Developing with the Plotting API

Scikit-learn defines a simple API for creating visualizations for machine learning. The key features of this API is to run calculations once and to have the flexibility to adjust the visualizations after the fact. This section is intended for developers who wish to develop or maintain plotting tools. For usage, users should refer to the :ref`User Guide <visualizations>`.

Plotting API Overview

This logic is encapsulated into a display object where the computed data is stored and the plotting is done in a `plot` method. The display object's `__init__` method contains only the data needed to create the visualization. The `plot` method takes in parameters that only have to do with visualization, such as a matplotlib axes. The `plot` method will store the matplotlib artists as attributes allowing for style adjustments through the display object. A `plot_*` helper function accepts parameters to do the computation and the parameters used for plotting. After the helper function creates the display object with the computed values, it calls the display's plot method. Note that the `plot` method defines attributes related to matplotlib, such as the line artist. This allows for customizations after calling the `plot` method.

For example, the `RocCurveDisplay` defines the following methods and attributes:

```python
class RocCurveDisplay:
    def __init__(self, fpr, tpr, roc_auc, estimator_name):
        ...
        self.fpr = fpr
        self.tpr = tpr
        self.roc_auc = roc_auc
        self.estimator_name = estimator_name

    def plot(self, ax=None, name=None, **kwargs):
        ...
        self.line_ = ...
        self.ax_ = ax
        self.figure_ = ax.figure_

def plot_roc_curve(estimator, X, y, pos_label=None, sample_weight=None,
                   drop_intermediate=True, response_method="auto",
                   name=None, ax=None, **kwargs):
    # do computation
    viz = RocCurveDisplay(fpr, tpr, roc_auc,
                          estimator.__class__.__name__)
    return viz.plot(ax=ax, name=name, **kwargs)
```

Read more in ROC Curve with Visualization API and the User Guide.

Plotting with Multiple Axes

Some of the plotting tools like **`plot_partial_dependence`** and **`PartialDependenceDisplay`** support plottong on multiple axes. Two different scenarios are supported:

1. If a list of axes is passed in, `plot` will check if the number of axes is consistent with the number of axes it expects and then draws on those axes. 2. If a single axes is passed in, that axes defines a space for multiple axes to be placed. In this case, we suggest using matplotlib's `~matplotlib.gridspec.GridSpecFromSubplotSpec` to split up the space:

```python
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpecFromSubplotSpec

fig, ax = plt.subplots()
gs = GridSpecFromSubplotSpec(2, 2, subplot_spec=ax.get_subplotspec())

ax_top_left = fig.add_subplot(gs[0, 0])
ax_top_right = fig.add_subplot(gs[0, 1])
ax_bottom = fig.add_subplot(gs[1, :])
```

By default, the `ax` keyword in `plot` is `None`. In this case, the single axes is created and the gridspec api is used to create the regions to plot in.

See for example, **`plot_partial_dependence`** which plots multiple lines and contours using this API. The axes defining the bounding box is saved in a `bounding_ax_` attribute. The individual axes created are stored in an `axes_` ndarray, corresponding to the axes position on the grid. Positions that are not used are set to `None`. Furthermore, the matplotlib Artists are stored in `lines_` and `contours_` where the key is the position on the grid. When a list of axes is passed in, the `axes_`, `lines_`, and `contours_` is a 1d ndarray corresponding to the list of axes passed in.

Toggle Menu