```
       1.,  1.,  1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
       1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
       1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
       1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
       1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
       1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Because we have 150 samples, the leave-one-out cross-validation yields scores for 150 trials, and the score indicates either successful (1.0) or unsuccessful (0.0) prediction. Taking the mean of these gives an estimate of the error rate:

```
In[9]: scores.mean()
```

```
Out[9]: 0.95999999999999996
```

Other cross-validation schemes can be used similarly. For a description of what is available in Scikit-Learn, use IPython to explore the `sklearn.cross_validation` submodule, or take a look at Scikit-Learn's online cross-validation documentation.

Selecting the Best Model

Now that we've seen the basics of validation and cross-validation, we will go into a little more depth regarding model selection and selection of hyperparameters. These issues are some of the most important aspects of the practice of machine learning, and I find that this information is often glossed over in introductory machine learning tutorials.

Of core importance is the following question: *if our estimator is underperforming, how should we move forward?* There are several possible answers:

- Use a more complicated/more flexible model
- Use a less complicated/less flexible model
- Gather more training samples
- Gather more data to add features to each sample

The answer to this question is often counterintuitive. In particular, sometimes using a more complicated model will give worse results, and adding more training samples may not improve your results! The ability to determine what steps will improve your model is what separates the successful machine learning practitioners from the unsuccessful.

The bias–variance trade-off

Fundamentally, the question of "the best model" is about finding a sweet spot in the trade-off between *bias* and *variance*. Consider Figure 5-24, which presents two regression fits to the same dataset.

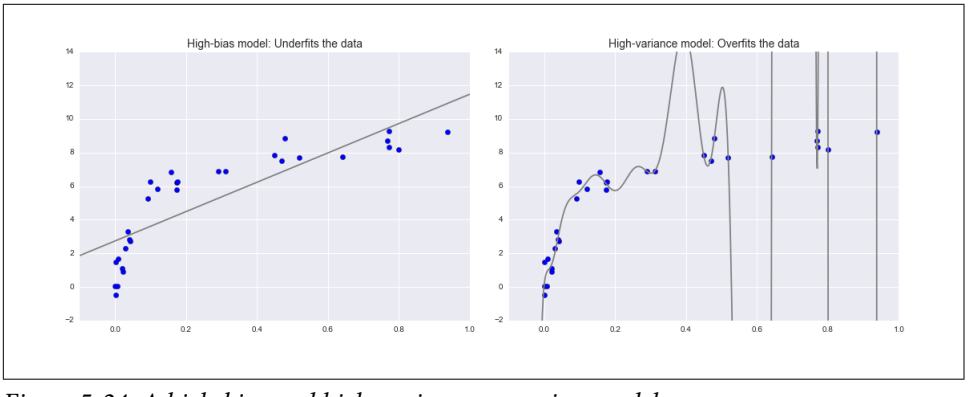

Figure 5-24. A high-bias and high-variance regression model

It is clear that neither of these models is a particularly good fit to the data, but they fail in different ways.

The model on the left attempts to find a straight-line fit through the data. Because the data are intrinsically more complicated than a straight line, the straight-line model will never be able to describe this dataset well. Such a model is said to *underfit* the data; that is, it does not have enough model flexibility to suitably account for all the features in the data. Another way of saying this is that the model has high *bias*.

The model on the right attempts to fit a high-order polynomial through the data. Here the model fit has enough flexibility to nearly perfectly account for the fine features in the data, but even though it very accurately describes the training data, its precise form seems to be more reflective of the particular noise properties of the data rather than the intrinsic properties of whatever process generated that data. Such a model is said to *overfit* the data; that is, it has so much model flexibility that the model ends up accounting for random errors as well as the underlying data distribution. Another way of saying this is that the model has high *variance*.

To look at this in another light, consider what happens if we use these two models to predict the *y*-value for some new data. In diagrams in Figure 5-25, the red/lighter points indicate data that is omitted from the training set.

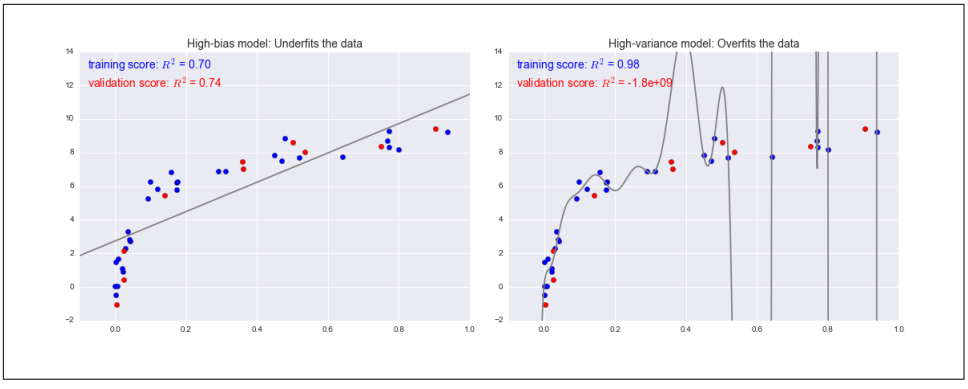

Figure 5-25. Training and validation scores in high-bias and high-variance models

The score here is the R^2 score, or coefficient of determination, which measures how well a model performs relative to a simple mean of the target values. $R^2 = 1$ indicates a perfect match, $R^2 = 0$ indicates the model does no better than simply taking the mean of the data, and negative values mean even worse models. From the scores associated with these two models, we can make an observation that holds more generally:

- For high-bias models, the performance of the model on the validation set is similar to the performance on the training set.
- For high-variance models, the performance of the model on the validation set is far worse than the performance on the training set.

If we imagine that we have some ability to tune the model complexity, we would expect the training score and validation score to behave as illustrated in Figure 5-26.

The diagram shown in Figure 5-26 is often called a *validation curve*, and we see the following essential features:

- The training score is everywhere higher than the validation score. This is generally the case: the model will be a better fit to data it has seen than to data it has not seen.
- For very low model complexity (a high-bias model), the training data is underfit, which means that the model is a poor predictor both for the training data and for any previously unseen data.
- For very high model complexity (a high-variance model), the training data is overfit, which means that the model predicts the training data very well, but fails for any previously unseen data.
- For some intermediate value, the validation curve has a maximum. This level of complexity indicates a suitable trade-off between bias and variance.

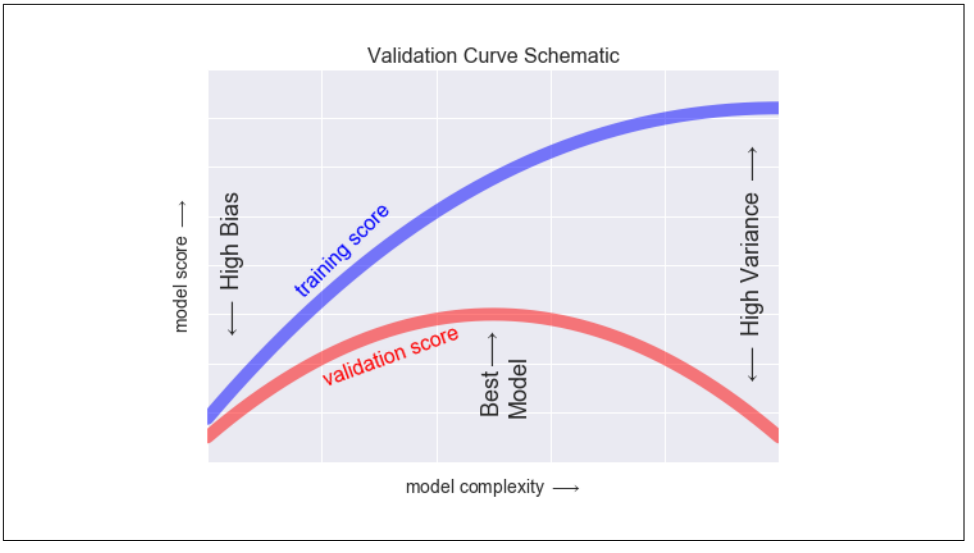

Figure 5-26. A schematic of the relationship between model complexity, training score, and validation score

The means of tuning the model complexity varies from model to model; when we discuss individual models in depth in later sections, we will see how each model allows for such tuning.

Validation curves in Scikit-Learn

Let's look at an example of using cross-validation to compute the validation curve for a class of models. Here we will use a *polynomial regression* model: this is a generalized linear model in which the degree of the polynomial is a tunable parameter. For example, a degree-1 polynomial fits a straight line to the data; for model parameters a and b:

$$y = ax + b$$

A degree-3 polynomial fits a cubic curve to the data; for model parameters a, b, c, d:

$$y = ax^3 + bx^2 + cx + d$$

We can generalize this to any number of polynomial features. In Scikit-Learn, we can implement this with a simple linear regression combined with the polynomial preprocessor. We will use a *pipeline* to string these operations together (we will discuss polynomial features and pipelines more fully in "Feature Engineering" on page 375):

```
In[10]: from sklearn.preprocessing import PolynomialFeatures
        from sklearn.linear_model import LinearRegression
        from sklearn.pipeline import make_pipeline

        def PolynomialRegression(degree=2, **kwargs):
            return make_pipeline(PolynomialFeatures(degree),
                                 LinearRegression(**kwargs))
```

Now let's create some data to which we will fit our model:

```
In[11]: import numpy as np

        def make_data(N, err=1.0, rseed=1):
            # randomly sample the data
            rng = np.random.RandomState(rseed)
            X = rng.rand(N, 1) ** 2
            y = 10 - 1. / (X.ravel() + 0.1)
            if err > 0:
                y += err * rng.randn(N)
            return X, y

        X, y = make_data(40)
```

We can now visualize our data, along with polynomial fits of several degrees (Figure 5-27):

```
In[12]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()  # plot formatting

        X_test = np.linspace(-0.1, 1.1, 500)[:, None]

        plt.scatter(X.ravel(), y, color='black')
        axis = plt.axis()
        for degree in [1, 3, 5]:
            y_test = PolynomialRegression(degree).fit(X, y).predict(X_test)
            plt.plot(X_test.ravel(), y_test, label='degree={0}'.format(degree))
        plt.xlim(-0.1, 1.0)
        plt.ylim(-2, 12)
        plt.legend(loc='best');
```

The knob controlling model complexity in this case is the degree of the polynomial, which can be any non-negative integer. A useful question to answer is this: what degree of polynomial provides a suitable trade-off between bias (underfitting) and variance (overfitting)?

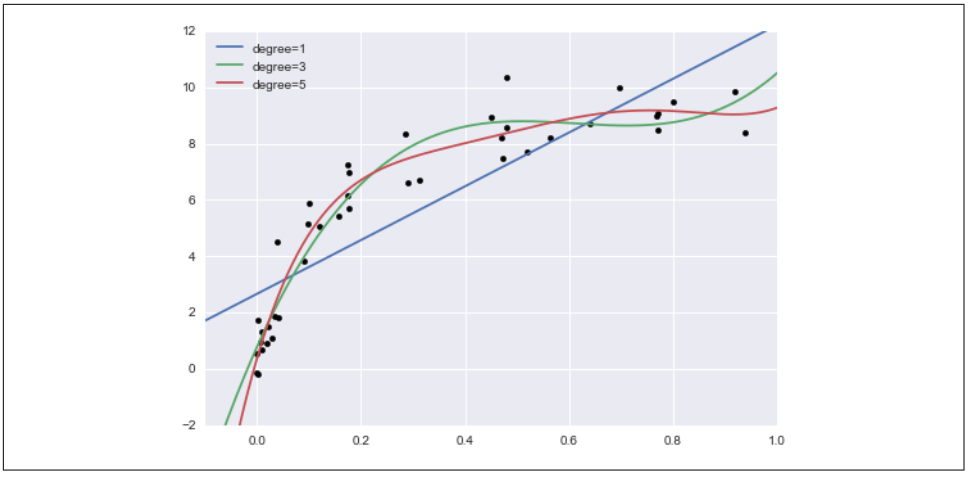

Figure 5-27. Three different polynomial models fit to a dataset

We can make progress in this by visualizing the validation curve for this particular data and model; we can do this straightforwardly using the `validation_curve` convenience routine provided by Scikit-Learn. Given a model, data, parameter name, and a range to explore, this function will automatically compute both the training score and validation score across the range (Figure 5-28):

```
In[13]:
from sklearn.learning_curve import validation_curve
degree = np.arange(0, 21)
train_score, val_score = validation_curve(PolynomialRegression(), X, y,
                                          'polynomialfeatures__degree',
                                          degree, cv=7)

plt.plot(degree, np.median(train_score, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score, 1), color='red', label='validation score')
plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

This shows precisely the qualitative behavior we expect: the training score is everywhere higher than the validation score; the training score is monotonically improving with increased model complexity; and the validation score reaches a maximum before dropping off as the model becomes overfit.

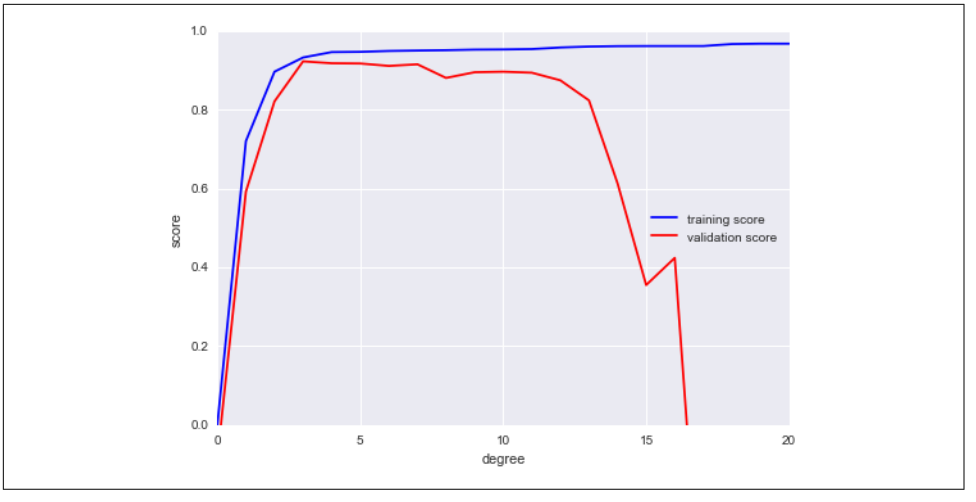

Figure 5-28. The validation curves for the data in Figure 5-27 (cf. Figure 5-26)

From the validation curve, we can read off that the optimal trade-off between bias and variance is found for a third-order polynomial; we can compute and display this fit over the original data as follows (Figure 5-29):

```
In[14]: plt.scatter(X.ravel(), y)
        lim = plt.axis()
        y_test = PolynomialRegression(3).fit(X, y).predict(X_test)
        plt.plot(X_test.ravel(), y_test);
        plt.axis(lim);
```

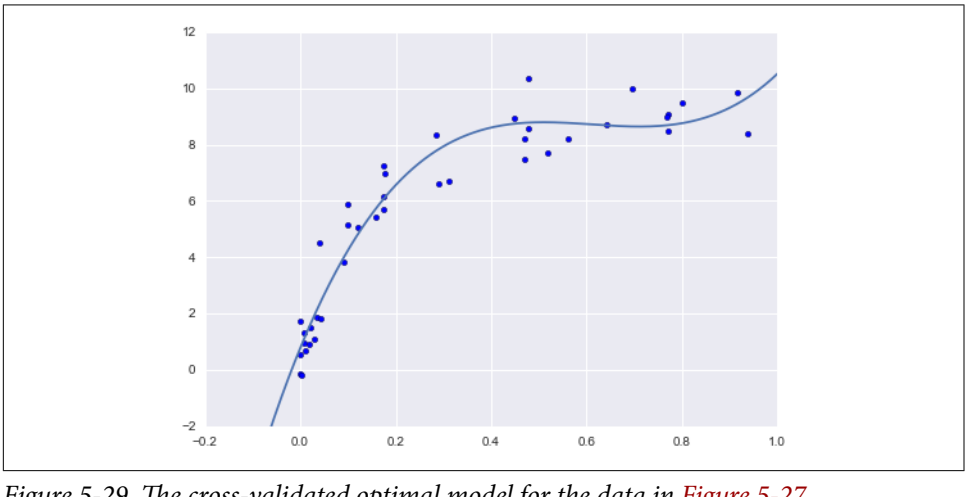

Figure 5-29. The cross-validated optimal model for the data in Figure 5-27

Notice that finding this optimal model did not actually require us to compute the training score, but examining the relationship between the training score and validation score can give us useful insight into the performance of the model.

Learning Curves

One important aspect of model complexity is that the optimal model will generally depend on the size of your training data. For example, let's generate a new dataset with a factor of five more points (Figure 5-30):

```
In[15]: X2, y2 = make_data(200)
        plt.scatter(X2.ravel(), y2);
```

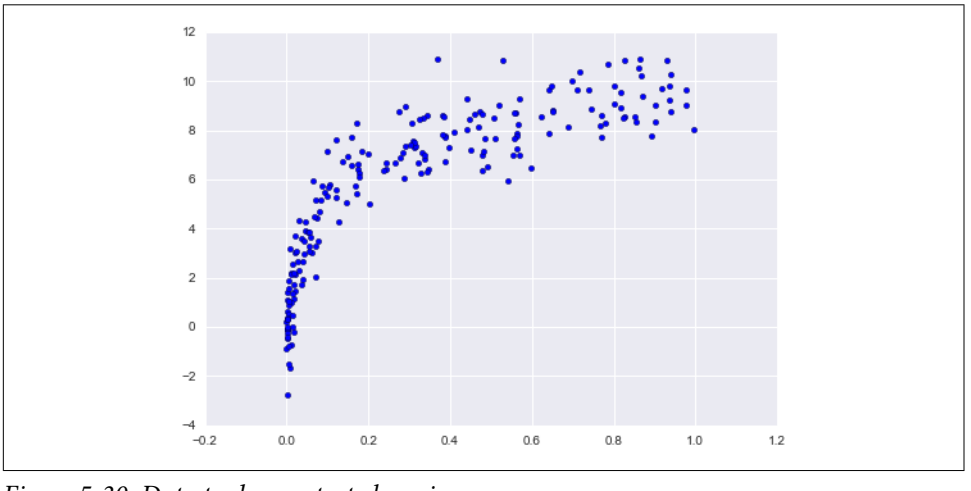

Figure 5-30. Data to demonstrate learning curves

We will duplicate the preceding code to plot the validation curve for this larger dataset; for reference let's over-plot the previous results as well (Figure 5-31):

```
In[16]:
degree = np.arange(21)
train_score2, val_score2 = validation_curve(PolynomialRegression(), X2, y2,
                                            'polynomialfeatures__degree',
                                            degree, cv=7)

plt.plot(degree, np.median(train_score2, 1), color='blue',
         label='training score')
plt.plot(degree, np.median(val_score2, 1), color='red', label='validation score')
plt.plot(degree, np.median(train_score, 1), color='blue', alpha=0.3,
         linestyle='dashed')
plt.plot(degree, np.median(val_score, 1), color='red', alpha=0.3,
         linestyle='dashed')
plt.legend(loc='lower center')
plt.ylim(0, 1)
```