Step 3: Move reusable code into .py modules

Moving code into distinct modules makes it easier to reuse them. Rather than copy-pasting parts of a script across different projects, you can maintain a single copy and include that copy in multiple projects. This not only speeds up your development process, but also makes it easier to collaborate on projects. After all, now you can change a module and all projects depending on it will automatically change as well.

Transforming code from a notebook into a module is relatively straightforward. You simply copy the code, wrap it in a class or function, save it as a `.py` file, and import that file in the script where you would like to use it. You don't need to change anything to the code itself; just how it's part of the notebook.

For example, if we have the following function:

```
def plot_confusion_matrix(cm: np.array,
                          target_names: List[Text],
                          title: Text = 'Confusion matrix',
                          cmap: matplotlib.colors.LinearSegmentedColormap
= None,
                          normalize: bool = True):
    """
    given a sklearn confusion matrix (cm), make a nice plot
    Arguments
    _________
    cm:           confusion matrix from sklearn.metrics.confusion_matrix
    target_names: given classification classes such as [0, 1, 2]
                  the class names, for example: ['high', 'medium', 'low']
    title:        the text to display at the top of the matrix
    cmap:         the gradient of the values displayed from
matplotlib.pyplot.cm
                  see
http://matplotlib.org/examples/color/colormaps_reference.html
                  plt.get_cmap('jet') or plt.cm.Blues
    normalize:    If False, plot the raw numbers
                  If True, plot the proportions
    Usage
    _____
    plot_confusion_matrix(cm           = cm,                  # confusion
matrix created by
                                                              #
sklearn.metrics.confusion_matrix
                          normalize    = True,                # show
proportions
                          target_names = y_labels_vals,       # list of
names of the classes
                          title        = best_estimator_name) # title of
graph
    Citiation
    _________
    http://scikit-
learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
    """

    accuracy = np.trace(cm) / float(np.sum(cm))
    misclass = 1 - accuracy

    if cmap is None:
        cmap = plt.get_cmap('Blues')

    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()

    if target_names is not None:
        tick_marks = np.arange(len(target_names))
        plt.xticks(tick_marks, target_names, rotation=45)
        plt.yticks(tick_marks, target_names)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 1.5 if normalize else cm.max() / 2
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        if normalize:
            plt.text(j, i, "{:0.4f}".format(cm[i, j]),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")
        else:
            plt.text(j, i, "{:,}".format(cm[i, j]),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label\naccuracy={:0.4f}; misclass=
{:0.4f}'.format(accuracy, misclass))

    return plt.gcf()
```

We can copy/paste that to a new `visualize.py` file and include its dependencies there:

```
import itertools
import matplotlib.colors
import matplotlib.pyplot as plt
import numpy as np
from typing import List, Text
```

Once we have done so, we can import the `visualize.py` module in our notebook and use its functions:

```
from src.report.visualization import plot_confusion_matrix
cm_plot = plot_confusion_matrix(cm, data.target_names, normalize=False)
```

`src` directory

In software development, it is a convention to include all source code in a dedicated `src` directory. This directory is also part of the generic repository structure we showed earlier. Within `src` you can create any structure you like. It is often convenient to categorize modules based on the stage in the pipeline they belong to. You could, for example, group modules in the following directories:

- `data`
- `evaluate`
- `features`
- `report`
- `train`

On the contrary, the Jupyter Notebook files would go into a dedicated `notebooks` directory that isn't part of `src`.