Grid-Searching Preprocessing Steps and Model Parameters

Using pipelines, we can encapsulate all the processing steps in our machine learning workflow in a single `scikit-learn` estimator. Another benefit of doing this is that we can now *adjust the parameters of the preprocessing* using the outcome of a supervised task like regression or classification. In previous chapters, we used polynomial features on the `boston` dataset before applying the ridge regressor. Let's model that using a pipeline instead. The pipeline contains three steps—scaling the data, computing polynomial features, and ridge regression:

In[27]:

```
from sklearn.datasets import load_boston
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target,
                                                    random_state=0)

from sklearn.preprocessing import PolynomialFeatures
pipe = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(),
    Ridge())
```

How do we know which degrees of polynomials to choose, or whether to choose any polynomials or interactions at all? Ideally we want to select the `degree` parameter based on the outcome of the classification. Using our pipeline, we can search over the `degree` parameter together with the parameter `alpha` of `Ridge`. To do this, we define a `param_grid` that contains both, appropriately prefixed by the step names:

In[28]:

```
param_grid = {'polynomialfeatures__degree': [1, 2, 3],
              'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
```

Now we can run our grid search again:

In[29]:

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5, n_jobs=-1)
grid.fit(X_train, y_train)
```

We can visualize the outcome of the cross-validation using a heat map (Figure 6-4), as we did in Chapter 5:

In[30]:

```
plt.matshow(grid.cv_results_['mean_test_score'].reshape(3, -1),
            vmin=0, cmap="viridis")
plt.xlabel("ridge__alpha")
plt.ylabel("polynomialfeatures__degree")
```

```
plt.xticks(range(len(param_grid['ridge__alpha'])), param_grid['ridge__alpha'])
plt.yticks(range(len(param_grid['polynomialfeatures__degree'])),
           param_grid['polynomialfeatures__degree'])

plt.colorbar()
```

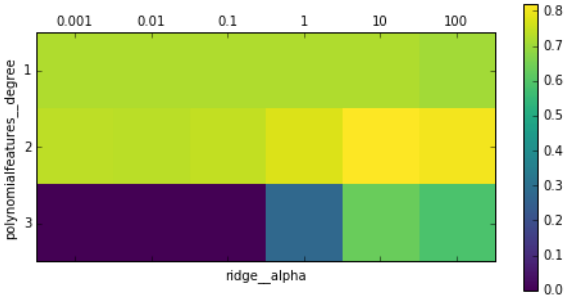

Figure 6-4. Heat map of mean cross-validation score as a function of the degree of the polynomial features and alpha parameter of Ridge

Looking at the results produced by the cross-validation, we can see that using polynomials of degree two helps, but that degree-three polynomials are much worse than either degree one or two. This is reflected in the best parameters that were found:

In[31]:

```
print("Best parameters: {}".format(grid.best_params_))
```

Out[31]:

```
Best parameters: {'polynomialfeatures__degree': 2, 'ridge__alpha': 10}
```

Which lead to the following score:

In[32]:

```
print("Test-set score: {:.2f}".format(grid.score(X_test, y_test)))
```

Out[32]:

```
Test-set score: 0.77
```

Let's run a grid search without polynomial features for comparison:

In[33]:

```
param_grid = {'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
pipe = make_pipeline(StandardScaler(), Ridge())
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
print("Score without poly features: {:.2f}".format(grid.score(X_test, y_test)))
```

Out[33]:

```
Score without poly features: 0.63
```

As we would expect looking at the grid search results visualized in Figure 6-4, using no polynomial features leads to decidedly worse results.

Searching over preprocessing parameters together with model parameters is a very powerful strategy. However, keep in mind that `GridSearchCV` tries *all possible combinations* of the specified parameters. Therefore, adding more parameters to your grid exponentially increases the number of models that need to be built.

Grid-Searching Which Model To Use

You can even go further in combining `GridSearchCV` and `Pipeline`: it is also possible to search over the actual steps being performed in the pipeline (say whether to use `StandardScaler` or `MinMaxScaler`). This leads to an even bigger search space and should be considered carefully. Trying all possible solutions is usually not a viable machine learning strategy. However, here is an example comparing a `RandomForestClassifier` and an `SVC` on the `iris` dataset. We know that the `SVC` might need the data to be scaled, so we also search over whether to use `StandardScaler` or no preprocessing. For the `RandomForestClassifier`, we know that no preprocessing is necessary. We start by defining the pipeline. Here, we explicitly name the steps. We want two steps, one for the preprocessing and then a classifier. We can instantiate this using `SVC` and `StandardScaler`:

In[34]:

```
pipe = Pipeline([('preprocessing', StandardScaler()), ('classifier', SVC())])
```

Now we can define the `parameter_grid` to search over. We want the `classifier` to be either `RandomForestClassifier` or `SVC`. Because they have different parameters to tune, and need different preprocessing, we can make use of the list of search grids we discussed in "Search over spaces that are not grids" on page 271. To assign an estimator to a step, we use the name of the step as the parameter name. When we wanted to skip a step in the pipeline (for example, because we don't need preprocessing for the `RandomForest`), we can set that step to `None`:

In[35]:

```
from sklearn.ensemble import RandomForestClassifier

param_grid = [
    {'classifier': [SVC()], 'preprocessing': [StandardScaler(), None],
     'classifier__gamma': [0.001, 0.01, 0.1, 1, 10, 100],
     'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100]},
    {'classifier': [RandomForestClassifier(n_estimators=100)],
     'preprocessing': [None], 'classifier__max_features': [1, 2, 3]}]
```