9-Clustering-IV-GMM-EM

October 6, 2016

1 Clustering with Mixtures of Gaussians

1.1 Soft Clustering

We have seen how to cluster objects using k-means:

1. start with an initial set of cluster centers,
2. assign each object to its closest cluster center, and
3. recompute the centers of the new clusters.
4. Repeat 2->3 until convergence.

Note: Every object is assigned to a **single** cluster.

This is called **hard** assignment.

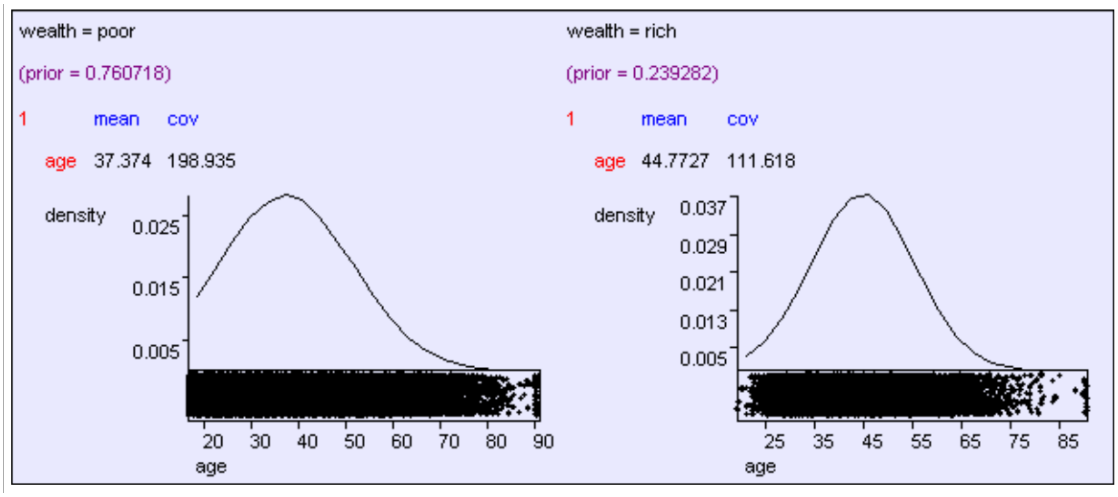
However, there may be cases were we either **cannot** use hard assignments or we do not **want** to do it!

In particular, we may believe that the best description of the data is a set of **overlapping** clusters.

For example:

Imagine that we believe society consists of just **two** kinds of individuals: poor, or rich.

Let's think about how we might model society as a mixture of poor and rich, when viewed in terms of age.



Clearly, viewed along the age dimension, there are two clusters that overlap.

Furthermore, given some particular individual at a given age, say 25, we cannot say for sure which cluster they belong to.

Rather, we will use *probability* to capture our uncertainty about the cluster that any single individual belongs to.

Thus, we could say that a given individual ("John Smith", age 25) belongs to the *rich* cluster with some probability, and the *poor* cluster with some different probability.

Naturally we expect the probabilities for John Smith to sum up to 1.

This is called **soft assignment,** and a clustering using this principle is called **soft clustering.**

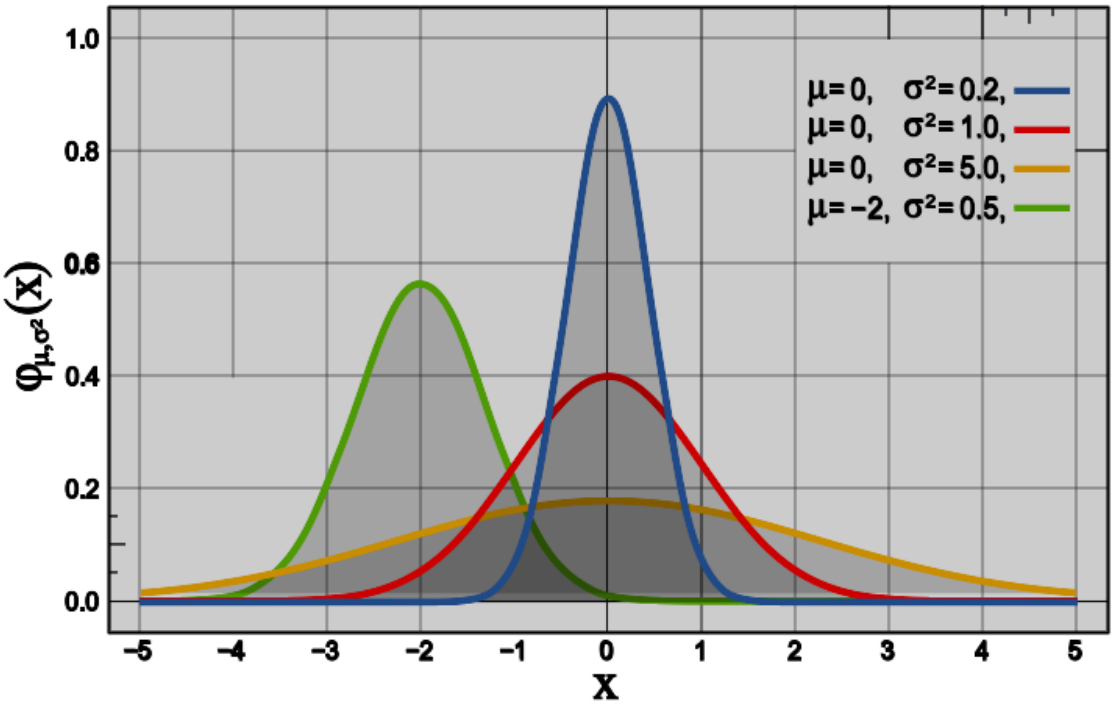
More formally, we say that an object can belong to each particular cluster with some probability, such that the sum of the probabilities adds up to 1 for each object.

For example, assuming that we have two clusters C_1 and C_2, we can have that an object x_1 belongs to C_1 with probability 0.3 and to C_2 with probability 0.7.

1.2 Mixtures of Gaussians

We're going to consider a particular model for each cluster: the Gaussian (or Normal) distribution.

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}}\, exp(-\frac{(x-\mu)^2}{2\sigma^2})$$



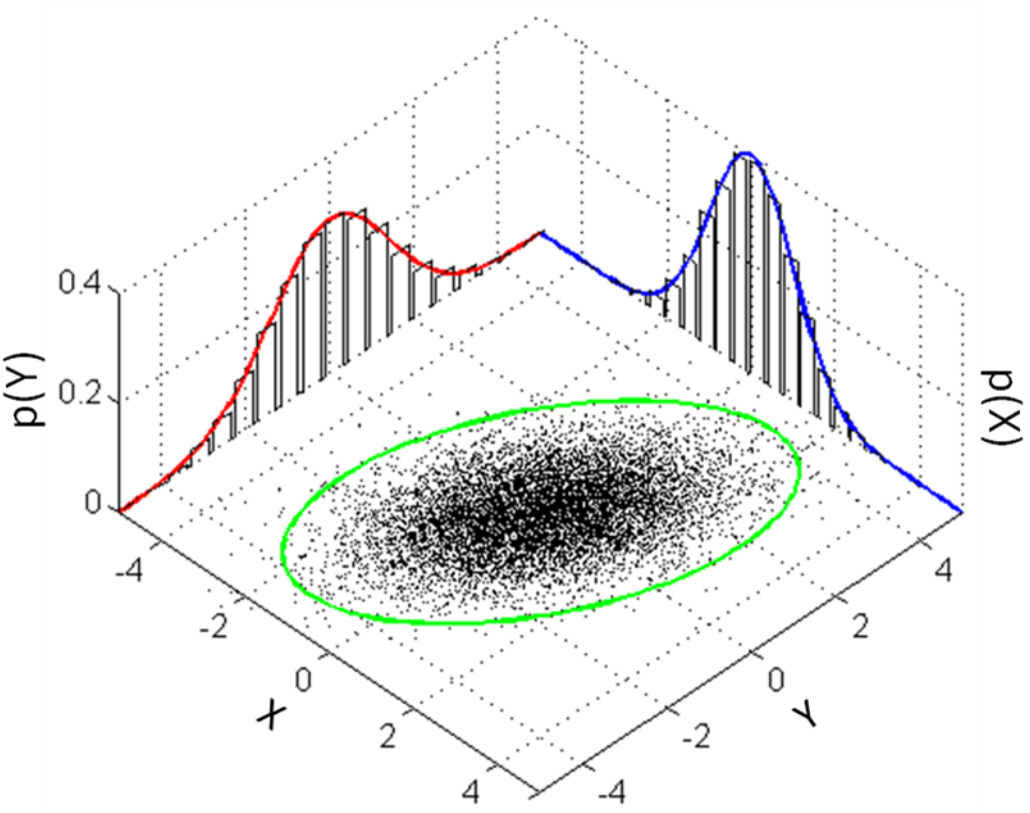
By Inductiveload - self-made, Mathematica, Inkscape, Public Domain, https://commons.wikimedia.org/w/index.php?curid=3817954

You can see that, for example, this is a reasonable model for the distribution of ages in each of the two population groups (rich and poor).

In the case of the population example, we have a single feature: age.

How do we use a Guassian when we have multiple features?

The answer is that we use a **multivariate Gaussian.**



By Bscan - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=25235145

Now each point is a vector in n dimensions: $\mathbf{x} = (x_1, \ldots, x_n)^T$.

Then a multivariate Gaussian has a pdf (density) function of:

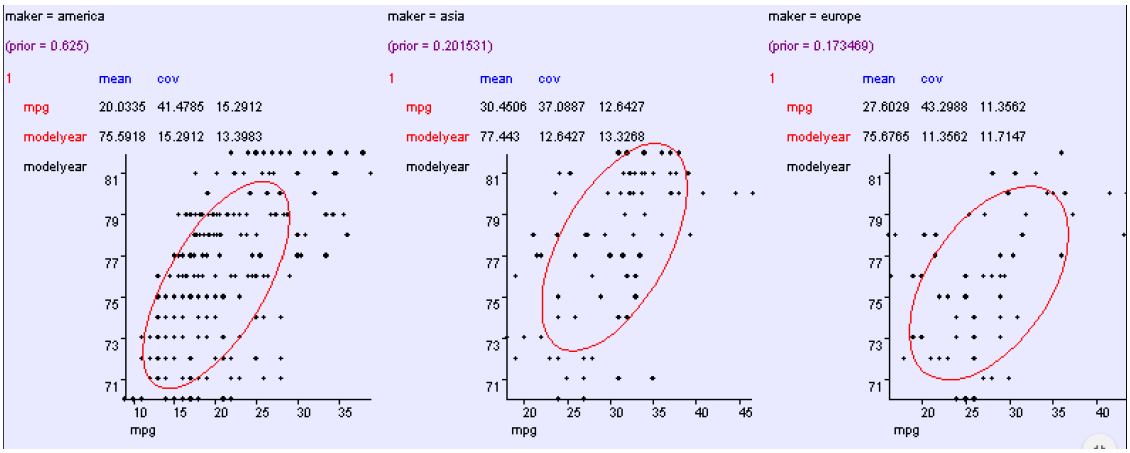
$$f(x_1, \ldots, x_n) = \frac{1}{\sqrt{(2\pi)^n \|\boldsymbol{\Sigma}\|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right)$$

Note that the shape of a multivariate Gaussian – the direction of its axes and the width along each axis – is determined by the **covariance matrix** Σ.

The covariance matrix is the multidimensional analog of the variance. It takes into account the extent to which vector components are correlated.

For example, let's say we are looking to classify cars based on their model year and miles per gallon (mpg).

To illustrate a particular model, let us consider the properties of cars produced in the US, Europe, and Asia.

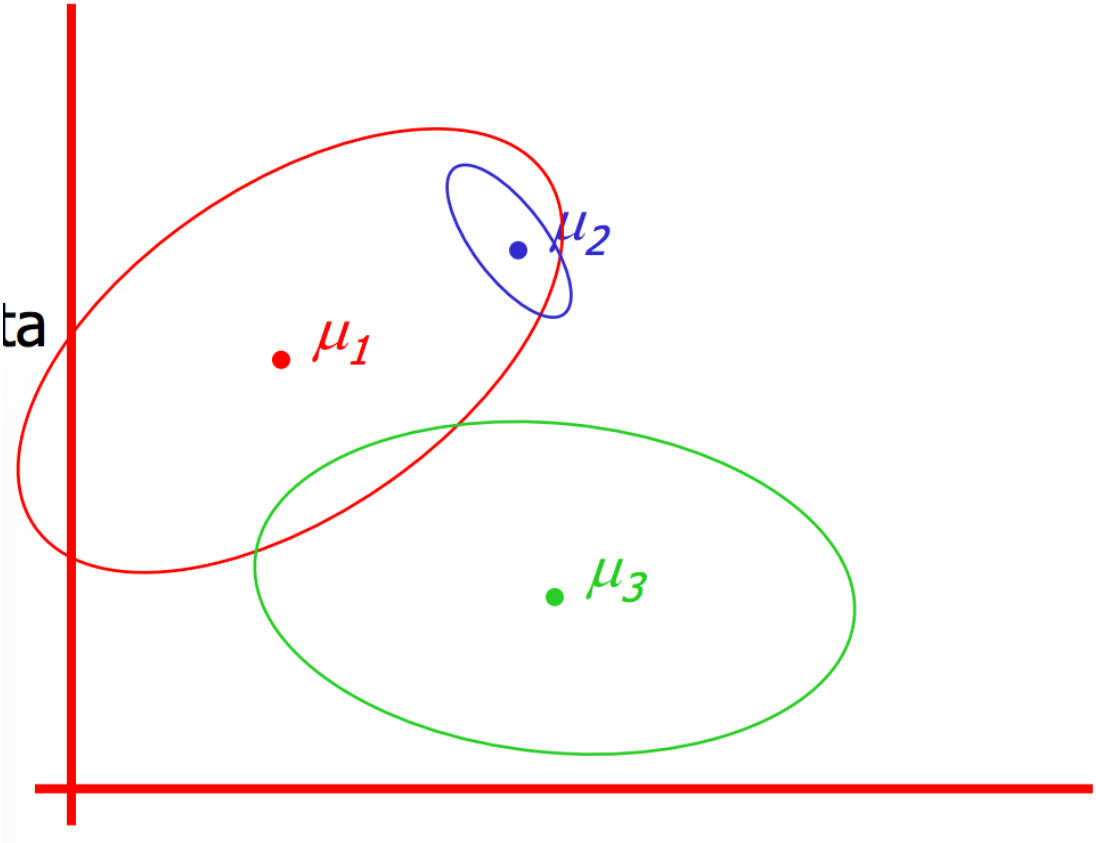


It seemes that the data can be described (roughly) as a mixture of **three multivariate** Gaussian distributions.

The general situation is that we assume the data was generated according to a collection of arbitrary Gaussian distributions.

This is called a **Gaussian Mixture Model.**

Aka a "GMM".



1.3 Estimating the parameters of a GMM

This model is all very well, but how do we estimate the parameters of such a model, given some data?

That is, assume we are told there are k clusters.

For each i in $1, \ldots, k$, how do we estimate the cluster mean μ_i and the cluster covariance Σ_i?

This is a learning problem.

There are a variety of ways of finding the best $(\mu_i, \Sigma_i) \quad i = 1, \ldots, k$.

We will consider the most popular method: **Expectation Maximization (EM).**

This is another famous algorithm, in the same "super-algorithm" league as k-means.

EM is formulated using a probabilistic model for data. It can solve a problem like:

> Given a set of data points and a parameter k, find the $(\mu_i, \Sigma_i) \quad i = 1, \ldots, k$ that **maximizes the likelihood of the data** assuming a GMM with those parameters.

(It can also solve lots of other problems involving maximizing likelihood of data under a model.)

However, note that problems of this type are often NP-hard.

EM only guarantees that it will find a **local** optimum of the objective function.

1.4 Probabilities We Will Use

At a high level, EM for the GMM problem has strong similarities to k-means.

However, there are two main differences:

1. The k-means problem posits a **hard** assignment of objects to clusters whereas GMM uses **soft** assignment.
2. The parameters of the soft assignment are chosed based on a **probability model** for the data.

Let's start by reviewing the situation probabilistically.

Assume a set data points $x_1, x_2, \ldots, x_n$ in a d dimensional space.

Also assume a set of k clusters $C_1, C_2, \ldots, C_k$. Each cluster is assumed to follow a Gaussian distribution:

$$C_i \sim \mathcal{N}(\mu_i, \mathbf{\Sigma}_i)$$

We will be working with conditional probabilities:

$P(x_i|C_j)$ is the probability of seeing data point x_i when sampling from cluster C_j.

That is, it is the value of a Gaussian pdf at the point x_i, for a Gaussian with parameters $(\mu_j, \mathbf{\Sigma}_j)$.

Clearly, if I give you the cluster parameters, it is a straightforward thing to compute this conditional probability.

We will also work with $P(C_j|x_i)$.

This is the probability that a data point at x_i was drawn from cluster C_j.

That is, the data point could have been drawn from any of the k clusters – what is the probability it was drawn from C_j in particular?

In other words, it is the probability of cluster C_j at point x_i.

How can we compute $P(C_j|x_i)$?

Note that the reverse conditional probability is easy to compute – so this is a job for **Bayes' Rule!**

$$P(C_j|x_i) = \frac{P(x_i|C_j)}{P(x_i)}P(C_j)$$

Finally, we will also need to estimate the parameters of each Gaussian, given some data.

This is also an easy problem.

For example, the best estimate for the mean of a Gaussian, given some data, is the **average** of the data points.

Likewise, there is a simple formula for the variance.

These are called **Maximum Likelihood Estimates** (MLE) of the parameters.

Often we use θ to denote "all the parameters of the model." In this case $\theta = (\mu_j, \mathbf{\Sigma}_j)$.

1.5 Expectation Maximization – The Algorithm

OK, now we have all the necessary pieces. Here is the algorithm.

Initialization:

Start with an initial set of clusters $C_1^1, C_2^1, \ldots, C_k^1$ and the initial probabilities that a random point belongs to each cluster $P(C_1), P(C_2), \ldots, P(C_k)$.

Just as in the k-means algorithm, these can be chosen essentially at random.

Step 1 (**Expectation**): For each point x_i, compute the probability that it belongs to each cluster C_j:

$$P(C_j|x_i) = \frac{P(x_i|C_j)}{P(x_i)}P(C_j)$$

(Thank you, Rev. Bayes!)

This is called the *posterior* probability of C_j given x_i.

We know how to compute everything on the right.

Note that: $P(x_i) = \sum_{j=1}^{k} P(x_i|C_j)P(C_j)$

Step 2 **(Maximization)**: Using the cluster membership probabilities computed in the previous step, compute new clusters (parameters) and cluster probabilities.

This is easy, using maximum likelihood estimates of the parameters θ.

$$P(C_j) = \frac{1}{n}\sum_{i=1}^{n} P(C_j|x_i)$$

Likewise, compute new parameters for the clusters $C_1, C_2, \ldots, C_n$ using MLE.

Repeat Steps 1 and 2 until stabilization.

Let's pause for a minute and compare GMM/EM with k-means.

GMM/EM:

1. Initialize randomly
2. Compute the probability that each point belongs in each cluster
3. Update the clusters (means and variances).
4. Repeat 2-3 until convergence.

k-means:

1. Initialize randomly
2. Assign each point to a single cluster
3. Update the clusters (means).
4. Repeat 2-3 until convergence.

From a practical standpoint, the main difference is that in GMM, data points do not belong to a **single** cluster, but have some probability of belonging to **each** cluster.

In other words, GMM uses soft assignment.

For that reason, GMM is also sometimes called **soft k-means.**

However, there is also an important conceptual difference.

The GMM starts by making an **explicit assumption** about how the data were generated.

It says: "the data came from a collection of multivariate Gaussians."

Note that we made no such assumption when we came up with the k-means problem. In that case, we simply defined an objective function and declared that it was a good one.

Nonetheless, it appears that we were making a sort of Gaussian assumption when we formulated the k-means objective function. However, **we didn't explicitly state it.**

The point is that because the GMM makes its assumptions explicit, we can

- examine them and think about whether they are valid
- replace them with different assumptions if we wish

This sort of assumption is called a "prior" on the data.

Philosophically, **exposing** this assumption is the heart of Bayesian statistics.

For example, it is perfectly possible to replace the Gaussian assumption with some other probility distribution. As long as we can estimate the parameters of such distributions from data (eg, have MLEs), we can use EM in that case as well.

1.6 Instantiating EM with the Gaussian Model

If we model each cluster as a multi-dimensional Gaussian, then we can instatiate every part of the algorithm.

This is the GMM (Gaussian Mixture Model) algorithm implemented in *sklearn.mixture* module.

In that case C_i is represented by (μ_i, Σ_i) and in EM Step 1 we compute:

$$P(x_i|C_j) = \frac{1}{\sqrt{(2\pi)^d|\Sigma_j|}} exp(-\frac{1}{2}(x_i - \mu_j)^T\Sigma_j^{-1}(x_i - \mu_j))$$

In EM Step 2, we estimate the parameters of the Gaussian using the appropriate MLEs:

$$\mu_j' = \frac{\sum_{i=1}^n P(C_j|x_i)x_i}{\sum_{i=1}^n P(C_j|x_i)}$$

and

$$\Sigma_j = \frac{\sum_{i=1}^n P(C_j|x_i)(x_i - \mu_j')(x_i - \mu_j')^T}{\sum_{i=1}^n P(C_j|x_i)}$$

A final statement about EM generally. EM is a versatile algorithm that can be used in many other settings. What is the main idea behind it?

Notice that the problem definition only required that we find the clusters, C_i, meaning that we were to find the (μ_i, Σ_i).

However, the EM algorithm posited that we should find as well the $P(C_j|x_i)$, that is, the probability that each point is a member of each cluster.

This is the true heart of what EM does.

The idea is called "data augmentation."

By **adding parameters** to the problem, it actually finds a way to make the problem solvable!

These parameters don't show up in the solution. They are sometimes called "hidden parameters."

So the basic strategy for using EM is: think up some **additional** information which, if you had it, would make the problem solvable.

Figure out how to estimate the additional information from a solved problem, and put the two steps into a loop.

Here is an example using **GMM**.

```
In [108]: import time
          import itertools

          import numpy as np
          from scipy import linalg
          import matplotlib.pyplot as plt
          import matplotlib as mpl

          from sklearn import mixture

          import seaborn as sns

          # Number of samples per component
          n_samples = 1000

          # Generate random sample, two components
          np.random.seed(0)
          # C is a transfomation that will make a heavily skewed 2-D Gaussian
          C = np.array([[0.1, -0.1], [1.7, .4]])
          print(C.T@C)
```

```
[[ 2.9   0.67]
 [ 0.67  0.17]]
```

```
In [109]: # now we construct a data matrix that has n_samples from the skewed distr
          # and n_samples from a symmetric distribution offset to position (-6,3)
          X = np.r_[(np.random.randn(n_samples, 2) @ C),
                    .7 * np.random.randn(n_samples, 2) + np.array([-6, 3])]
          print(X[:10])
```

```
[[ 0.85667249 -0.01634235]
 [ 3.90739224  0.79848348]
 [-1.4746166  -0.57766695]
 [-0.16229841 -0.15555173]
 [ 0.68769557  0.17456129]]
```

```
 [ 2.48666932  0.56730505]
 [ 0.2829513  -0.02743377]
 [ 0.61163268  0.08908341]
 [-0.19936114 -0.23147121]
 [-1.42065599 -0.37294507]]
```

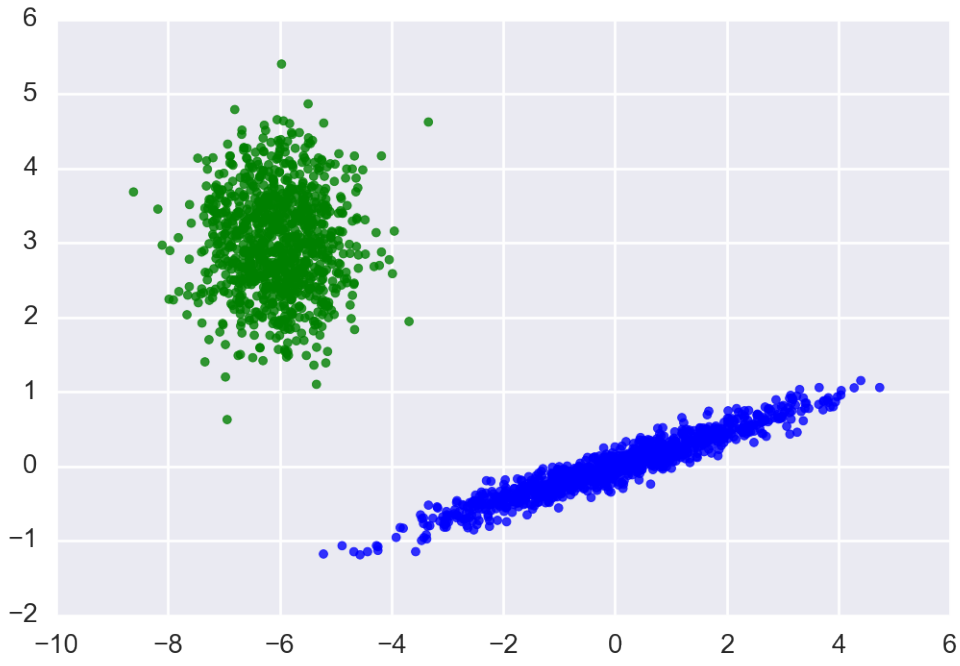
```
In [110]: # Fit a mixture of Gaussians with EM using two components
          gmm = mixture.GMM(n_components=2, covariance_type='full')

          gmm.fit(X)

          y_pred = gmm.predict(X)
```

```
In [111]: colors = ['bg'[p] for p in y_pred]
          plt.scatter(X[:, 0], X[:, 1], color=colors, s=10, alpha=0.8)
```

```
Out[111]: <matplotlib.collections.PathCollection at 0x127029dd8>
```

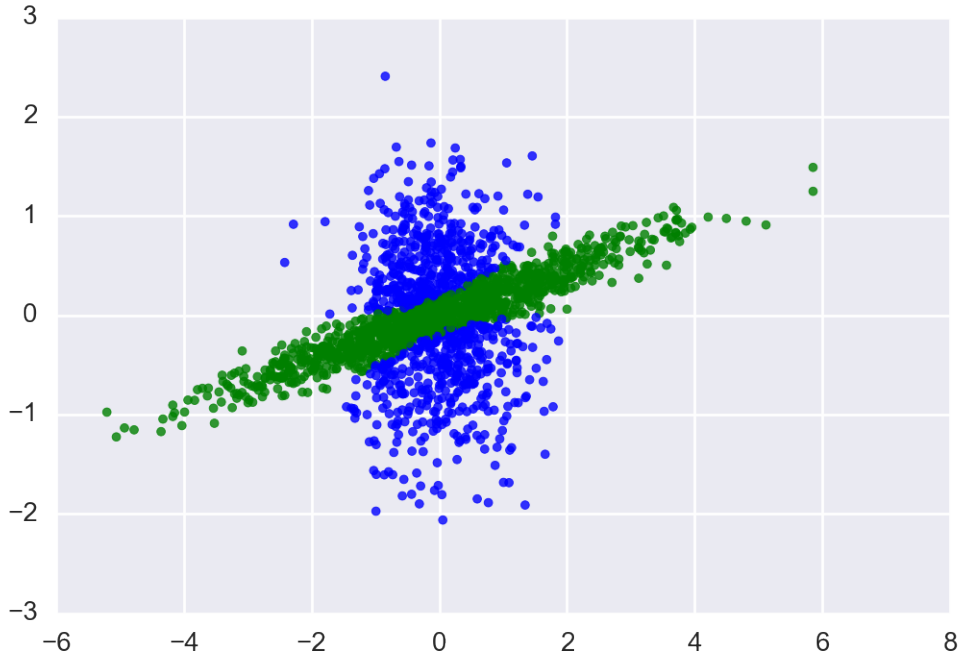


Now, let's construct **overlapping** clusters. What will happen?

```
In [121]: X = np.r_[np.random.randn(n_samples, 2) @ C,
                    0.7 * np.random.randn(n_samples, 2) ]
```

```
In [122]: gmm = mixture.GMM(n_components=2, covariance_type='full')
          gmm.fit(X)
          y_pred = gmm.predict(X)
          colors = ['bg'[p] for p in y_pred]
          plt.scatter(X[:, 0], X[:, 1], color=colors, s=10, alpha=0.8)
```

```
Out[122]: <matplotlib.collections.PathCollection at 0x1284f4208>
```




```
In [114]: gmm.means_

Out[114]: array([[-0.04347597,  0.02001986],
                 [ 0.02529301,  0.00051634]])

In [115]: gmm.covars_

Out[115]: array([[[ 0.52255641, -0.01812917],
                  [-0.01812917,  0.53131311]],

                 [[ 3.03412607,  0.70727119],
                  [ 0.70727119,  0.18215318]]])
```

1.7 How many parameters are estimated?

Most of the parameters in the model are contained in the covariance matrices.

In the most general case, for k clusters of points in n dimensions, there are k covariance matrices each of size $n \times n$.

So we need kn^2 parameters to specify this model.

It can happen that you may not have enough data to estimate so many parameters.

Also, it can happen that you believe that clusters should have some constraints on their shapes.

Here is where the GMM assumptions become really useful.

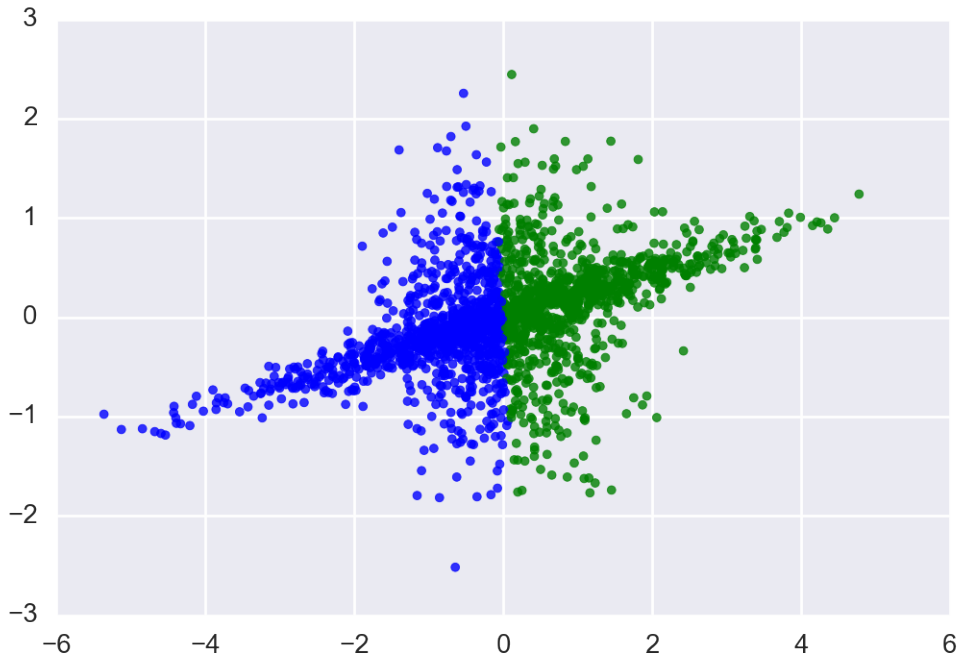
Let's say you believe all the clusters should have the same shape, but the shape can be arbitrary.

Then you only need to estimate **one** covariance matrix - just n^2 parameters.

This is specified by the GMM parameter `covariance_type='tied'`.

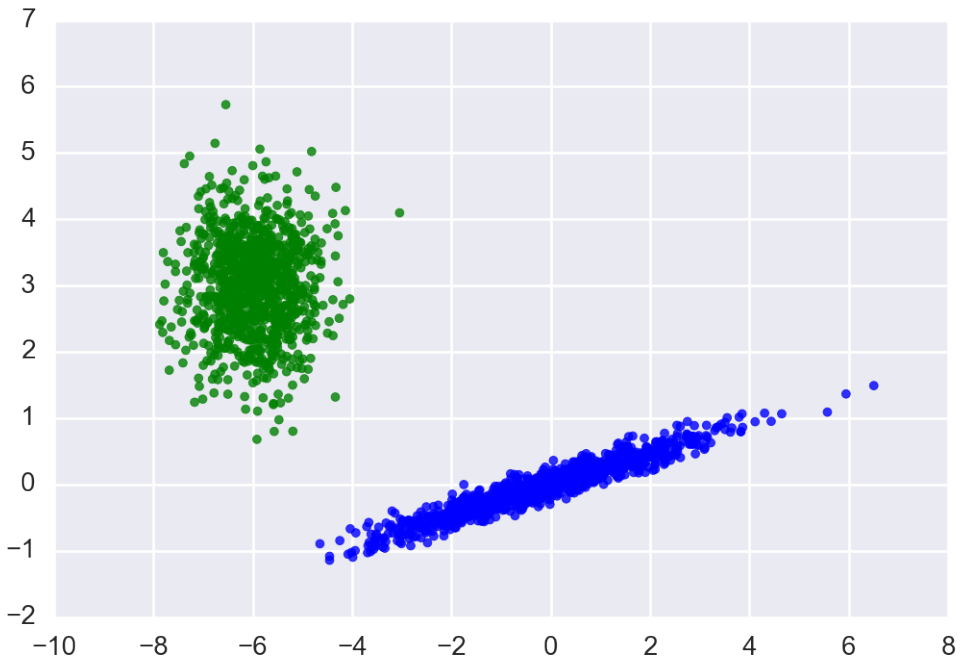
```
In [116]: X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
                    0.7 * np.random.randn(n_samples, 2) ]
          gmm = mixture.GMM(n_components=2, covariance_type='tied')
          gmm.fit(X)
          y_pred = gmm.predict(X)
          colors = ['bg'[p] for p in y_pred]
          plt.scatter(X[:, 0], X[:, 1], color=colors, s=10, alpha=0.8)

Out[116]: <matplotlib.collections.PathCollection at 0x1276ae080>
```




```
In [117]: X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
                    0.7 * np.random.randn(n_samples, 2) + np.array([-6,3])]
          gmm = mixture.GMM(n_components=2, covariance_type='tied')
          gmm.fit(X)
          y_pred = gmm.predict(X)
          colors = ['bg'[p] for p in y_pred]
          plt.scatter(X[:, 0], X[:, 1], color=colors, s=10, alpha=0.8)

Out[117]: <matplotlib.collections.PathCollection at 0x1279e3d30>
```



Perhaps you believe in even more restricted shapes: all clusters should have their axes aligned with the coordinate axes.

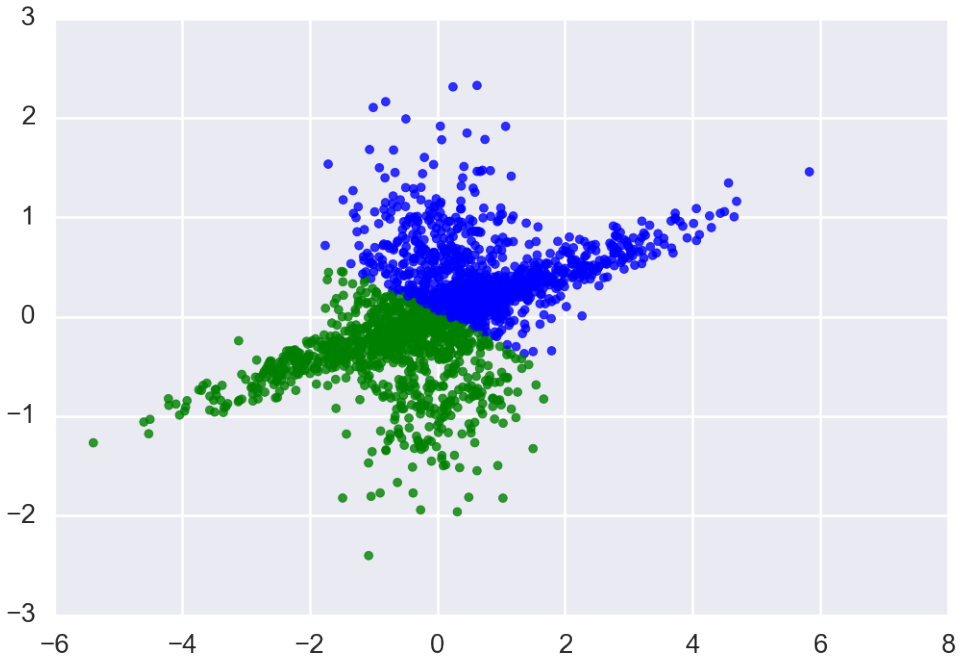
That is, clusters are not skewed.

Then you only need to estimate the diagonals of the covariance matrices - just kn parameters.

This is specified by the GMM parameter `covariance_type='diag'`.

```
In [118]: X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
                    0.7 * np.random.randn(n_samples, 2)]
          gmm = mixture.GMM(n_components=2, covariance_type='diag')
          gmm.fit(X)
          y_pred = gmm.predict(X)
          colors = ['bg'[p] for p in y_pred]
          plt.scatter(X[:, 0], X[:, 1], color=colors, s=10, alpha=0.8)

Out[118]: <matplotlib.collections.PathCollection at 0x127d2e1d0>
```



Finally, if you believe that all clusters should be round, then you only need to estimate the k variances.

This is specified by the GMM parameter `covariance_type='spherical'`.