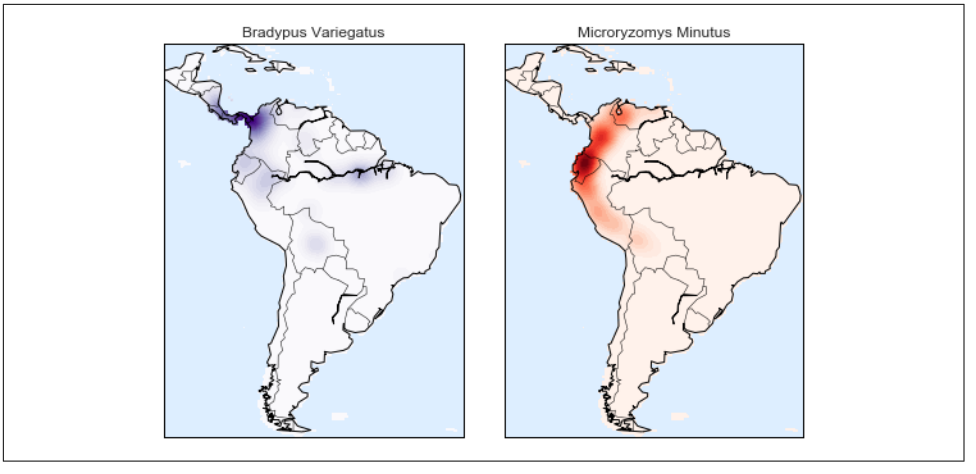

Figure 5-147. A kernel density representation of the species distributions

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.

The algorithm is straightforward and intuitive to understand; the more difficult piece is couching it within the Scikit-Learn framework in order to make use of the grid search and cross-validation architecture.

This is the code that implements the algorithm within the Scikit-Learn framework; we will step through it following the code block:

```
In[16]: from sklearn.base import BaseEstimator, ClassifierMixin


        class KDEClassifier(BaseEstimator, ClassifierMixin):
            """Bayesian generative classification based on KDE

            Parameters
            ----------
            bandwidth : float
                the kernel bandwidth within each class
            kernel : str
                the kernel name, passed to KernelDensity
            """
            def __init__(self, bandwidth=1.0, kernel='gaussian'):
                self.bandwidth = bandwidth
                self.kernel = kernel

            def fit(self, X, y):
                self.classes_ = np.sort(np.unique(y))
                training_sets = [X[y == yi] for yi in self.classes_]
                self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                              kernel=self.kernel).fit(Xi)
                                for Xi in training_sets]
                self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                                   for Xi in training_sets]
                return self

            def predict_proba(self, X):
                logprobs = np.array([model.score_samples(X)
                                     for model in self.models_]).T
                result = np.exp(logprobs + self.logpriors_)
                return result / result.sum(1, keepdims=True)

            def predict(self, X):
                return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

The anatomy of a custom estimator

Let's step through this code and discuss the essential features:

```
from sklearn.base import BaseEstimator, ClassifierMixin

class KDEClassifier(BaseEstimator, ClassifierMixin):
    """Bayesian generative classification based on KDE
```

```
    Parameters
    ----------
    bandwidth : float
        the kernel bandwidth within each class
    kernel : str
        the kernel name, passed to KernelDensity
    """
```

Each estimator in Scikit-Learn is a class, and it is most convenient for this class to inherit from the `BaseEstimator` class as well as the appropriate mixin, which provides standard functionality. For example, among other things, here the `BaseEstimator` contains the logic necessary to clone/copy an estimator for use in a cross-validation procedure, and `ClassifierMixin` defines a default `score()` method used by such routines. We also provide a docstring, which will be captured by IPython's help functionality (see "Help and Documentation in IPython" on page 3).

Next comes the class initialization method:

```
    def __init__(self, bandwidth=1.0, kernel='gaussian'):
        self.bandwidth = bandwidth
        self.kernel = kernel
```

This is the actual code executed when the object is instantiated with `KDEClassifier()`. In Scikit-Learn, it is important that *initialization contains no operations* other than assigning the passed values by name to `self`. This is due to the logic contained in `BaseEstimator` required for cloning and modifying estimators for cross-validation, grid search, and other functions. Similarly, all arguments to `__init__` should be explicit; that is, `*args` or `**kwargs` should be avoided, as they will not be correctly handled within cross-validation routines.

Next comes the `fit()` method, where we handle training data:

```
    def fit(self, X, y):
        self.classes_ = np.sort(np.unique(y))
        training_sets = [X[y == yi] for yi in self.classes_]
        self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                      kernel=self.kernel).fit(Xi)
                        for Xi in training_sets]
        self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                           for Xi in training_sets]
        return self
```

Here we find the unique classes in the training data, train a `KernelDensity` model for each class, and compute the class priors based on the number of input samples. Finally, `fit()` should always return `self` so that we can chain commands. For example:

```
label = model.fit(X, y).predict(X)
```

Notice that each persistent result of the fit is stored with a trailing underscore (e.g., `self.logpriors_`). This is a convention used in Scikit-Learn so that you can quickly scan the members of an estimator (using IPython's tab completion) and see exactly which members are fit to training data.

Finally, we have the logic for predicting labels on new data:

```
    def predict_proba(self, X):
        logprobs = np.vstack([model.score_samples(X)
                              for model in self.models_]).T
        result = np.exp(logprobs + self.logpriors_)
        return result / result.sum(1, keepdims=True)

    def predict(self, X):
        return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

Because this is a probabilistic classifier, we first implement `predict_proba()`, which returns an array of class probabilities of shape `[n_samples, n_classes]`. Entry `[i, j]` of this array is the posterior probability that sample `i` is a member of class `j`, computed by multiplying the likelihood by the class prior and normalizing.

Finally, the `predict()` method uses these probabilities and simply returns the class with the largest probability.

Using our custom estimator

Let's try this custom estimator on a problem we have seen before: the classification of handwritten digits. Here we will load the digits, and compute the cross-validation score for a range of candidate bandwidths using the `GridSearchCV` meta-estimator (refer back to "Hyperparameters and Model Validation" on page 359 for more information on this):

```
In[17]: from sklearn.datasets import load_digits
        from sklearn.grid_search import GridSearchCV

        digits = load_digits()

        bandwidths = 10 ** np.linspace(0, 2, 100)
        grid = GridSearchCV(KDEClassifier(), {'bandwidth': bandwidths})
        grid.fit(digits.data, digits.target)

        scores = [val.mean_validation_score for val in grid.grid_scores_]
```

Next we can plot the cross-validation score as a function of bandwidth (Figure 5-148):

```
In[18]: plt.semilogx(bandwidths, scores)
        plt.xlabel('bandwidth')
        plt.ylabel('accuracy')
        plt.title('KDE Model Performance')
```

```
        print(grid.best_params_)
        print('accuracy =', grid.best_score_)
```

```
{'bandwidth': 7.0548023107186433}
accuracy = 0.966611018364
```

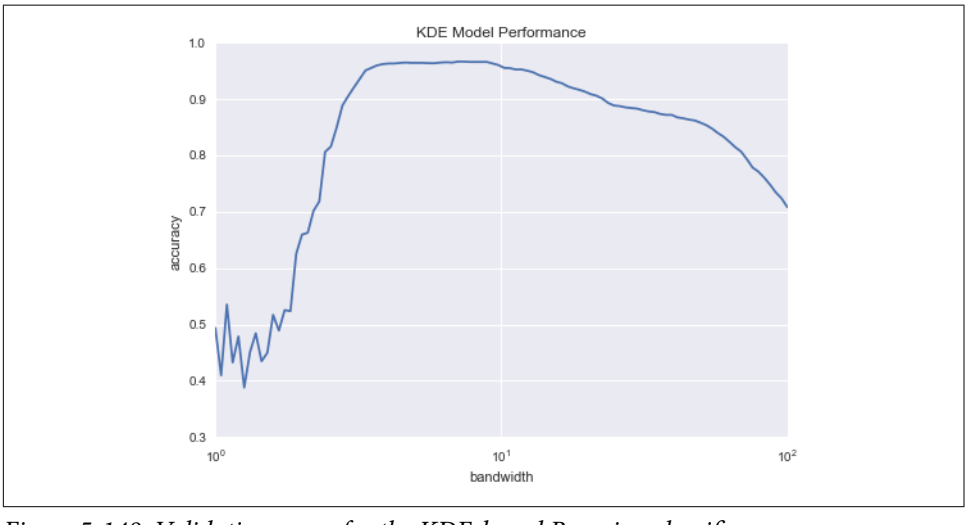

Figure 5-148. Validation curve for the KDE-based Bayesian classifier

We see that this not-so-naive Bayesian classifier reaches a cross-validation accuracy of just over 96%; this is compared to around 80% for the naive Bayesian classification:

```
In[19]: from sklearn.naive_bayes import GaussianNB
        from sklearn.cross_validation import cross_val_score
        cross_val_score(GaussianNB(), digits.data, digits.target).mean()
```

```
Out[19]: 0.81860038035501381
```

One benefit of such a generative classifier is interpretability of results: for each unknown sample, we not only get a probabilistic classification, but a *full model* of the distribution of points we are comparing it to! If desired, this offers an intuitive window into the reasons for a particular classification that algorithms like SVMs and random forests tend to obscure.

If you would like to take this further, there are some improvements that could be made to our KDE classifier model:

- We could allow the bandwidth in each class to vary independently.
- We could optimize these bandwidths not based on their prediction score, but on the likelihood of the training data under the generative model within each class (i.e., use the scores from `KernelDensity` itself rather than the global prediction accuracy).

Finally, if you want some practice building your own estimator, you might tackle building a similar Bayesian classifier using Gaussian mixture models instead of KDE.

Application: A Face Detection Pipeline

This chapter has explored a number of the central concepts and algorithms of machine learning. But moving from these concepts to real-world application can be a challenge. Real-world datasets are noisy and heterogeneous, may have missing features, and may include data in a form that is difficult to map to a clean `[n_samples, n_features]` matrix. Before applying any of the methods discussed here, you must first extract these features from your data; there is no formula for how to do this that applies across all domains, and thus this is where you as a data scientist must exercise your own intuition and expertise.

One interesting and compelling application of machine learning is to images, and we have already seen a few examples of this where pixel-level features are used for classification. In the real world, data is rarely so uniform and simple pixels will not be suitable, a fact that has led to a large literature on *feature extraction* methods for image data (see "Feature Engineering" on page 375).

In this section, we will take a look at one such feature extraction technique, the Histogram of Oriented Gradients (HOG), which transforms image pixels into a vector representation that is sensitive to broadly informative image features regardless of confounding factors like illumination. We will use these features to develop a simple face detection pipeline, using machine learning algorithms and concepts we've seen throughout this chapter. We begin with the standard imports:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

HOG Features

The Histogram of Gradients is a straightforward feature extraction procedure that was developed in the context of identifying pedestrians within images. HOG involves the following steps:

1. Optionally prenormalize images. This leads to features that resist dependence on variations in illumination.
2. Convolve the image with two filters that are sensitive to horizontal and vertical brightness gradients. These capture edge, contour, and texture information.
3. Subdivide the image into cells of a predetermined size, and compute a histogram of the gradient orientations within each cell.