Microsoft: DAT210x Programming with Python for Data Science

Bookmark

Pipelining

You've already seen the power of cross validation, and how integrating it with grid searches help power-tune your predictor's parameters to maximize accuracy. But what do you do when you're running multiple machine learning algorithms on your data? SciKit-Learn's cross validation method only takes in a single estimator as input. Sure, you could run cross validation multiple times, but that defeat the purpose and it'd also allow knowledge from one layer of your withheld data to leak into the underlying layers, as we've seen earlier in this module. Moreover, each of your models would be trained *separately* rather than in tandem. Not the greatest approach when your goal is to the find the best set of parameters that maximizes **all** of them.

To work around this, and to make your code a lot cleaner, SciKit-Learn has created a pipelining class. It wraps around your entire data analysis pipeline from start to finish, and allows you to interact with the pipeline as if it were a single white-box, configurable estimator. The other added benefit is that once your pipeline has been built, since the pipeline inherits from the `estimator` base class, you can use it pretty much anywhere you'd use regular estimators—including in your cross validator method. Doing so, you can simultaneously fine tune the parameters of each of the estimators and predictors that comprise your data-analysis pipeline.

Usage

If you don't want to encounter errors, there are a few rules you must abide by while using SciKit-Learn's pipeline:

Dive Deeper

- Every intermediary model, or step within the pipeline must be a *transformer*. That means its class must implement both the `.fit()` and the `.transform()` methods. This is rather important, as the output from each step will serve as the input to the subsequent step! Every algorithm you've learned about in this class implements `.fit()` so you're good there, but not all of them implement `.transform()`. Be sure to take a look at the SciKit-Learn documentation for each algorithm to learn if it qualifier as a transformer, and make note of that on your course map.
- The very last step in your analysis pipeline only needs to implement the `.fit()` method, since it will not be feeding data into another step

The code to get up and running with your own pipelines looks like this:

```
>>> from sklearn.pipeline import Pipeline

>>> svc = svm.SVC(kernel='linear')
>>> pca = RandomizedPCA()

>>> pipeline = Pipeline([
  ('pca', pca),
  ('svc', svc)
])
>>> pipeline.set_params(pca__n_components=5, svc__C=1, svc__gamma=0.0001)
>>> pipeline.fit(X, y)
```

Notice that when you define parameters, you have to lead with the name you specified for that parameter when you added it to your pipeline, followed by two underscores and the parameter name. This is important because there are many estimators that share the same parameter names within SciKit-Learn's API. Without this, there would be ambiguity.

The pipeline class only has a single attribute called `.named_steps`, which is a dictionary containing the estimator names you specified as keys. You can use it to gain access to the underlying estimator steps within your pipeline. Besides directly specifying estimators, you can also have feature unions and nested pipelines as well! On top of that, you can implement your own custom transformers as a minimal class, so long as you provide end-points for `.fit()`, and `.transform()`. For a more in-depth study of this, be sure to check out the links included in the Dive Deeper section.

Many of the predictors you learned about in the last few chapters don't actually implement `.transform()`! Due to this, by default, you won't be able to use SVC, Linear Regression, or Decision Trees, etc. as intermediary steps within your pipeline. A very nifty hack you should be aware of to circumvent this is by writing your own transformer class, which simply wraps a predictor and masks it as a transformer:

```
from sklearn.base import TransformerMixin

class ModelTransformer(TransformerMixin):
  def __init__(self, model):
    self.model = model

  def fit(self, *args, **kwargs):
    self.model.fit(*args, **kwargs)
    return self

  def transform(self, X, **transform_params):
    # This is the magic =)
    return DataFrame(self.model.predict(X))
```