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()
        # Get matrices/arrays of species IDs and locations
        latlon = np.vstack([data.train['dd lat'],
                            data.train['dd long']]).T
        species = np.array([d.decode('ascii').startswith('micro')
                            for d in data.train['species']], dtype='int')
```

With this data loaded, we can use the Basemap toolkit (mentioned previously in "Geographic Data with Basemap" on page 298) to plot the observed locations of these two species on the map of South America (Figure 5-146):

```
In[14]: from mpl_toolkits.basemap import Basemap
        from sklearn.datasets.species_distributions import construct_grids

        xgrid, ygrid = construct_grids(data)

        # plot coastlines with Basemap
        m = Basemap(projection='cyl', resolution='c',
                    llcrnrlat=ygrid.min(), urcrnrlat=ygrid.max(),
                    llcrnrlon=xgrid.min(), urcrnrlon=xgrid.max())
        m.drawmapboundary(fill_color='#DDEEFF')
        m.fillcontinents(color='#FFEEDD')
        m.drawcoastlines(color='gray', zorder=2)
        m.drawcountries(color='gray', zorder=2)

        # plot locations
        m.scatter(latlon[:, 1], latlon[:, 0], zorder=3,
                  c=species, cmap='rainbow', latlon=True);
```

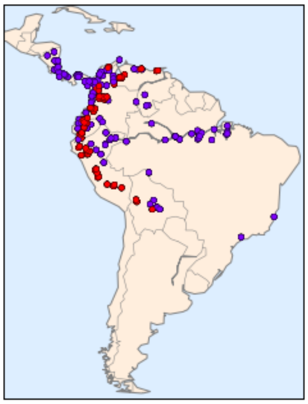

Figure 5-146. Location of species in training data

Unfortunately, this doesn't give a very good idea of the density of the species, because points in the species range may overlap one another. You may not realize it by looking at this plot, but there are over 1,600 points shown here!

Let's use kernel density estimation to show this distribution in a more interpretable way: as a smooth indication of density on the map. Because the coordinate system here lies on a spherical surface rather than a flat plane, we will use the `haversine` distance metric, which will correctly represent distances on a curved surface.

There is a bit of boilerplate code here (one of the disadvantages of the Basemap toolkit), but the meaning of each code block should be clear (Figure 5-147):

```
In[15]:
# Set up the data grid for the contour plot
X, Y = np.meshgrid(xgrid[::5], ygrid[::5][::-1])
land_reference = data.coverages[6][::5, ::5]
land_mask = (land_reference > -9999).ravel()
xy = np.vstack([Y.ravel(), X.ravel()]).T
xy = np.radians(xy[land_mask])

# Create two side-by-side plots
fig, ax = plt.subplots(1, 2)
fig.subplots_adjust(left=0.05, right=0.95, wspace=0.05)
species_names = ['Bradypus Variegatus', 'Microryzomys Minutus']
cmaps = ['Purples', 'Reds']

for i, axi in enumerate(ax):
    axi.set_title(species_names[i])

    # plot coastlines with Basemap
    m = Basemap(projection='cyl', llcrnrlat=Y.min(),
                urcrnrlat=Y.max(), llcrnrlon=X.min(),
                urcrnrlon=X.max(), resolution='c', ax=axi)
    m.drawmapboundary(fill_color='#DDEEFF')
    m.drawcoastlines()
    m.drawcountries()

    # construct a spherical kernel density estimate of the distribution
    kde = KernelDensity(bandwidth=0.03, metric='haversine')
    kde.fit(np.radians(latlon[species == i]))

    # evaluate only on the land: -9999 indicates ocean
    Z = np.full(land_mask.shape[0], -9999.0)
    Z[land_mask] = np.exp(kde.score_samples(xy))
    Z = Z.reshape(X.shape)

    # plot contours of the density
    levels = np.linspace(0, Z.max(), 25)
    axi.contourf(X, Y, Z, levels=levels, cmap=cmaps[i])
```

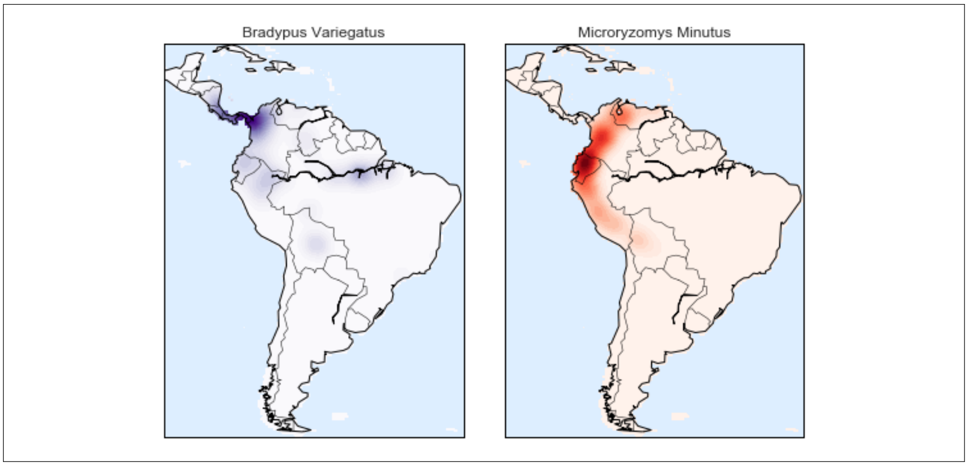

Figure 5-147. A kernel density representation of the species distributions

Compared to the simple scatter plot we initially used, this visualization paints a much clearer picture of the geographical distribution of observations of these two species.

Example: Not-So-Naive Bayes

This example looks at Bayesian generative classification with KDE, and demonstrates how to use the Scikit-Learn architecture to create a custom estimator.

In "In Depth: Naive Bayes Classification" on page 382, we took a look at naive Bayesian classification, in which we created a simple generative model for each class, and used these models to build a fast classifier. For naive Bayes, the generative model is a simple axis-aligned Gaussian. With a density estimation algorithm like KDE, we can remove the "naive" element and perform the same classification with a more sophisticated generative model for each class. It's still Bayesian classification, but it's no longer naive.

The general approach for generative classification is this:

1. Split the training data by label.
2. For each set, fit a KDE to obtain a generative model of the data. This allows you for any observation x and label y to compute a likelihood $P(x \mid y)$.
3. From the number of examples of each class in the training set, compute the *class prior*, $P(y)$.
4. For an unknown point x, the posterior probability for each class is $P(y \mid x) \propto P(x \mid y)P(y)$. The class that maximizes this posterior is the label assigned to the point.