The pipeline objects `pipe_long` and `pipe_short` do exactly the same thing, but `pipe_short` has steps that were automatically named. We can see the names of the steps by looking at the `steps` attribute:

In[18]:

```
print("Pipeline steps:\n{}".format(pipe_short.steps))
```

Out[18]:

```
Pipeline steps:
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))),
 ('svc', SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
             decision_function_shape=None, degree=3, gamma='auto',
             kernel='rbf', max_iter=-1, probability=False,
             random_state=None, shrinking=True, tol=0.001,
             verbose=False))]
```

The steps are named `minmaxscaler` and `svc`. In general, the step names are just lowercase versions of the class names. If multiple steps have the same class, a number is appended:

In[19]:

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

pipe = make_pipeline(StandardScaler(), PCA(n_components=2), StandardScaler())
print("Pipeline steps:\n{}".format(pipe.steps))
```

Out[19]:

```
Pipeline steps:
[('standardscaler-1', StandardScaler(copy=True, with_mean=True, with_std=True)),
 ('pca', PCA(copy=True, iterated_power=4, n_components=2, random_state=None,
             svd_solver='auto', tol=0.0, whiten=False)),
 ('standardscaler-2', StandardScaler(copy=True, with_mean=True, with_std=True))]
```

As you can see, the first `StandardScaler` step was named `standardscaler-1` and the second `standardscaler-2`. However, in such settings it might be better to use the `Pipeline` construction with explicit names, to give more semantic names to each step.

Accessing Step Attributes

Often you will want to inspect attributes of one of the steps of the pipeline—say, the coefficients of a linear model or the components extracted by `PCA`. The easiest way to access the steps in a pipeline is via the `named_steps` attribute, which is a dictionary from the step names to the estimators:

In[20]:

```
# fit the pipeline defined before to the cancer dataset
pipe.fit(cancer.data)
# extract the first two principal components from the "pca" step
components = pipe.named_steps["pca"].components_
print("components.shape: {}".format(components.shape))
```

Out[20]:

```
components.shape: (2, 30)
```

Accessing Attributes in a Grid-Searched Pipeline

As we discussed earlier in this chapter, one of the main reasons to use pipelines is for doing grid searches. A common task is to access some of the steps of a pipeline inside a grid search. Let's grid search a `LogisticRegression` classifier on the `cancer` dataset, using `Pipeline` and `StandardScaler` to scale the data before passing it to the `LogisticRegression` classifier. First we create a pipeline using the `make_pipeline` function:

In[21]:

```
from sklearn.linear_model import LogisticRegression

pipe = make_pipeline(StandardScaler(), LogisticRegression())
```

Next, we create a parameter grid. As explained in Chapter 2, the regularization parameter to tune for `LogisticRegression` is the parameter `C`. We use a logarithmic grid for this parameter, searching between 0.01 and 100. Because we used the `make_pipeline` function, the name of the `LogisticRegression` step in the pipeline is the lowercased class name, `logisticregression`. To tune the parameter `C`, we therefore have to specify a parameter grid for `logisticregression__C`:

In[22]:

```
param_grid = {'logisticregression__C': [0.01, 0.1, 1, 10, 100]}
```

As usual, we split the `cancer` dataset into training and test sets, and fit a grid search:

In[23]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=4)
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
```

So how do we access the coefficients of the best `LogisticRegression` model that was found by `GridSearchCV`? From Chapter 5 we know that the best model found by `GridSearchCV`, trained on all the training data, is stored in `grid.best_estimator_`: