Using Pipelines for multiple preprocessing steps

Pipelines can be used to tie together many steps into one object. This allows for easier tuning and better access to the configuration of the entire model, not just one of the steps.

Getting ready

- This is the first section where we'll combine multiple data processing steps into a single step.
- In scikit-learn, this is known as a Pipeline.
- In this section, we'll first deal with missing data via imputation.
- However, after that, we'll scale the data to get a mean of zero and a standard deviation of one.

Let's create a dataset that is missing some values, and then we'll look at how to create a Pipeline:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> mat = datasets.make_spd_matrix(10)
>>> masking_array = np.random.binomial(1, .1, mat.shape).astype(bool)
>>> mat[masking_array] = np.nan
>>> mat[:4, :4]
array([[ 0.56716186, -0.20344151, nan, -0.22579163],
       [ nan, 1.98881836, -2.25445983, 1.27024191],
       [ 0.29327486, -2.25445983, 3.15525425, -1.64685403],
       [-0.22579163, 1.27024191, -1.64685403, 1.32240835]])
```

Great, now we can create a Pipeline.

How to do it

Without Pipelines, the process will look something like the following:

```
>>> from sklearn import preprocessing
>>> impute = preprocessing.Imputer()
>>> scaler = preprocessing.StandardScaler()
>>> mat_imputed = impute.fit_transform(mat)
>>> mat_imputed[:4, :4]
array([[ 0.56716186, -0.20344151, -0.80554023, -0.22579163],
[ 0.04235695, 1.98881836, -2.25445983, 1.27024191],
[ 0.29327486, -2.25445983, 3.15525425, -1.64685403],
[-0.22579163, 1.27024191, -1.64685403, 1.32240835]])
>>> mat_imp_and_scaled = scaler.fit_transform(mat_imputed)
array([[ 2.235e+00, -6.291e-01, 1.427e-16, -7.496e-01],
[ 0.000e+00, 1.158e+00, -9.309e-01, 9.072e-01],
[ 1.068e+00, -2.301e+00, 2.545e+00, -2.323e+00],
[ -1.142e+00, 5.721e-01, -5.405e-01, 9.650e-01]])
```

Notice that the prior missing value is 0. This is expected because this value was imputed using the mean strategy, and scaling subtracts the mean. Now that we've looked at a non-Pipeline example, let's look at how we can incorporate a Pipeline:

```
>>> from sklearn import pipeline
>>> pipe = pipeline.Pipeline([('impute', impute), ('scaler', scaler)])
```

Take a look at the Pipeline. As we can see, Pipeline defines the steps that designate the progression of methods:

```
>>> pipe
Pipeline(steps=[('impute', Imputer(axis=0, copy=True, missing_
values='NaN', strategy='mean', verbose=0)), ('scalar',
StandardScaler(copy=True, with_mean=True, with_std=True))])
```

This is the best part; simply call the `fit_transform` method on the pipe object. These separate steps are completed in a single step:

```
>>> new_mat = pipe.fit_transform(mat)
>>> new_mat [:4, :4]
array([[ 2.235e+00, -6.291e-01, 1.427e-16, -7.496e-01],
[ 0.000e+00, 1.158e+00, -9.309e-01, 9.072e-01],
[ 1.068e+00, -2.301e+00, 2.545e+00, -2.323e+00],
[ -1.142e+00, 5.721e-01, -5.405e-01, 9.650e-01]])
```

```
>>> np.array_equal(new_mat, mat_imp_and_scaled)
True
Beautiful!
```

Later in the book, we'll see just how powerful this concept is. It doesn't stop at preprocessing steps. It can easily extend to dimensionality reduction as well, fitting different learning methods. Dimensionality reduction is handled on it's own in the recipe Reducing dimensionality with PCA.

0.0.1 Implementation

As mentioned earlier, almost every scikit-learn has a similar interface. The important ones that allow Pipelines to function are:

- `fit`
- `transform`
- `fit_transform` (a convenience method)

To be specific, if a Pipeline has N objects, the first N-1 objects must implement both fit and transform, and the Nth object must implement at least fit. If this doesn't happen, an error will be thrown. Pipeline will work correctly if these conditions are met, but it is still possible that not every method will work properly. For example, pipe has a method, `inverse_transform`, which does exactly what the name entails. However, because the impute step doesn't have an `inverse_transform` method, this method call will fail:

```
>>> pipe.inverse_transform(new_mat)
AttributeError: 'Imputer' object has no attribute 'inverse_transform'
However, this is possible with the scalar object:
>>> scaler.inverse_transform(new_mat) [:4, :4]
array([[ 0.567, -0.203, -0.806, -0.226],
[ 0.042, 1.989, -2.254, 1.27 ],
[ 0.293, -2.254, 3.155, -1.647],
[-0.226, 1.27 , -1.647, 1.322]])
```

Once a proper Pipeline is set up, it functions almost exactly how you'd expect. It's a series of for loops that fit and transform at each intermediate step, feeding the output to the subsequent transformation. To conclude this recipe, I'll try to answer the "why?" question. There are two main reasons:

1. The first reason is convenience. The code becomes quite a bit cleaner; instead of calling fit and transform over and over, it is offloaded to sklearn.

2. The second, and probably the more important, reason is cross validation. Models can become very complex. If a single step in Pipeline has tuning parameters, they might need to be tested; with a single step, the code overhead to test the parameters is low.

However, five steps with all of their respective parameters can become difficult to test. Pipelines ease a lot of the burden.