```
            return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                     self.width_, axis=1)

gauss_model = make_pipeline(GaussianFeatures(20),
                            LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);
```

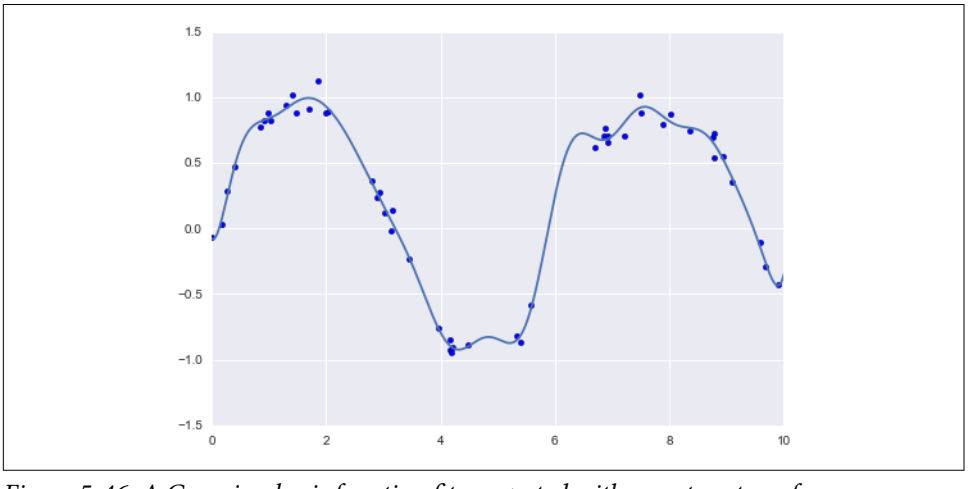

Figure 5-46. A Gaussian basis function fit computed with a custom transformer

We put this example here just to make clear that there is nothing magic about polynomial basis functions: if you have some sort of intuition into the generating process of your data that makes you think one basis or another might be appropriate, you can use them as well.

Regularization

The introduction of basis functions into our linear regression makes the model much more flexible, but it also can very quickly lead to overfitting (refer back to "Hyperparameters and Model Validation" on page 359 for a discussion of this). For example, if we choose too many Gaussian basis functions, we end up with results that don't look so good (Figure 5-47):

```
In[10]: model = make_pipeline(GaussianFeatures(30),
                              LinearRegression())
        model.fit(x[:, np.newaxis], y)

        plt.scatter(x, y)
        plt.plot(xfit, model.predict(xfit[:, np.newaxis]))
```

```
plt.xlim(0, 10)
plt.ylim(-1.5, 1.5);
```

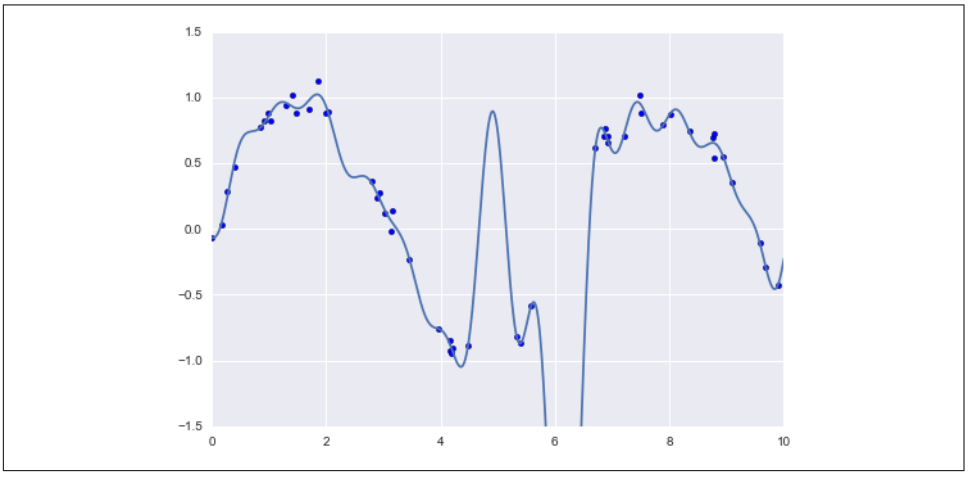

Figure 5-47. An overly complex basis function model that overfits the data

With the data projected to the 30-dimensional basis, the model has far too much flexibility and goes to extreme values between locations where it is constrained by data. We can see the reason for this if we plot the coefficients of the Gaussian bases with respect to their locations (Figure 5-48):

```
In[11]: def basis_plot(model, title=None):
            fig, ax = plt.subplots(2, sharex=True)
            model.fit(x[:, np.newaxis], y)
            ax[0].scatter(x, y)
            ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
            ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

            if title:
                ax[0].set_title(title)

            ax[1].plot(model.steps[0][1].centers_,
                       model.steps[1][1].coef_)
            ax[1].set(xlabel='basis location',
                      ylabel='coefficient',
                      xlim=(0, 10))

        model = make_pipeline(GaussianFeatures(30), LinearRegression())
        basis_plot(model)
```

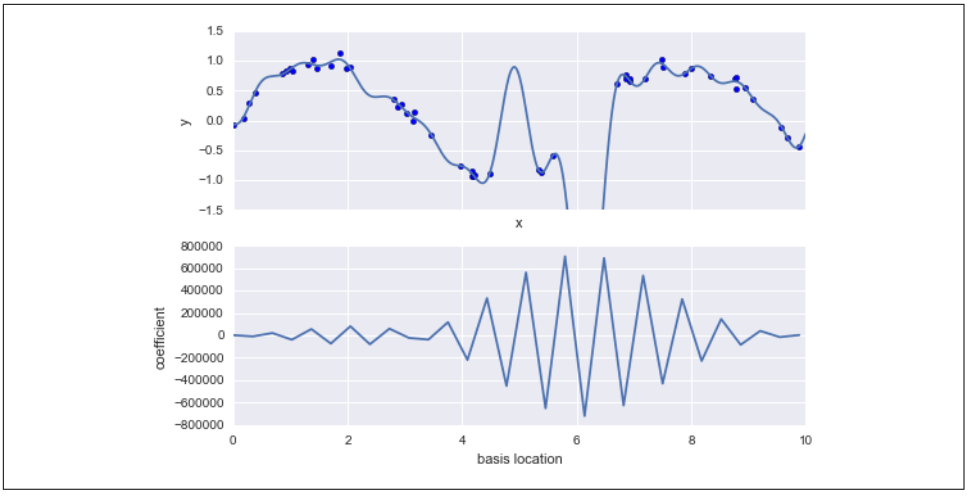

Figure 5-48. The coefficients of the Gaussian bases in the overly complex model

The lower panel in Figure 5-48 shows the amplitude of the basis function at each location. This is typical overfitting behavior when basis functions overlap: the coefficients of adjacent basis functions blow up and cancel each other out. We know that such behavior is problematic, and it would be nice if we could limit such spikes explicitly in the model by penalizing large values of the model parameters. Such a penalty is known as *regularization*, and comes in several forms.

Ridge regression (L_2 regularization)

Perhaps the most common form of regularization is known as *ridge regression* or L_2 *regularization*, sometimes also called *Tikhonov regularization*. This proceeds by penalizing the sum of squares (2-norms) of the model coefficients; in this case, the penalty on the model fit would be:

$$P = \alpha \sum_{n=1}^{N} \theta_n^2$$

where α is a free parameter that controls the strength of the penalty. This type of penalized model is built into Scikit-Learn with the `Ridge` estimator (Figure 5-49):

```
In[12]: from sklearn.linear_model import Ridge
        model = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
        basis_plot(model, title='Ridge Regression')
```

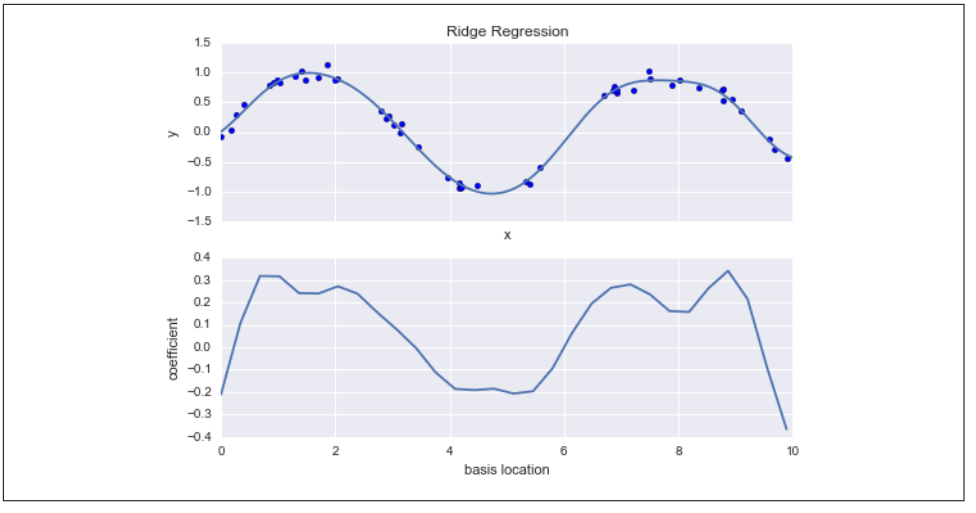

Figure 5-49. Ridge (L_2) regularization applied to the overly complex model (compare to Figure 5-48)

The α parameter is essentially a knob controlling the complexity of the resulting model. In the limit $\alpha \to 0$, we recover the standard linear regression result; in the limit $\alpha \to \infty$, all model responses will be suppressed. One advantage of ridge regression in particular is that it can be computed very efficiently—at hardly more computational cost than the original linear regression model.

Lasso regularization (L_1)

Another very common type of regularization is known as lasso, and involves penalizing the sum of absolute values (1-norms) of regression coefficients:

$$P = \alpha \Sigma_{n=1}^{N} |\theta_n|$$

Though this is conceptually very similar to ridge regression, the results can differ surprisingly: for example, due to geometric reasons lasso regression tends to favor *sparse models* where possible; that is, it preferentially sets model coefficients to exactly zero.

We can see this behavior in duplicating the plot shown in Figure 5-49, but using L1-normalized coefficients (Figure 5-50):

```
In[13]: from sklearn.linear_model import Lasso
        model = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.001))
        basis_plot(model, title='Lasso Regression')
```

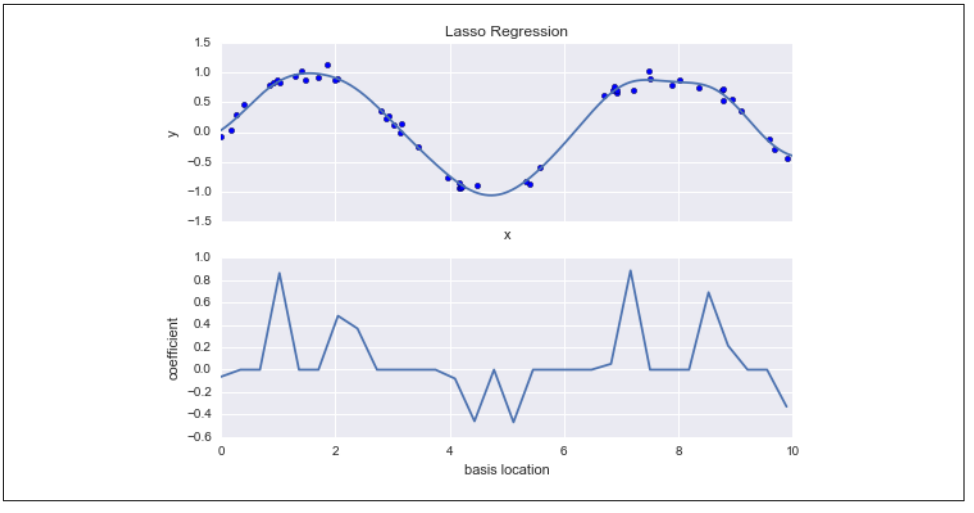

Figure 5-50. Lasso (L_1) regularization applied to the overly complex model (compare to Figure 5-48)

With the lasso regression penalty, the majority of the coefficients are exactly zero, with the functional behavior being modeled by a small subset of the available basis functions. As with ridge regularization, the α parameter tunes the strength of the penalty, and should be determined via, for example, cross-validation (refer back to "Hyperparameters and Model Validation" on page 359 for a discussion of this).

Example: Predicting Bicycle Traffic

As an example, let's take a look at whether we can predict the number of bicycle trips across Seattle's Fremont Bridge based on weather, season, and other factors. We have seen this data already in "Working with Time Series" on page 188.

In this section, we will join the bike data with another dataset, and try to determine the extent to which weather and seasonal factors—temperature, precipitation, and daylight hours—affect the volume of bicycle traffic through this corridor. Fortunately, the NOAA makes available their daily weather station data (I used station ID USW00024233) and we can easily use Pandas to join the two data sources. We will perform a simple linear regression to relate weather and other information to bicycle counts, in order to estimate how a change in any one of these parameters affects the number of riders on a given day.

In particular, this is an example of how the tools of Scikit-Learn can be used in a statistical modeling framework, in which the parameters of the model are assumed to have interpretable meaning. As discussed previously, this is not a standard approach within machine learning, but such interpretation is possible for some models.