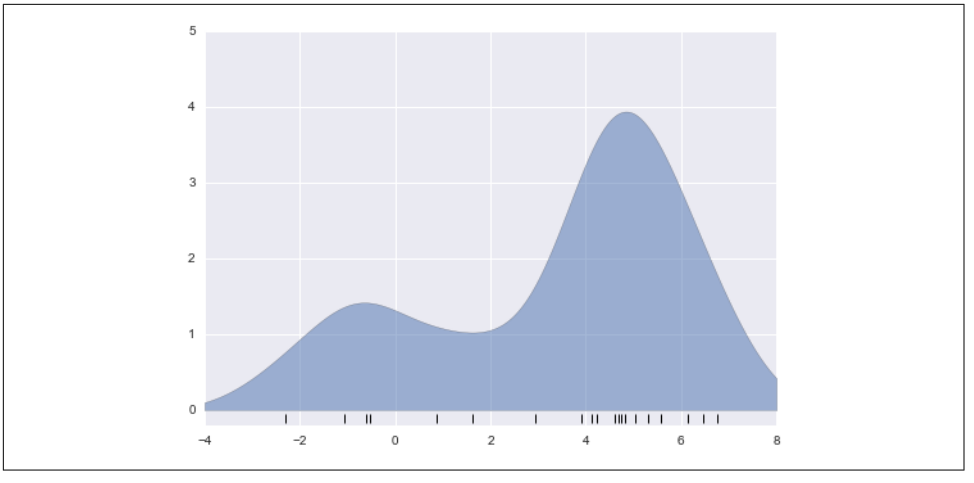

Figure 5-144. A kernel density estimate with a Gaussian kernel

This smoothed-out plot, with a Gaussian distribution contributed at the location of each input point, gives a much more accurate idea of the shape of the data distribution, and one that has much less variance (i.e., changes much less in response to differences in sampling).

These last two plots are examples of kernel density estimation in one dimension: the first uses a so-called "tophat" kernel and the second uses a Gaussian kernel. We'll now look at kernel density estimation in more detail.

Kernel Density Estimation in Practice

The free parameters of kernel density estimation are the *kernel*, which specifies the shape of the distribution placed at each point, and the *kernel bandwidth*, which controls the size of the kernel at each point. In practice, there are many kernels you might use for a kernel density estimation: in particular, the Scikit-Learn KDE implementation supports one of six kernels, which you can read about in Scikit-Learn's Density Estimation documentation.

While there are several versions of kernel density estimation implemented in Python (notably in the SciPy and StatsModels packages), I prefer to use Scikit-Learn's version because of its efficiency and flexibility. It is implemented in the `sklearn.neighbors.KernelDensity` estimator, which handles KDE in multiple dimensions with one of six kernels and one of a couple dozen distance metrics. Because KDE can be fairly computationally intensive, the Scikit-Learn estimator uses a tree-based algorithm under the hood and can trade off computation time for accuracy using the `atol` (absolute tolerance) and `rtol` (relative tolerance) parameters. We can determine the

kernel bandwidth, which is a free parameter, using Scikit-Learn's standard cross-validation tools, as we will soon see.

Let's first see a simple example of replicating the preceding plot using the Scikit-Learn `KernelDensity` estimator (Figure 5-145):

```
In[10]: from sklearn.neighbors import KernelDensity

        # instantiate and fit the KDE model
        kde = KernelDensity(bandwidth=1.0, kernel='gaussian')
        kde.fit(x[:, None])

        # score_samples returns the log of the probability density
        logprob = kde.score_samples(x_d[:, None])

        plt.fill_between(x_d, np.exp(logprob), alpha=0.5)
        plt.plot(x, np.full_like(x, -0.01), '|k', markeredgewidth=1)
        plt.ylim(-0.02, 0.22)

Out[10]: (-0.02, 0.22)
```

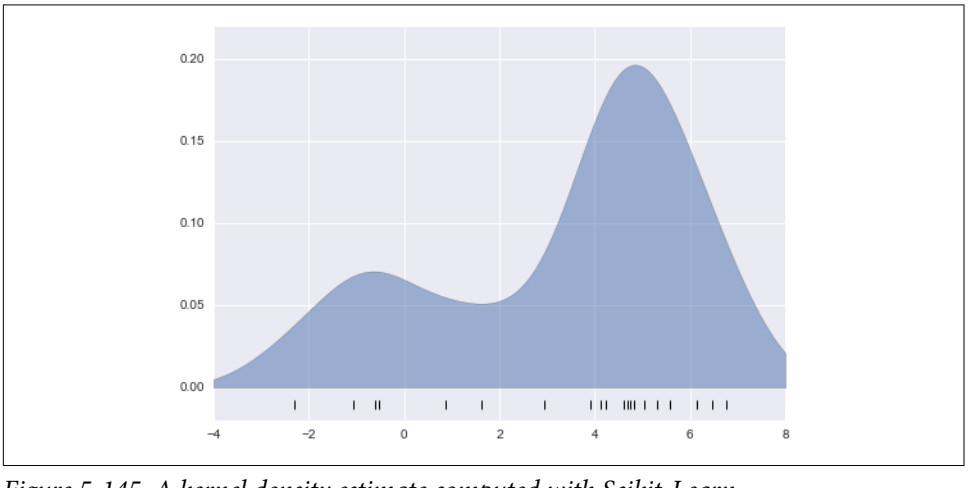

Figure 5-145. A kernel density estimate computed with Scikit-Learn

The result here is normalized such that the area under the curve is equal to 1.

Selecting the bandwidth via cross-validation

The choice of bandwidth within KDE is extremely important to finding a suitable density estimate, and is the knob that controls the bias–variance trade-off in the estimate of density: too narrow a bandwidth leads to a high-variance estimate (i.e., overfitting), where the presence or absence of a single point makes a large difference. Too wide a bandwidth leads to a high-bias estimate (i.e., underfitting) where the structure in the data is washed out by the wide kernel.

There is a long history in statistics of methods to quickly estimate the best bandwidth based on rather stringent assumptions about the data: if you look up the KDE implementations in the SciPy and StatsModels packages, for example, you will see implementations based on some of these rules.

In machine learning contexts, we've seen that such hyperparameter tuning often is done empirically via a cross-validation approach. With this in mind, the `KernelDensity` estimator in Scikit-Learn is designed such that it can be used directly within Scikit-Learn's standard grid search tools. Here we will use `GridSearchCV` to optimize the bandwidth for the preceding dataset. Because we are looking at such a small dataset, we will use leave-one-out cross-validation, which minimizes the reduction in training set size for each cross-validation trial:

```
In[11]: from sklearn.grid_search import GridSearchCV
        from sklearn.cross_validation import LeaveOneOut

        bandwidths = 10 ** np.linspace(-1, 1, 100)
        grid = GridSearchCV(KernelDensity(kernel='gaussian'),
                            {'bandwidth': bandwidths},
                            cv=LeaveOneOut(len(x)))
        grid.fit(x[:, None]);
```

Now we can find the choice of bandwidth that maximizes the score (which in this case defaults to the log-likelihood):

```
In[12]: grid.best_params_

Out[12]: {'bandwidth': 1.1233240329780276}
```

The optimal bandwidth happens to be very close to what we used in the example plot earlier, where the bandwidth was 1.0 (i.e., the default width of `scipy.stats.norm`).

Example: KDE on a Sphere

Perhaps the most common use of KDE is in graphically representing distributions of points. For example, in the Seaborn visualization library (discussed earlier in "Visualization with Seaborn" on page 311), KDE is built in and automatically used to help visualize points in one and two dimensions.

Here we will look at a slightly more sophisticated use of KDE for visualization of distributions. We will make use of some geographic data that can be loaded with Scikit-Learn: the geographic distributions of recorded observations of two South American mammals, *Bradypus variegatus* (the brown-throated sloth) and *Microryzomys minutus* (the forest small rice rat).

With Scikit-Learn, we can fetch this data as follows:

```
In[13]: from sklearn.datasets import fetch_species_distributions

        data = fetch_species_distributions()
```