tor. The `Pipeline` class itself has `fit`, `predict`, and `score` methods and behaves just like any other model in `scikit-learn`. The most common use case of the `Pipeline` class is in chaining preprocessing steps (like scaling of the data) together with a supervised model like a classifier.

Building Pipelines

Let's look at how we can use the `Pipeline` class to express the workflow for training an SVM after scaling the data with `MinMaxScaler` (for now without the grid search). First, we build a pipeline object by providing it with a list of steps. Each step is a tuple containing a name (any string of your choosing[1]) and an instance of an estimator:

In[5]:

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC())])
```

Here, we created two steps: the first, called `"scaler"`, is an instance of `MinMaxScaler`, and the second, called `"svm"`, is an instance of `SVC`. Now, we can fit the pipeline, like any other `scikit-learn` estimator:

In[6]:

```
pipe.fit(X_train, y_train)
```

Here, `pipe.fit` first calls `fit` on the first step (the scaler), then transforms the training data using the scaler, and finally fits the SVM with the scaled data. To evaluate on the test data, we simply call `pipe.score`:

In[7]:

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

Out[7]:

```
Test score: 0.95
```

Calling the `score` method on the pipeline first transforms the test data using the scaler, and then calls the `score` method on the SVM using the scaled test data. As you can see, the result is identical to the one we got from the code at the beginning of the chapter, when doing the transformations by hand. Using the pipeline, we reduced the code needed for our "preprocessing + classification" process. The main benefit of using the pipeline, however, is that we can now use this single estimator in `cross_val_score` or `GridSearchCV`.

1 With one exception: the name can't contain a double underscore, `__`.

Using Pipelines in Grid Searches

Using a pipeline in a grid search works the same way as using any other estimator. We define a parameter grid to search over, and construct a `GridSearchCV` from the pipeline and the parameter grid. When specifying the parameter grid, there is a slight change, though. We need to specify for each parameter which step of the pipeline it belongs to. Both parameters that we want to adjust, `C` and `gamma`, are parameters of `SVC`, the second step. We gave this step the name `"svm"`. The syntax to define a parameter grid for a pipeline is to specify for each parameter the step name, followed by `__` (a double underscore), followed by the parameter name. To search over the `C` parameter of `SVC` we therefore have to use `"svm__C"` as the key in the parameter grid dictionary, and similarly for `gamma`:

In[8]:

```
param_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],
              'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

With this parameter grid we can use `GridSearchCV` as usual:

In[9]:

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation accuracy: {:.2f}".format(grid.best_score_))
print("Test set score: {:.2f}".format(grid.score(X_test, y_test)))
print("Best parameters: {}".format(grid.best_params_))
```

Out[9]:

```
Best cross-validation accuracy: 0.98
Test set score: 0.97
Best parameters: {'svm__C': 1, 'svm__gamma': 1}
```

In contrast to the grid search we did before, now for each split in the cross-validation, the `MinMaxScaler` is refit with only the training splits and no information is leaked from the test split into the parameter search. Compare this (Figure 6-2) with Figure 6-1 earlier in this chapter:

In[10]:

```
mglearn.plots.plot_proper_processing()
```