHub page.

Once forked, use the following steps to get your development environment set up on your computer:

1. Clone the repository.

After clicking the fork button, you should be redirected to the GitHub page of the repository in your user account. You can then clone a copy of the code to your local machine.

```

$ git clone https://github.com/[YOURUSERNAME]/yellowbrick

$ cd yellowbrick

```

Optionally, you can also [add the upstream remote](https://help.github.com/articles/configuring-a-remote-for-a-fork/) to synchronize with changes made by other contributors:

```

$ git remote add upstream https://github.com/DistrictDataLabs/yellowbrick

```

See "Branching Conventions" below for more on this topic.

2. Create a virtual environment.

Yellowbrick developers typically use [virtualenv](https://virtualenv.pypa.io/en/stable/) (and [virtualenvwrapper](https://virtualenvwrapper.readthedocs.io/en/latest/), [pyenv](https://github.com/pyenv/pyenv-virtualenv) or [conda envs](https://conda.io/docs/using/envs.html) 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!

### Branching Conventions

The Yellowbrick repository is set up in a typical production/release/development cycle as described in "[A Successful Git Branching Model](http://nvie.com/posts/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 maintainers will push to master.

\*\*NOTE:\*\* All pull requests should be into the `yellowbrick/develop` branch from your forked repository.

You should 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](https://help.github.com/articles/configuring-a-remote-for-a-fork/)). You can do that as follows:

```

$ git remote add upstream https://github.com/DistrictDataLabs/yellowbrick.git`

$ git remote -v

origin https://github.com/YOUR\_USERNAME/YOUR\_FORK.git (fetch)

origin https://github.com/YOUR\_USERNAME/YOUR\_FORK.git (push)

upstream https://github.com/DistrictDataLabs/yellowbrick.git (fetch)

upstream https://github.com/DistrictDataLabs/yellowbrick.git (push)

```

When you're ready, request a code review for your pull request. Then, when reviewed and approved, you can merge your fork into our main branch. Make sure to use the "Squash and Merge" option in order to create a Git history that is understandable.

\*\*NOTE to maintainers\*\*: When merging a pull request, use the "squash and merge" option and make sure to edit the both the subject and the body of the commit message so that when we're putting the changelog together, we know what happened in the PR. I recommend reading [Chris Beams' \_How to Write a Git Commit Message\_](https://chris.beams.io/posts/git-commit/) so we're all on the same page!

Core contributors and those who are planning on contributing multiple PRs might want to consider using feature branches to reduce the number of merges (and merge conflicts). Create a feature branch as follows:

```

$ git checkout -b feature-myfeature develop

$ git push --set-upstream origin feature-myfeature

```

Once you are done working (and everything is tested) you can submit a PR from your feature branch. Synchronize with `upstream` once the PR has been merged and delete the feature branch:

```

$ git checkout develop

$ git pull upstream develop

$ git push origin develop

$ git branch -d feature-myfeature

$ git push origin --delete feature-myfeature

```

Head back to Github and checkout another issue!

## 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.

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"](http://scikit-learn.org/stable/developers/contributing.html#apis-of-scikit-learn-objects) and ["rolling your own estimator"](http://scikit-learn.org/stable/developers/contributing.html#rolling-your-own-estimator). In terms of matplotlib, check out [Nicolas P. Rougier's Matplotlib tutorial](https://www.labri.fr/perso/nrougier/teaching/matplotlib/).

### Visualizer API

There are two basic types of Visualizers:

- \*\*Feature Visualizers\*\* are high dimensional data visualizations that are essentially transformers.

- \*\*Score Visualizers\*\* wrap a scikit-learn regressor, classifier, or clusterer and visualize the behavior or performance of the model on test data.

These two basic types of visualizers map well to the two basic estimator objects in scikit-learn:

- \*\*Transformers\*\* take input data and return a new data set.

- \*\*Models\*\* are fit to training data and can make predictions.

The scikit-learn API is object oriented, and estimators are initialized with parameters by instantiating their class. Hyperparameters can also be set using the `set\_attrs()` method and retrieved with the corresponding `get\_attrs()` method. All scikit-learn estimators have a `fit(X, y=None)` method that accepts a two dimensional data array, `X`, and optionally a vector `y` of target values. The `fit()` method trains the estimator, making it ready to transform data or make predictions. Transformers have an associated `transform(X)` method that returns a new dataset, `Xprime` and models have a `predict(X)` method that returns a vector of predictions, `yhat`. Models may also have a `score(X, y)` method that evaluate the performance of the model.

Visualizers interact with scikit-learn objects by intersecting with them at the methods defined above. Specifically, visualizers perform actions related to `fit()`, `transform()`, `predict()`, and `score()` then call a `draw()` method which initializes the underlying figure associated with the visualizer. The user calls the visualizer's `show()` method, which in turn calls a `finalize()` method on the visualizer to draw legends, titles, etc. and then `show()` renders the figure. The Visualizer API is therefore:

- `draw()`: add visual elements to the underlying axes object

- `finalize()`: prepare the figure for rendering, adding final touches such as legends, titles, axis labels, etc.

- `show()`: render the figure for the user.

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 plot

from yellowbrick.base import Visualizer

class MyVisualizer(Visualizer):

def \_\_init\_\_(self, ax=None, \*\*kwargs):

super(MyVisualizer, self).\_\_init\_\_(ax, \*\*kwargs)

def fit(self, X, y=None):

super(MyVisualizer, self).fit(X, y)

self.draw(X)

return self

def draw(self, X):

self.ax.plot(X)

return self.ax

def finalize(self):

self.set\_title("My Visualizer")

```

This simple visualizer simply draws a line graph for some input dataset X, intersecting with the scikit-learn API at the `fit()` method. A user would use this visualizer in the typical style::

```python

visualizer = MyVisualizer()

visualizer.fit(X)

visualizer.show()

```

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 a helper, `VisualTestCase`. Create your unit test as follows::

```python

import pytest

from yellowbrick.datasets import load\_occupancy

from tests.base import VisualTestCase

class MyVisualizerTests(VisualTestCase):

def test\_my\_visualizer(self):

"""

Test MyVisualizer on a real dataset

"""

# Load the data

X,y = load\_occupancy()

try:

visualizer = MyVisualizer()

visualizer.fit(X)

visualizer.show()

except Exception as e:

pytest.fail("my visualizer didn't work")

```

The entire test suite can be run as follows::

```

$ pytest

```

You can also run your own test file as follows::

```

$ pytest tests/test\_your\_visualizer.py

```

The Makefile uses the pytest runner and testing suite as well as the coverage library, so make sure you have those dependencies installed!

\*\*Note\*\*: Advanced developers can use our \_image comparison tests\_ to assert that an image generated matches a baseline image. Read more about this in our [testing documentation](https://www.scikit-yb.org/en/latest/contributing/developing\_visualizers.html#image-comparison-tests).

### 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).

kwargs : dict

Keyword arguments that are passed to the base class and may influence

the visualization as defined in other Visualizers.

Examples

--------

>>> model = MyVisualizer()

>>> model.fit(X)

>>> model.show()

Notes

-----

In the notes section specify any gotchas or other info.

"""

```

This is a very good start to producing a high quality visualizer, but unless it is part of the documentation on our website, it will not be visible. For details on including documentation in the `docs` directory see the [Contributing Documentation](https://www.scikit-yb.org/en/latest/contributing/index.html) section in the larger contributing guide.