```
    ax[i].set_ylim(0, 1)
    ax[i].set_xlim(N[0], N[-1])
    ax[i].set_xlabel('training size')
    ax[i].set_ylabel('score')
    ax[i].set_title('degree = {0}'.format(degree), size=14)
    ax[i].legend(loc='best')
```

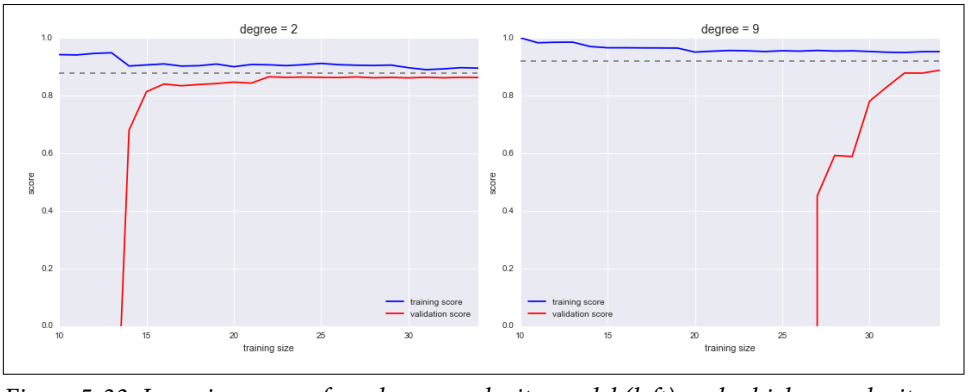

Figure 5-33. Learning curves for a low-complexity model (left) and a high-complexity model (right)

This is a valuable diagnostic, because it gives us a visual depiction of how our model responds to increasing training data. In particular, when your learning curve has already converged (i.e., when the training and validation curves are already close to each other), *adding more training data will not significantly improve the fit!* This situation is seen in the left panel, with the learning curve for the degree-2 model.

The only way to increase the converged score is to use a different (usually more complicated) model. We see this in the right panel: by moving to a much more complicated model, we increase the score of convergence (indicated by the dashed line), but at the expense of higher model variance (indicated by the difference between the training and validation scores). If we were to add even more data points, the learning curve for the more complicated model would eventually converge.

Plotting a learning curve for your particular choice of model and dataset can help you to make this type of decision about how to move forward in improving your analysis.

Validation in Practice: Grid Search

The preceding discussion is meant to give you some intuition into the trade-off between bias and variance, and its dependence on model complexity and training set size. In practice, models generally have more than one knob to turn, and thus plots of validation and learning curves change from lines to multidimensional surfaces. In these cases, such visualizations are difficult and we would rather simply find the particular model that maximizes the validation score.

Scikit-Learn provides automated tools to do this in the `grid_search` module. Here is an example of using grid search to find the optimal polynomial model. We will explore a three-dimensional grid of model features—namely, the polynomial degree, the flag telling us whether to fit the intercept, and the flag telling us whether to normalize the problem. We can set this up using Scikit-Learn's `GridSearchCV` meta-estimator:

```
In[18]: from sklearn.grid_search import GridSearchCV

        param_grid = {'polynomialfeatures__degree': np.arange(21),
                      'linearregression__fit_intercept': [True, False],
                      'linearregression__normalize': [True, False]}

        grid = GridSearchCV(PolynomialRegression(), param_grid, cv=7)
```

Notice that like a normal estimator, this has not yet been applied to any data. Calling the `fit()` method will fit the model at each grid point, keeping track of the scores along the way:

```
In[19]: grid.fit(X, y);
```

Now that this is fit, we can ask for the best parameters as follows:

```
In[20]: grid.best_params_

Out[20]: {'linearregression__fit_intercept': False,
          'linearregression__normalize': True,
          'polynomialfeatures__degree': 4}
```

Finally, if we wish, we can use the best model and show the fit to our data using code from before (Figure 5-34):

```
In[21]: model = grid.best_estimator_

        plt.scatter(X.ravel(), y)
        lim = plt.axis()
        y_test = model.fit(X, y).predict(X_test)
        plt.plot(X_test.ravel(), y_test, hold=True);
        plt.axis(lim);
```

The grid search provides many more options, including the ability to specify a custom scoring function, to parallelize the computations, to do randomized searches, and more. For information, see the examples in "In-Depth: Kernel Density Estimation" on page 491 and "Application: A Face Detection Pipeline" on page 506, or refer to Scikit-Learn's grid search documentation.

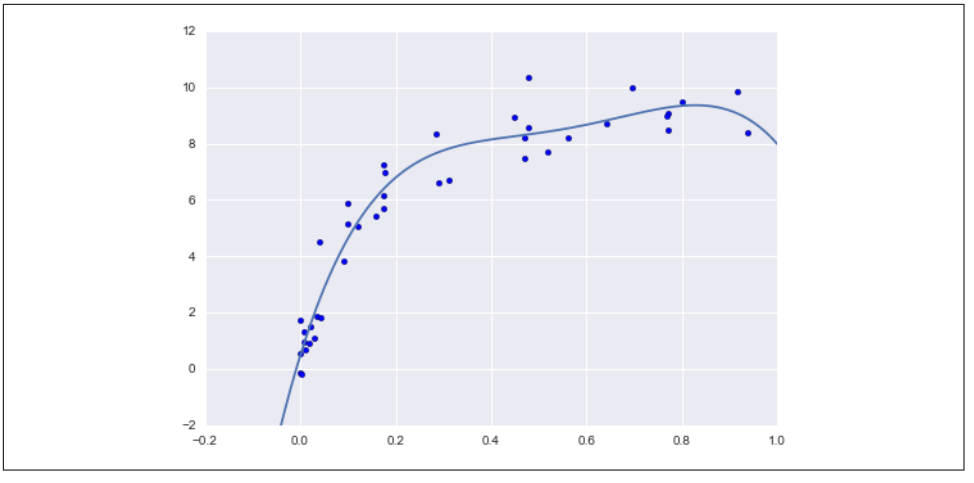

Figure 5-34. The best-fit model determined via an automatic grid-search

Summary

In this section, we have begun to explore the concept of model validation and hyperparameter optimization, focusing on intuitive aspects of the bias–variance trade-off and how it comes into play when fitting models to data. In particular, we found that the use of a validation set or cross-validation approach is *vital* when tuning parameters in order to avoid overfitting for more complex/flexible models.

In later sections, we will discuss the details of particularly useful models, and throughout will talk about what tuning is available for these models and how these free parameters affect model complexity. Keep the lessons of this section in mind as you read on and learn about these machine learning approaches!

Feature Engineering

The previous sections outline the fundamental ideas of machine learning, but all of the examples assume that you have numerical data in a tidy, `[n_samples, n_features]` format. In the real world, data rarely comes in such a form. With this in mind, one of the more important steps in using machine learning in practice is *feature engineering*—that is, taking whatever information you have about your problem and turning it into numbers that you can use to build your feature matrix.

In this section, we will cover a few common examples of feature engineering tasks: features for representing *categorical data*, features for representing *text*, and features for representing *images*. Additionally, we will discuss *derived features* for increasing model complexity and *imputation* of missing data. Often this process is known as *vectorization*, as it involves converting arbitrary data into well-behaved vectors.