cles, so as to account for noncircular clusters. It turns out these are two essential components of a different type of clustering model, Gaussian mixture models.

Generalizing E–M: Gaussian Mixture Models

A Gaussian mixture model (GMM) attempts to find a mixture of multidimensional Gaussian probability distributions that best model any input dataset. In the simplest case, GMMs can be used for finding clusters in the same manner as *k*-means (Figure 5-127):

```
In[7]: from sklearn.mixture import GMM
       gmm = GMM(n_components=4).fit(X)
       labels = gmm.predict(X)
       plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```

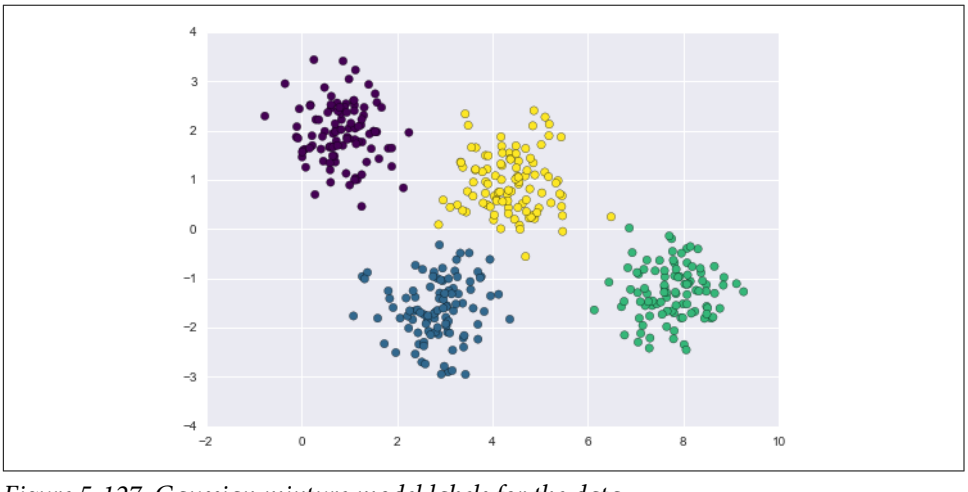

Figure 5-127. Gaussian mixture model labels for the data

But because GMM contains a probabilistic model under the hood, it is also possible to find probabilistic cluster assignments—in Scikit-Learn we do this using the `predict_proba` method. This returns a matrix of size `[n_samples, n_clusters]` that measures the probability that any point belongs to the given cluster:

```
In[8]: probs = gmm.predict_proba(X)
       print(probs[:5].round(3))
```

```
[[ 0.     0.     0.475  0.525]
 [ 0.     1.     0.     0.   ]
 [ 0.     1.     0.     0.   ]
 [ 0.     0.     0.     1.   ]
 [ 0.     1.     0.     0.   ]]
```

We can visualize this uncertainty by, for example, making the size of each point proportional to the certainty of its prediction; looking at Figure 5-128, we can see that it

is precisely the points at the boundaries between clusters that reflect this uncertainty of cluster assignment:

```
In[9]: size = 50 * probs.max(1) ** 2  # square emphasizes differences
       plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=size);
```

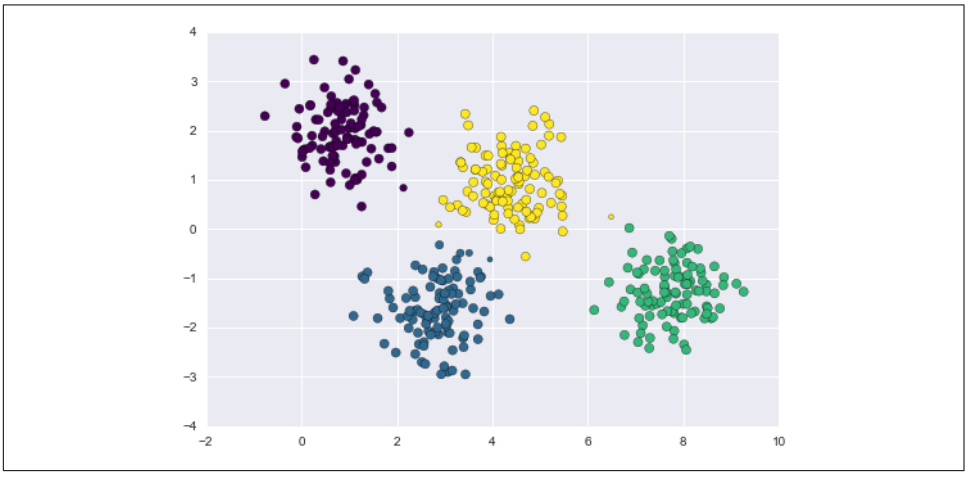

Figure 5-128. GMM probablistic labels: probabilities are shown by the size of points

Under the hood, a Gaussian mixture model is very similar to *k*-means: it uses an expectation–maximization approach that qualitatively does the following:

1. Choose starting guesses for the location and shape
2. Repeat until converged:
   a. *E-step*: for each point, find weights encoding the probability of membership in each cluster
   b. *M-step*: for each cluster, update its location, normalization, and shape based on *all* data points, making use of the weights

The result of this is that each cluster is associated not with a hard-edged sphere, but with a smooth Gaussian model. Just as in the *k*-means expectation–maximization approach, this algorithm can sometimes miss the globally optimal solution, and thus in practice multiple random initializations are used.

Let's create a function that will help us visualize the locations and shapes of the GMM clusters by drawing ellipses based on the `gmm` output:

```
In[10]:
from matplotlib.patches import Ellipse

def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Draw an ellipse with a given position and covariance"""
```

```
        ax = ax or plt.gca()

        # Convert covariance to principal axes
        if covariance.shape == (2, 2):
            U, s, Vt = np.linalg.svd(covariance)
            angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
            width, height = 2 * np.sqrt(s)
        else:
            angle = 0
            width, height = 2 * np.sqrt(covariance)

        # Draw the ellipse
        for nsig in range(1, 4):
            ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                                 angle, **kwargs))

    def plot_gmm(gmm, X, label=True, ax=None):
        ax = ax or plt.gca()
        labels = gmm.fit(X).predict(X)
        if label:
            ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)
        else:
            ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
        ax.axis('equal')

        w_factor = 0.2 / gmm.weights_.max()
        for pos, covar, w in zip(gmm.means_, gmm.covars_, gmm.weights_):
            draw_ellipse(pos, covar, alpha=w * w_factor)
```

With this in place, we can take a look at what the four-component GMM gives us for our initial data (Figure 5-129):

```
In[11]: gmm = GMM(n_components=4, random_state=42)
        plot_gmm(gmm, X)
```

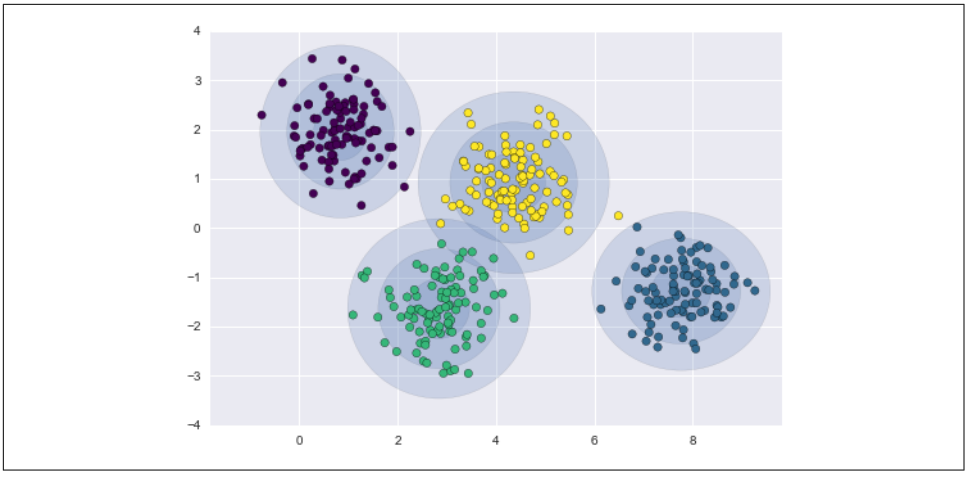

Figure 5-129. Representation of the four-component GMM in the presence of circular clusters

Similarly, we can use the GMM approach to fit our stretched dataset; allowing for a full covariance, the model will fit even very oblong, stretched-out clusters (Figure 5-130):

```
In[12]: gmm = GMM(n_components=4, covariance_type='full', random_state=42)
        plot_gmm(gmm, X_stretched)
```

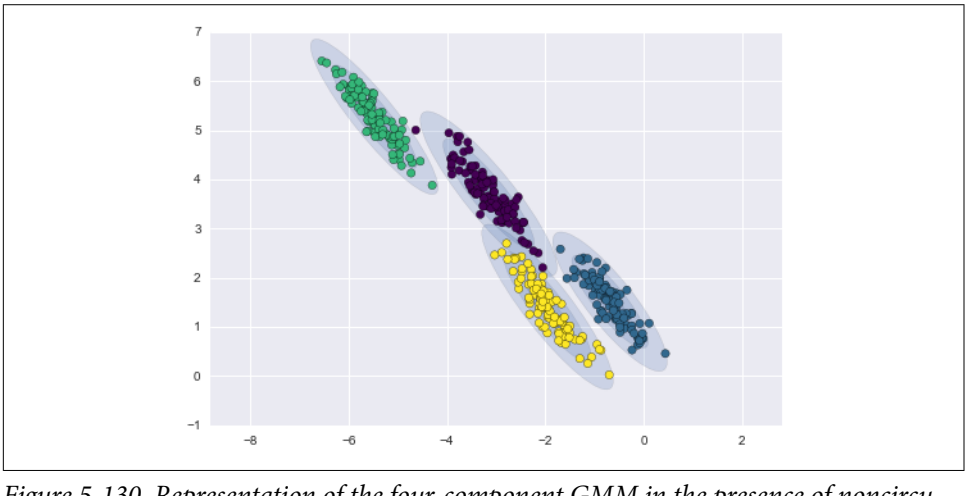

Figure 5-130. Representation of the four-component GMM in the presence of noncircular clusters

This makes clear that GMMs address the two main practical issues with *k*-means encountered before.

Choosing the covariance type

If you look at the details of the preceding fits, you will see that the `covariance_type` option was set differently within each. This hyperparameter controls the degrees of freedom in the shape of each cluster; it is essential to set this carefully for any given problem. The default is `covariance_type="diag"`, which means that the size of the cluster along each dimension can be set independently, with the resulting ellipse constrained to align with the axes. A slightly simpler and faster model is `covariance_type="spherical"`, which constrains the shape of the cluster such that all dimensions are equal. The resulting clustering will have similar characteristics to that of *k*-means, though it is not entirely equivalent. A more complicated and computationally expensive model (especially as the number of dimensions grows) is to use `covariance_type="full"`, which allows each cluster to be modeled as an ellipse with arbitrary orientation.

We can see a visual representation of these three choices for a single cluster within Figure 5-131:

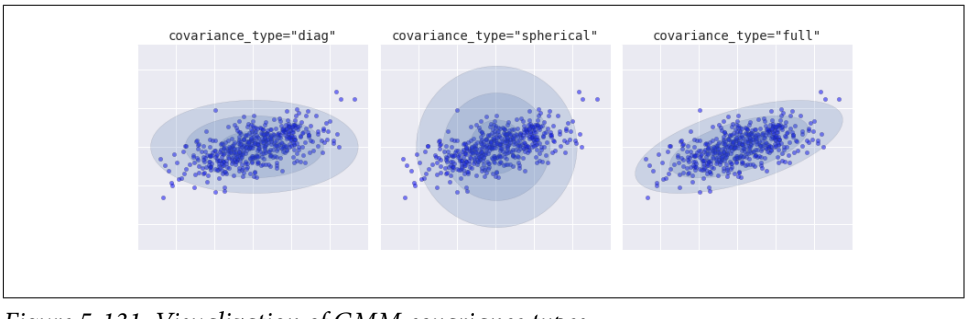

Figure 5-131. Visualization of GMM covariance types

GMM as Density Estimation

Though GMM is often categorized as a clustering algorithm, fundamentally it is an algorithm for *density estimation*. That is to say, the result of a GMM fit to some data is technically not a clustering model, but a generative probabilistic model describing the distribution of the data.

As an example, consider some data generated from Scikit-Learn's `make_moons` function (visualized in Figure 5-132), which we saw in "In Depth: k-Means Clustering" on page 462:

```
In[13]: from sklearn.datasets import make_moons
        Xmoon, ymoon = make_moons(200, noise=.05, random_state=0)
        plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);
```