Python for Machine Learning and Data Analysis

Dr. Handan Liu

h.liu@northeastern.edu

Northeastern University

Spring 2019

Machine Learning 06

Lecture 12

Content

1. k-Means Clustering
   - 1.1 Introducing k-Means
   - 1.2 k-Means Algorithm: Expectation–Maximization
   - 1.3 Caveats of expectation–maximization
   - 1.4 Examples 1&2
2. Gaussian Mixture Models
   - 2.1 Motivating GMM: Weaknesses of k-Means
   - 2.2 Generalizing E-M: Gaussian Mixture Models
   - 2.3 GMM as *Density Estimation*
   - 2.4 Example

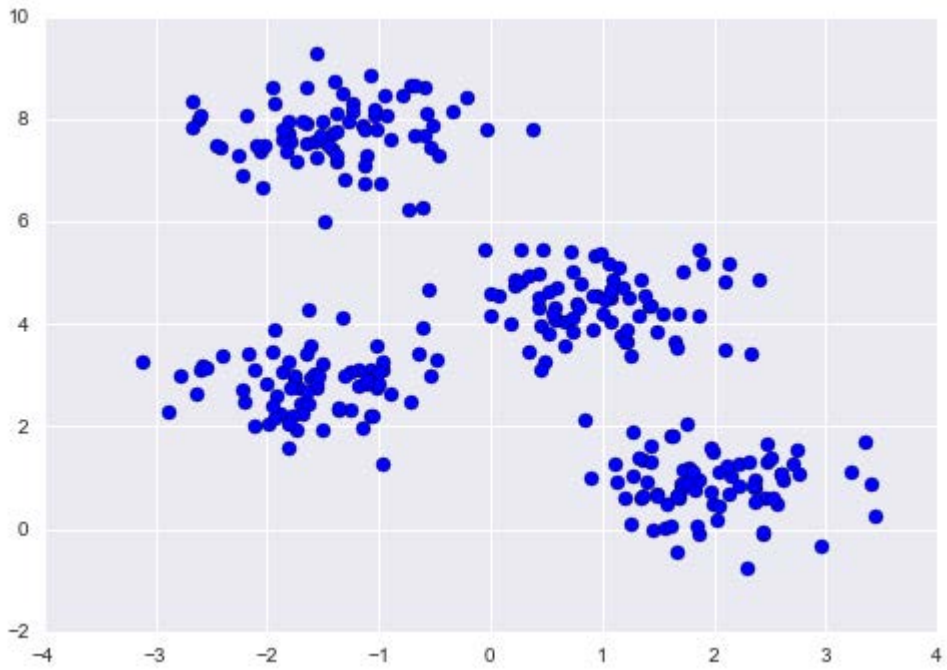
k-Means Clustering

1.1 Introducing k-Means

- The k-means algorithm searches for a pre-determined number of clusters within an unlabeled multidimensional dataset. It accomplishes this using a simple conception of what the optimal clustering looks like:
  - The "cluster center" is the arithmetic mean of all the points belonging to the cluster.
  - Each point is closer to its own cluster center than to other cluster centers.

- Those two assumptions are the basis of the k-means model.

First, generate a simple two-dimensional dataset containing four distinct blobs. To emphasize that this is an unsupervised algorithm, leave the labels out of the visualization.



Visualize the results by plotting the data colored by these labels. Also plot the cluster centers as determined by the k-means estimator:

- The number of possible combinations of cluster assignments is exponential in the number of data points — an exhaustive search would be very costly.
- Instead, the typical approach to k-means involves an iterative approach known as ***expectation-maximization***.

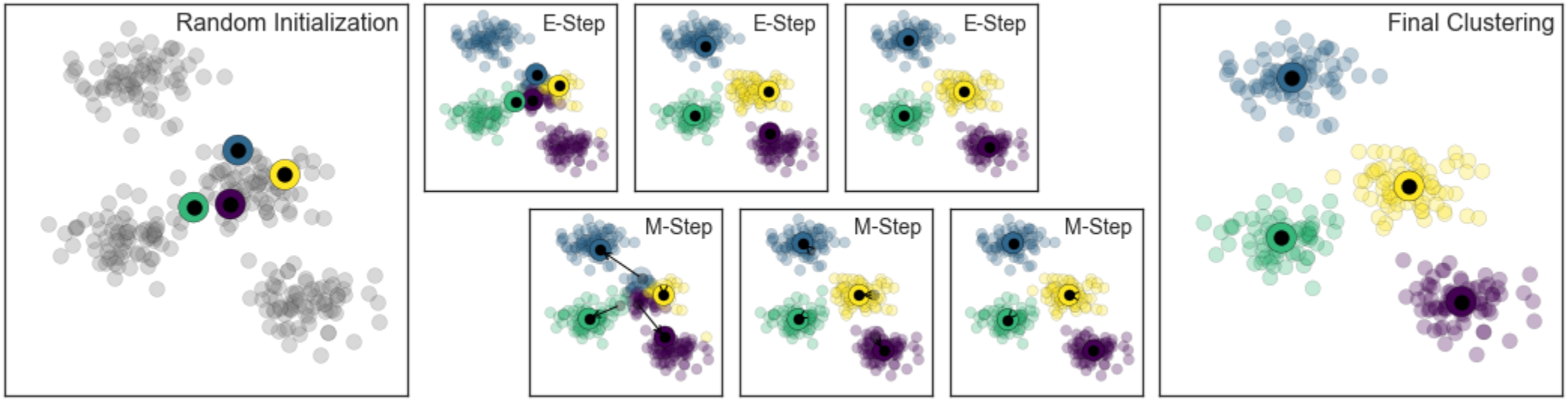
1.2 k-Means Algorithm: Expectation–Maximization

- Expectation-maximization (E–M) is a powerful algorithm that comes up in a variety of contexts within data science.
- *k*-means is a particularly simple and easy-to-understand application of the algorithm.
- In short, the expectation-maximization approach here consists of the following procedure:
    1. Guess some cluster centers
    2. Repeat until converged
        - E-Step: assign points to the nearest cluster center
        - M-Step: set the cluster centers to the mean

- The "E-step" or "Expectation step" is so-named because it involves updating our expectation of which cluster each point belongs to.
- The "M-step" or "Maximization step" is so-named because it involves maximizing some fitness function that defines the location of the cluster centers — in this case, that maximization is accomplished by taking a simple mean of the data in each cluster.

- The literature about this algorithm is vast, but can be summarized as follows:
    - under typical circumstances, each repetition of the E-step and M-step will always result in a better estimate of the cluster characteristics.

We can visualize the algorithm as shown in the following figure. For the particular initialization shown here, the clusters converge in just three iterations.



Most well-tested implementations will do a bit more than this, but the preceding function gives the gist of the expectation-maximization approach.

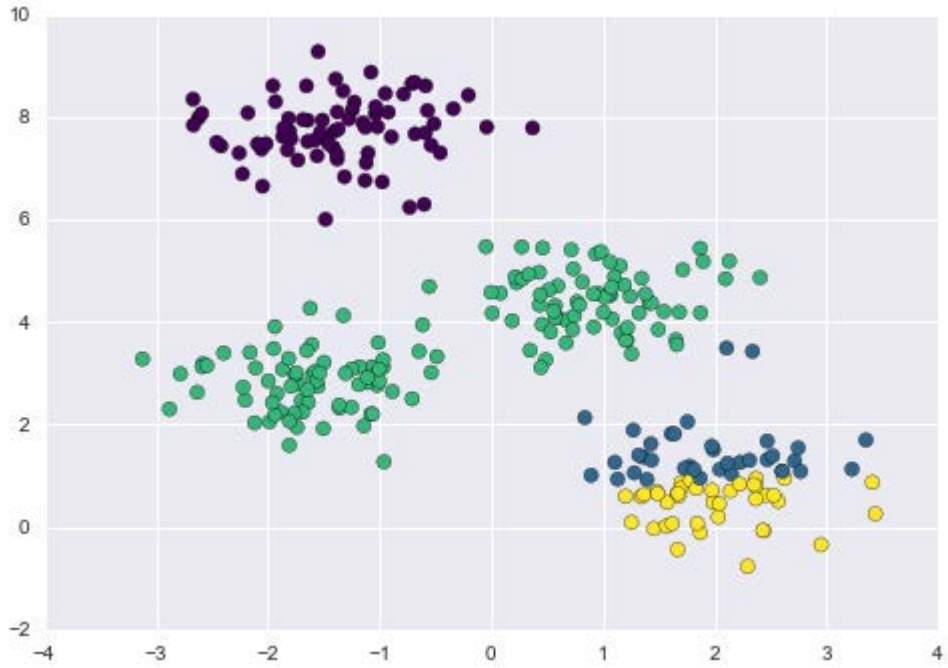
1.3 Caveats of expectation-maximization (E-M)

There are a few issues to be aware of when using the expectation–maximization algorithm.

(1) The globally optimal result may not be achieved

First, although the E-M procedure is guaranteed to improve the result in each step, there is no assurance that it will lead to the *global* best solution.

For example, if we use a different random seed in our simple procedure, the particular starting guesses lead to poor results:



Here the E-M approach has converged, but has not converged to a globally optimal configuration.

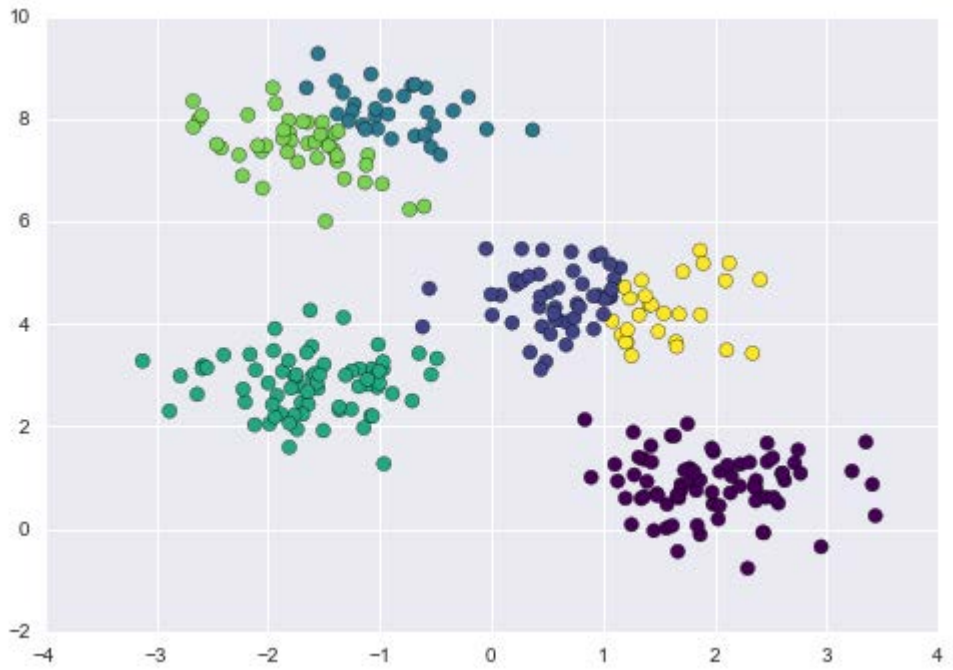
For this reason, it is common for the algorithm to be run for multiple starting guesses, as indeed Scikit-Learn does by default (set by the n_init parameter, which defaults to 10).

(2) The number of clusters must be selected beforehand

Another common challenge with k-means is that you must tell it how many clusters you expect: it cannot learn the number of clusters from the data. For example, if we ask the algorithm to identify six clusters, it will happily proceed and find the best six clusters:

Whether the result is meaningful is a question that is difficult to answer definitively; one approach that is rather intuitive is called silhouette analysis.

Alternatively, a more complicated clustering algorithm may be used, which has a better quantitative measure of the fitness per number of clusters (e.g., Gaussian mixture models) or which can be chosen a suitable number of clusters (e.g., DBSCAN, mean-shift, or affinity propagation, all available in the sklearn.cluster submodule)



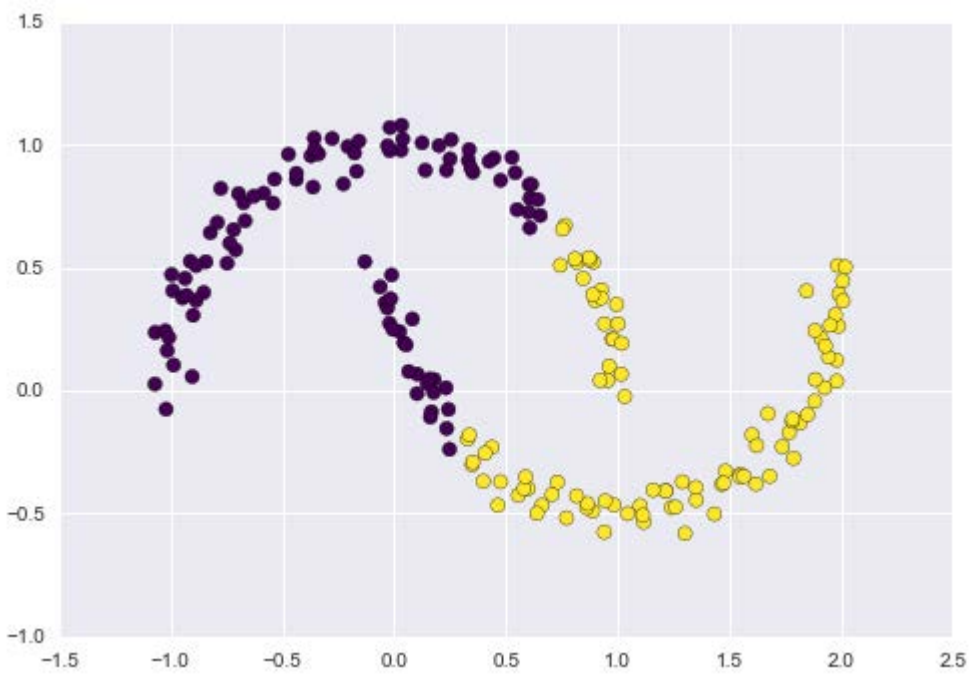
(3) k-means is limited to linear cluster boundaries

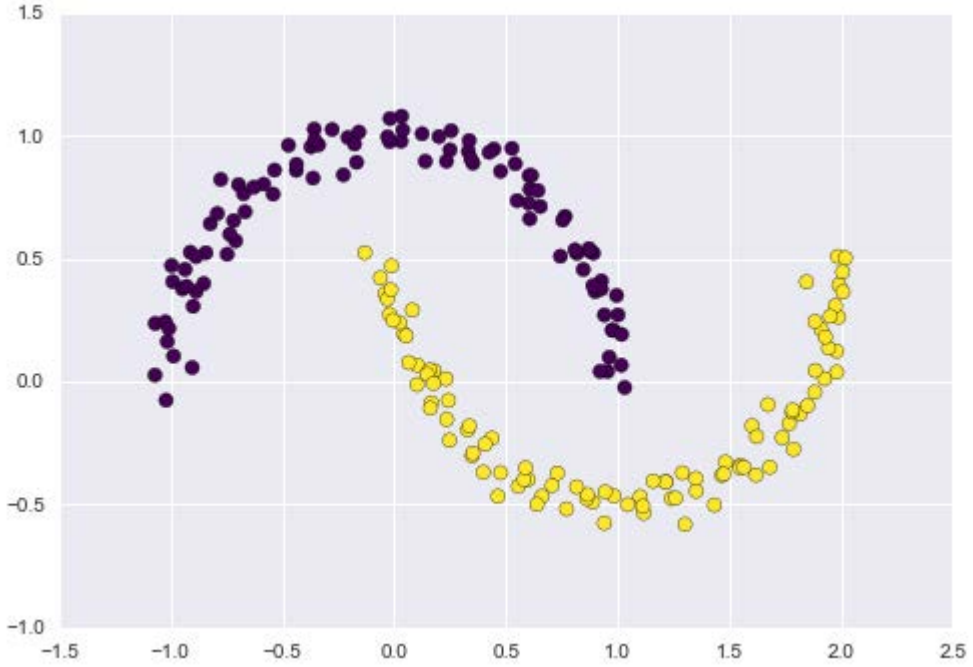
The fundamental model assumptions of k-means (points will be closer to their own cluster center than to others) means that the algorithm will often be ineffective if the clusters have complicated geometries.

In particular, the boundaries between k-means clusters will always be linear, which means that it will fail due to more complicated boundaries.

Similar that a kernel transformation to project the data into a higher dimension where a linear separation is possible, imagine to discover non-linear boundaries in k-means.

One version of this kernelized k-means is implemented in Scikit-Learn within the SpectralClustering estimator. It uses the graph of *nearest neighbors* to compute a higher-dimensional representation of the data, and then assigns labels using a k-means algorithm:





With this kernel transform approach, the kernelized *k*-means is able to find the more complicated nonlinear boundaries between clusters.

(4) k-means can be slow for large numbers of samples

Because each iteration of k-means must access every point in the dataset, the algorithm can be relatively slow as the number of samples grows.

You might wonder if this requirement to use all data at each iteration can be relaxed; for example, you might just use a subset of the data to update the cluster centers at each step.

This is the idea behind batch-based k-means algorithms, one form of which is implemented in sklearn.cluster.MiniBatchKMeans. The interface for this is the same as for standard Kmeans.

1.4 Examples

Example 1: k-means on digits

- Recall that the digits consist of 1,797 samples with 64 features, where each of the 64 features is the brightness of one pixel in an 8×8 image;
- Perform the clustering to get 10 clusters;
- Visualize what these cluster centers look like.

We see that even without the labels, KMeans is able to find clusters whose centers are recognizable digits, with perhaps the exception of 1 and 8.

Because k-means knows nothing about the identity of the cluster, the 0 – 9 labels may be permuted. We can fix this by matching each learned cluster label with the true labels found in them:

- As we might expect from the cluster centers we visualized before, the main point of confusion is between the eights ‘8’ and ones ‘1’. But this still shows that using k-means, we can essentially build a digit classifier without reference to any known labels!

- Then, use the t-distributed stochastic neighbor embedding (t-SNE) algorithm (mentioned in “Manifold Learning”) to pre-process the data before performing k-means. t-SNE is a nonlinear embedding algorithm that is particularly adept at preserving points within clusters. Let's see how it does:

Example 2: k-means for color compression

One interesting application of clustering is in color compression within images. For example, imagine you have an image with millions of colors. In most images, a large number of the colors will be unused, and many of the pixels in the image will have similar or even identical colors.

For example, consider the image shown in the figure, which is from the Scikit-Learn datasets module (for this to work, you'll have to have the pillow Python package installed).

The image itself is stored in a three-dimensional array of size (height, width, RGB), containing red/blue/green contributions as integers from 0 to 255.

One way we can view this set of pixels is as a cloud of points in a three-dimensional color space. We will reshape the data to [n_samples x n_features], and rescale the colors so that they lie between 0 and 1.

The result is a re-coloring of the original pixels, where each pixel is assigned the color of its closest cluster center. Plotting these new colors in the image space rather than the pixel space shows us the effect of this:

Some detail is certainly lost in the right panel, but the overall image is still easily recognizable. This image on the right achieves a compression factor of around 1 million!
While this is an interesting application of *k*-means, there are certainly better way to compress information in images. But the example shows the power of thinking outside of the box with unsupervised methods like *k*-means.

Gaussian Mixture Models

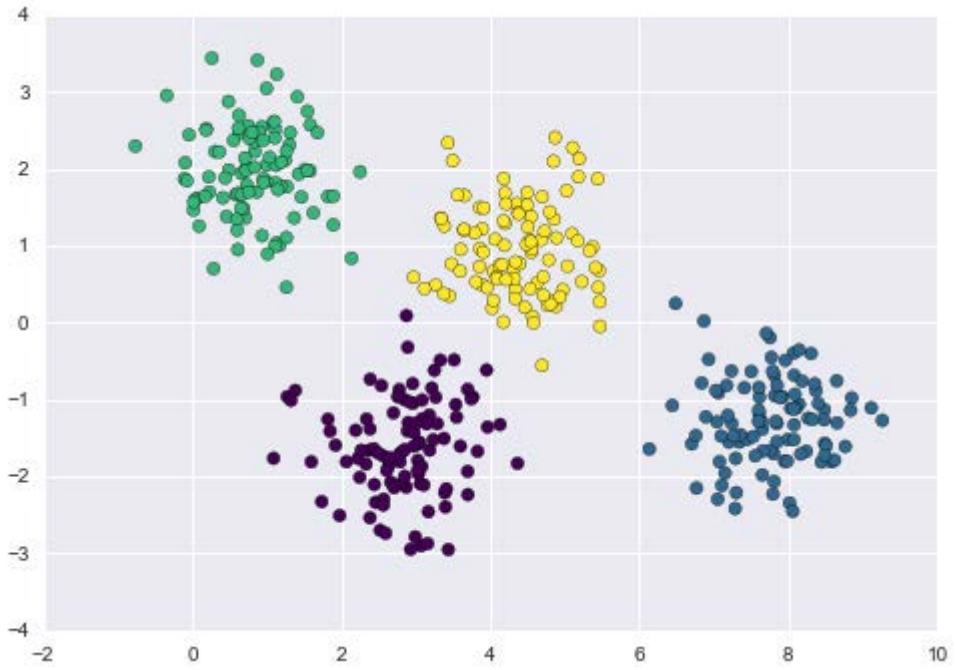
Gaussian Mixture Models

- The *k*-means clustering model is simple and relatively easy to understand, but its simplicity leads to practical challenges in its application.
- Especially, the non-probabilistic nature of *k*-means and its use of simple distance-from-cluster-center to assign cluster membership leads to poor performance for many real-world situations.
- Gaussian mixture models (GMMs) can be viewed as an extension of the ideas behind *k*-means, but can also be a powerful tool for estimation beyond simple clustering.

2.1 Motivating GMM: Weaknesses of k-Means

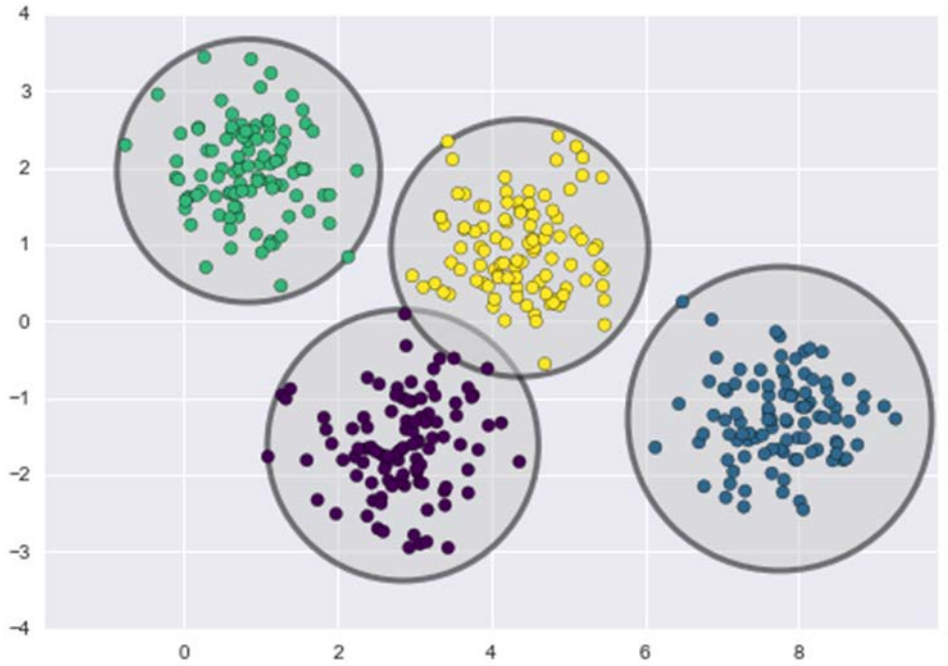
- In the previous section, given simple, well-separated data, *k*-means finds suitable clustering results.
- From an intuitive standpoint, expect that the clustering assignment for some points is more certain than others:
  - for example, there appears to be a very slight overlap between the two middle clusters, such that we might not have complete confidence in the cluster assignment of points between them.
- The k-means model has no intrinsic measure of probability or uncertainty of cluster assignments.

A simple, well-separated data



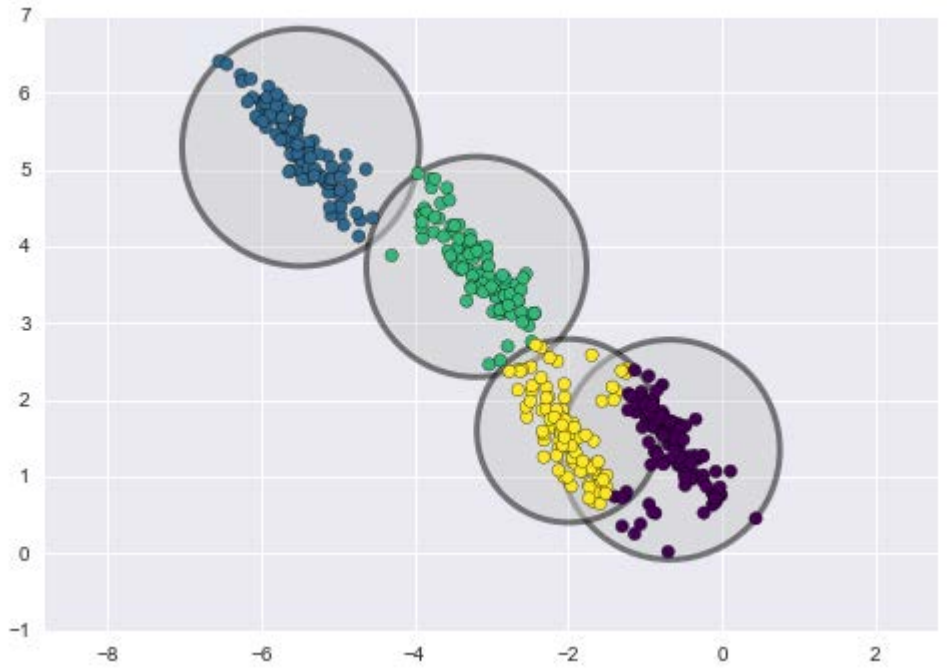
One way to think about the k-means model is that it places a circle (or, in higher dimensions, a hyper-sphere) at the center of each cluster, with a radius defined by the most distant point in the cluster. This radius acts as a hard cutoff for cluster assignment within the training set: any point outside this circle is not considered a member of the cluster.

Visualize this cluster model with the following function:



An important observation for k-means is that these cluster models *must be circular*: k-means has no built-in way of accounting for oblong or elliptical clusters.

So, for example, if we take the same data and transform it, the cluster assignments end up becoming muddled:



- By eye, we recognize that these transformed clusters are non-circular, and thus circular clusters would be a poor fit. Nevertheless, k-means is not flexible enough to account for this, and tries to force-fit the data into four circular clusters.
- This results in a mixing of cluster assignments where the resulting circles overlap: see especially the bottom-right of this plot.
- One might imagine addressing this particular situation by preprocessing the data with PCA (see “Principal Component Analysis”), but in practice there is no guarantee that such a global operation will circularize the individual data.

These two disadvantages of k-means:

- Its lack of flexibility in cluster shape and
- Its lack of probabilistic cluster assignment

For many datasets (especially low-dimensional datasets) the disadvantages may not perform as well as you might hope.

You might imagine addressing these weaknesses by generalizing the k-means model:

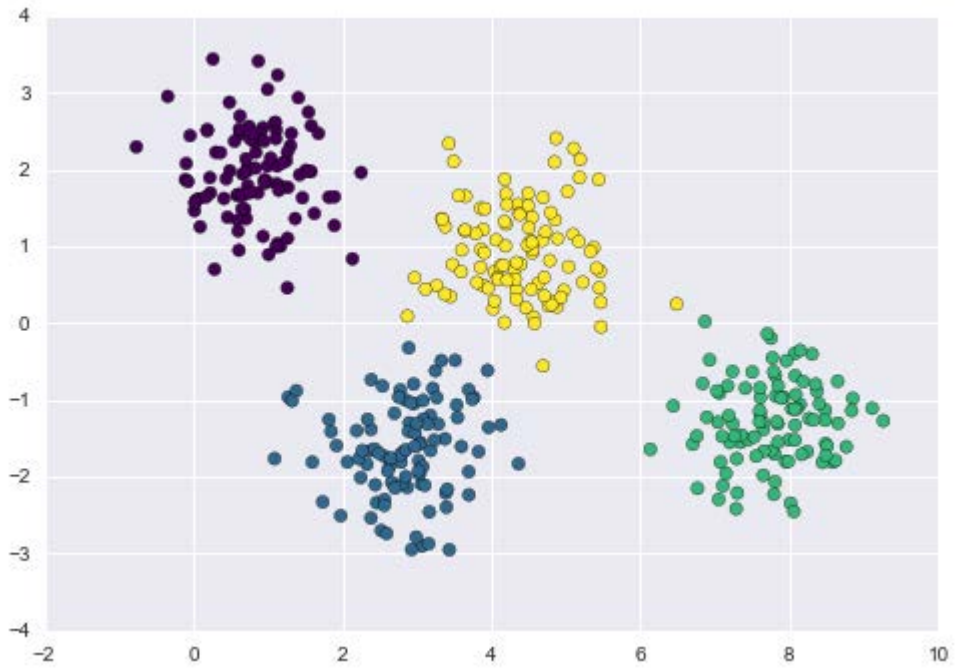
- For example, you could measure uncertainty in cluster assignment by comparing the distances of each point to all cluster centers, rather than focusing on just the closest.
- You might also imagine allowing the cluster boundaries to be ellipses rather than circles, so as to account for non-circular clusters.
- It turns out these are two essential components of a different type of clustering model, Gaussian mixture models.

2.2 Generalizing E-M: Gaussian Mixture Models

A Gaussian mixture model (GaussianMixture) attempts to find a mixture of multi-dimensional 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:

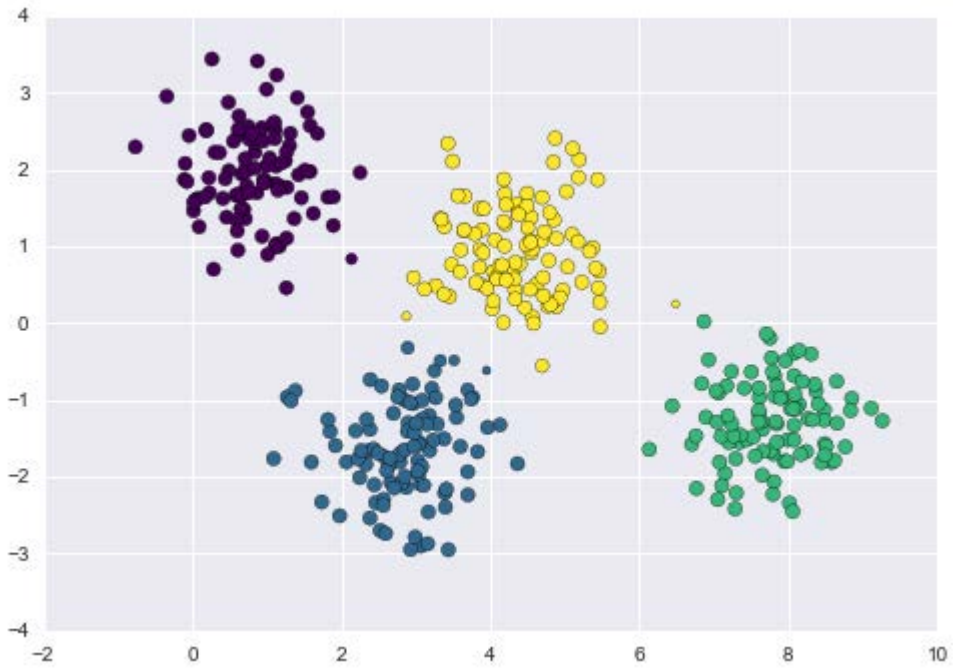
Because GaussianMixture contains a probabilistic model, it is also possible to find probabilistic cluster assignments — in Scikit-Learn this is done using the predict_proba method. This returns a matrix of size *[n_samples, n_clusters]* which measures the probability that any point belongs to the given cluster:

```
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.   ]]
```



Visualize this uncertainty by making the size of each point proportional to the certainty of its prediction;

Looking at this figure, it is precisely the points at the boundaries between clusters that reflect this uncertainty of cluster assignment.



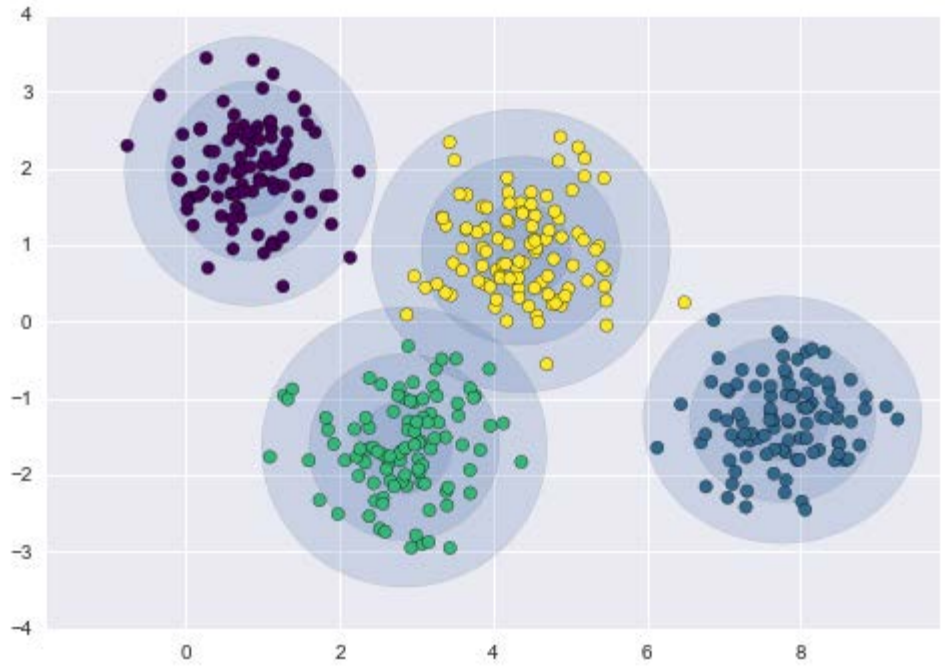
A Gaussian mixture model is very similar to k-means: it uses an expectation–maximization approach which qualitatively does the following:

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

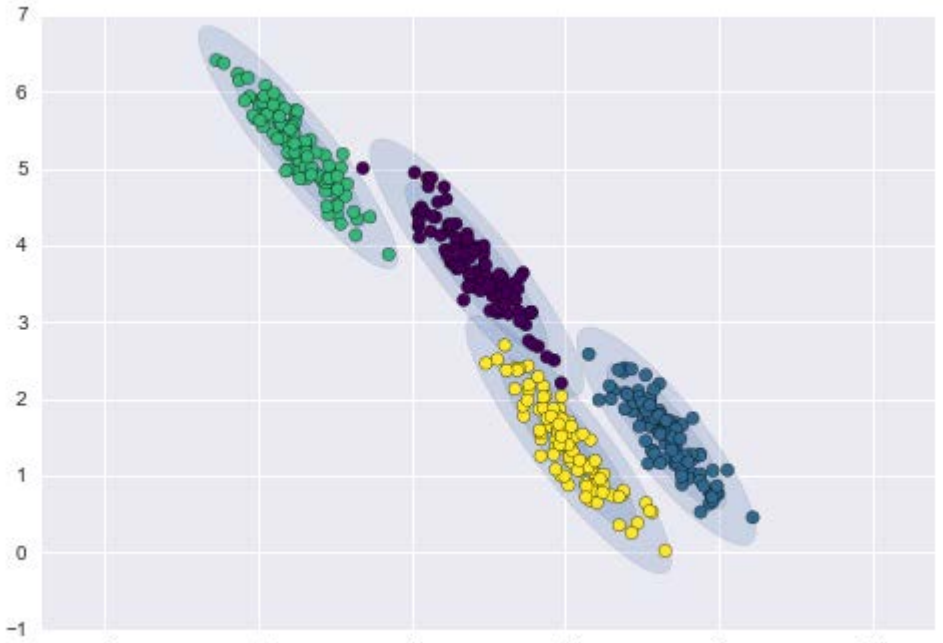
The result of this is that each cluster is associated not with a hard-edged sphere, but with a smooth Gaussian model. Just as in the k-means expectation–maximization approach, this algorithm can sometimes miss the globally optimal solution, and thus in practice multiple random initializations are used.

Visualize the locations and shapes of the GMM clusters by drawing ellipses based on the GMM output.

With this in place, take a look at what the four-component GMM gives us for our initial data:



Similarly, use the GMM approach to fit the stretched dataset; allowing for a full covariance the model will fit even very oblong, stretched-out clusters:



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

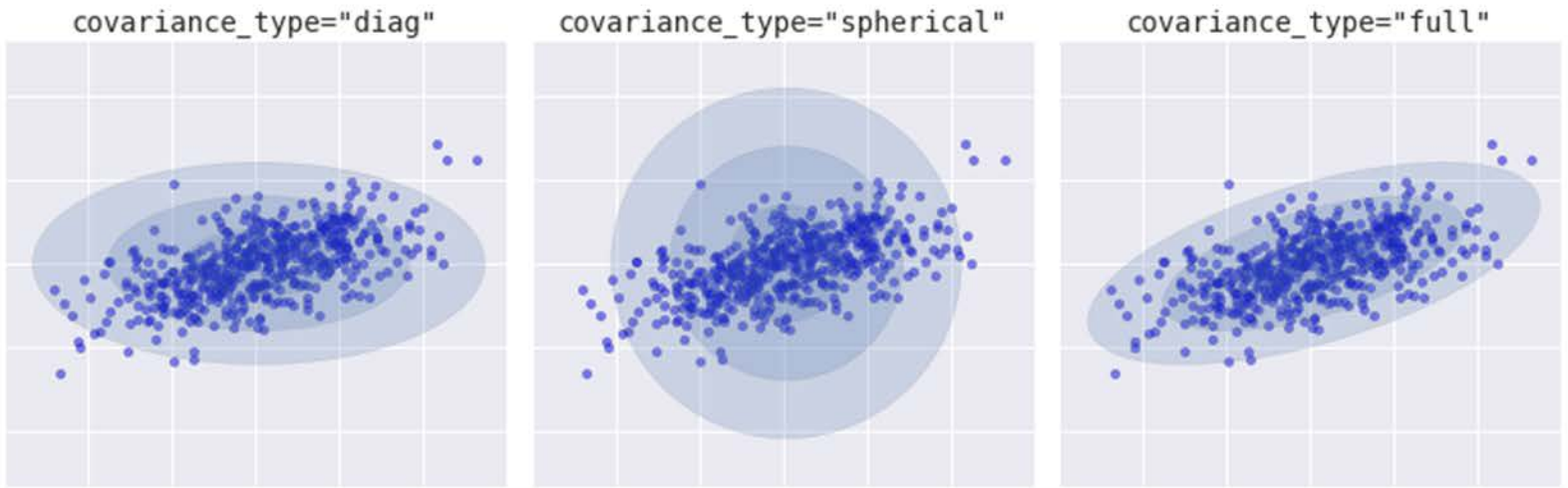
2.3 Choosing the covariance type

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.

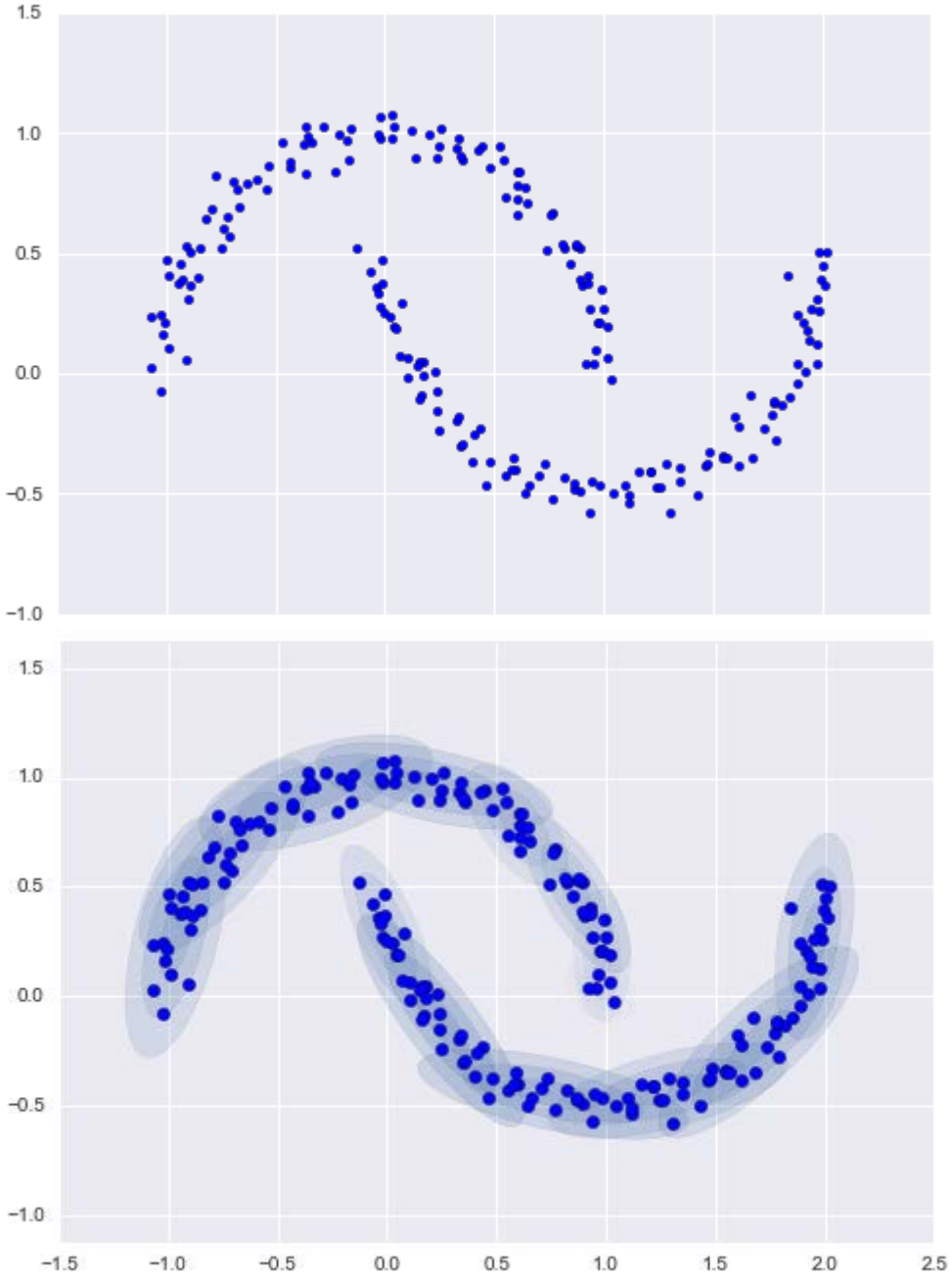
We can see a visual representation of these three choices for a single cluster within the following figure:



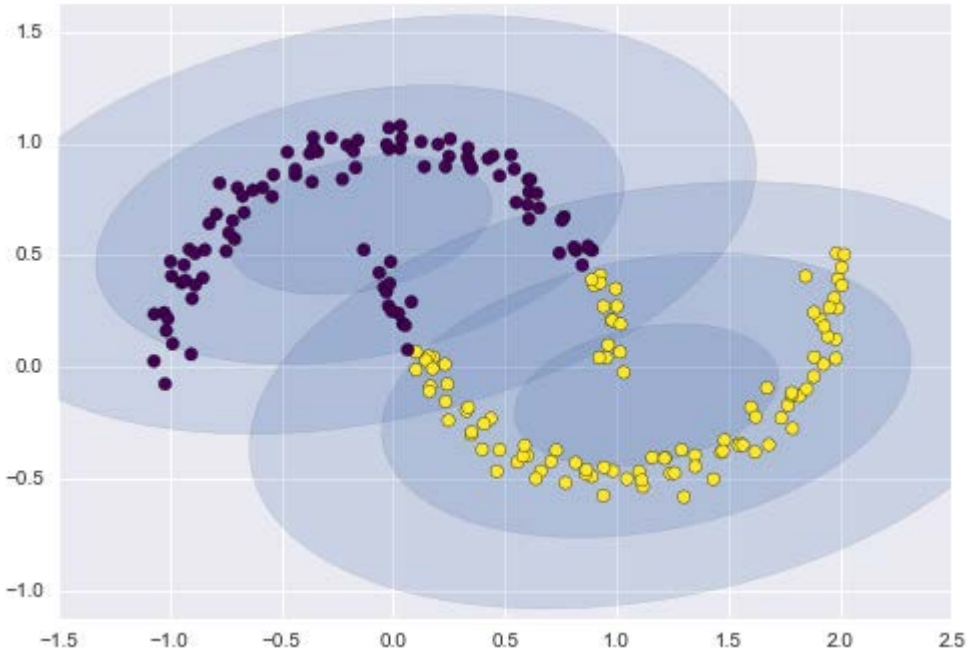
2.4 GMM as *Density Estimation*

- 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.

Consider some data generated from Scikit-Learn's *make_moons* function



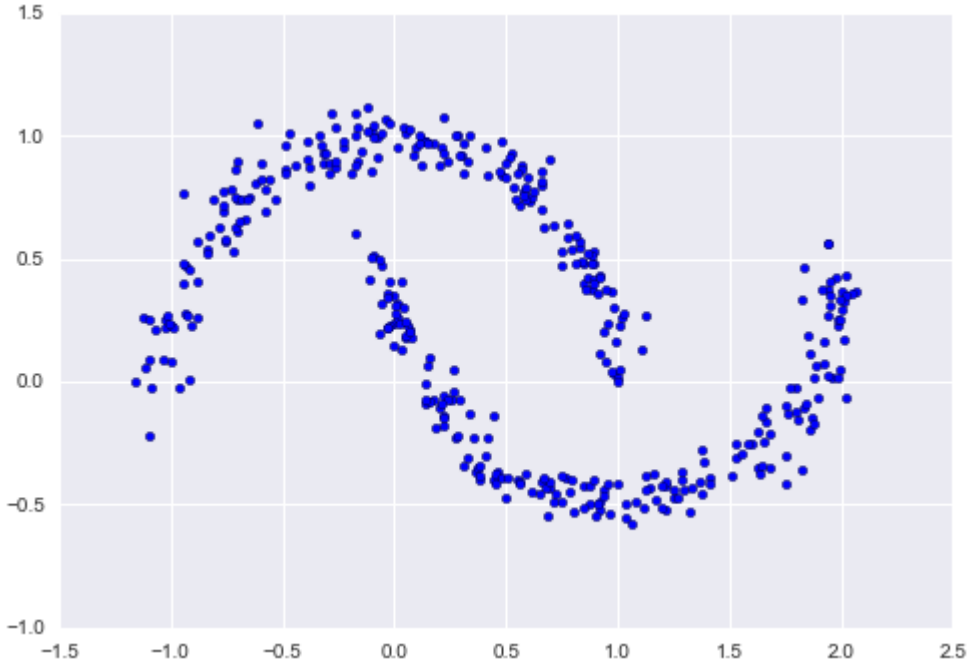
If we try to fit this with a two-component GMM viewed as a clustering model, the results are not particularly useful:



But if we instead use many more (16 Gaussians) components and ignore the cluster labels, we find a fit that is much closer to the input data.

Here the mixture of 16 Gaussians serves not to find separated clusters of data, but rather to model the overall distribution of the input data.

This is a generative model of the distribution, meaning that the GMM gives us the recipe to generate new random data distributed similarly to our input.

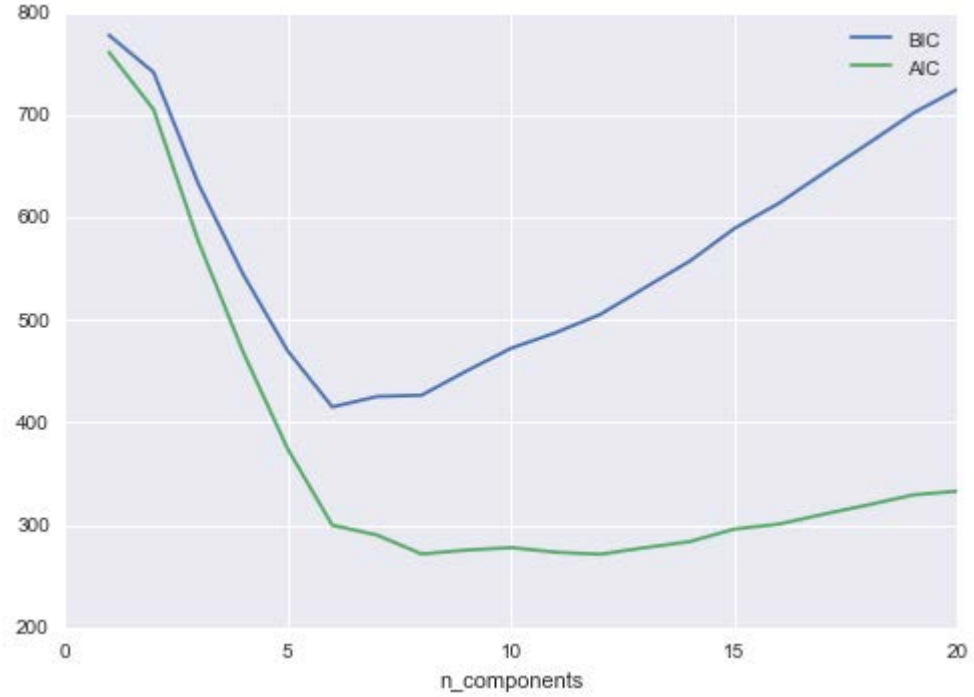


GMM is convenient as a flexible means of modeling an arbitrary multi-dimensional distribution of data.

2.5 How many components?

- The fact that GMM is a generative model gives us a natural means of determining the optimal number of components for a given dataset.
- A generative model is inherently a probability distribution for the dataset, and so we can simply evaluate the likelihood of the data under the model, using cross-validation to avoid over-fitting.
- Another means of correcting for over-fitting is to adjust the model likelihoods using some analytic criterion such as the Akaike information criterion (AIC) or the Bayesian information criterion (BIC).
- Scikit-Learn's GMM estimator actually includes built-in methods that compute both of these, and so it is very easy to operate on this approach.

- Let's look at the AIC and BIC as a function as the number of GMM components for the moon dataset:
- The optimal number of clusters is the value that minimizes the AIC or BIC, depending on which approximation we wish to use.
- The AIC tells us that our choice of 16 components above was probably too many: around 8-12 components would have been a better choice.
- As is typical with this sort of problem, the BIC recommends a simpler model.



Notice the important point: this choice of number of components measures how well GMM works as a density estimator, not how well it works as a clustering algorithm. I'd encourage you to think of GMM primarily as a density estimator, and use it for clustering only when warranted within simple datasets.

Example: GMM for Generating New Data

- We just saw a simple example of using GMM as a generative model of data in order to create new samples from the distribution defined by the input data. Here we will run with this idea and generate new handwritten digits from the standard digits corpus that we have used before.

- We have nearly 1,800 digits in 64 dimensions, and we can build a GMM on top of these to generate more. GMMs can have difficulty converging in such a high dimensional space, so we will start with an invertible dimensionality reduction algorithm on the data. Here we will use a straightforward PCA, asking it to preserve 99% of the variance in the projected data:

- Consider what we've done here: given a sampling of handwritten digits, we have modeled the distribution of that data in such a way that we can generate brand new samples of digits from the data:
  - these are "handwritten digits" which do not individually appear in the original dataset, but rather capture the general features of the input data as modeled by the mixture model.
  - Such a generative model of digits can prove very useful as a component of a Bayesian generative classifier, as we shall see in the next section.

The End!