Abdul Qayyum

Lecturer at University of Burgundy, France

- Postdoc in Electrical and Informatics Engineering
- PhD in Electrical & Electronics Engineering
- Masters in Electronics Engineering
- Bachelor in Computer Engineering

<u>Collaborations & Expertise:</u>

Topic: Clustering and GMM Algorithms

Instructor: Abdul Qayyum, PhD

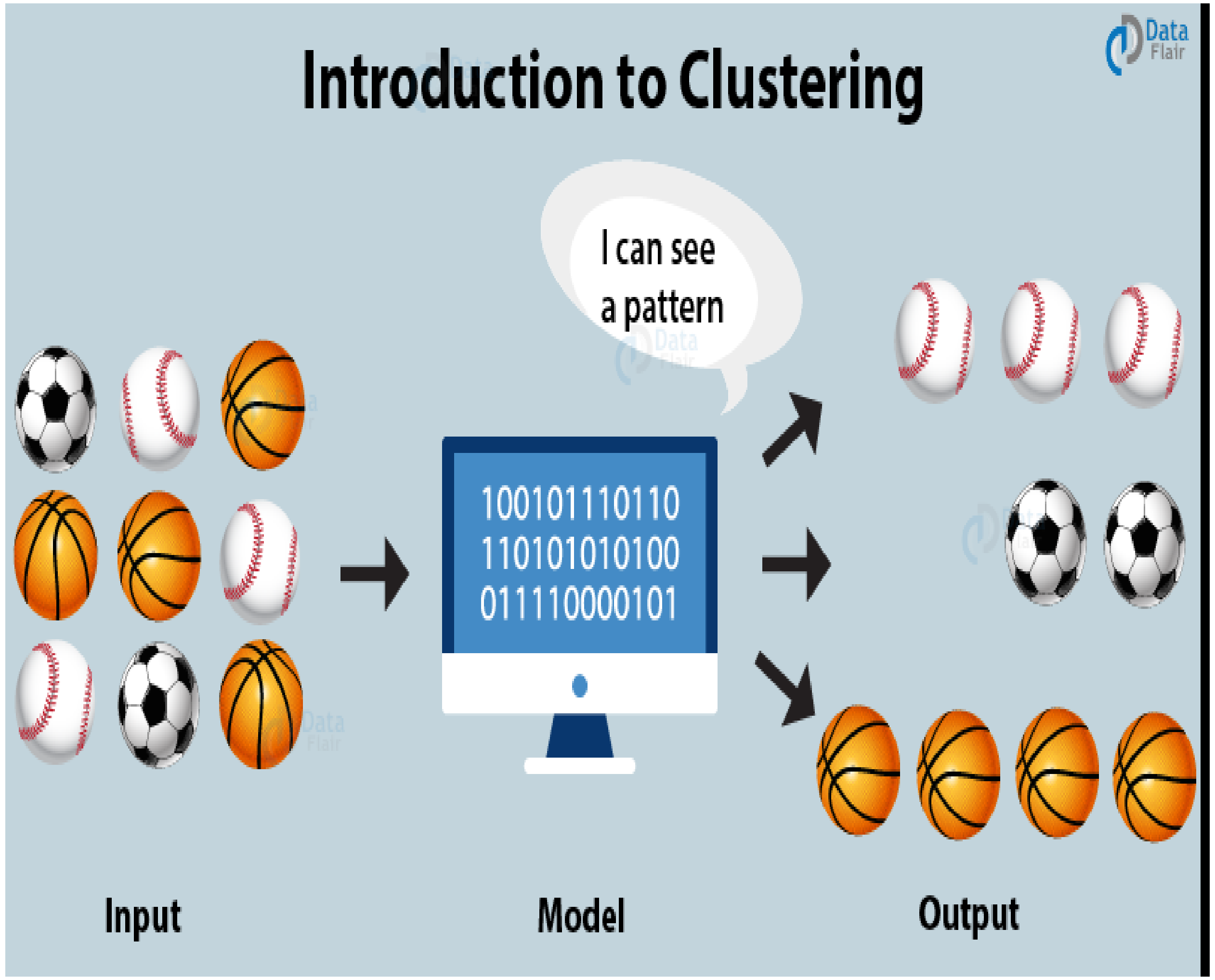
Class: MSCV

University of Burgundy, France

Clustering Analysis

- In this chapter, we will cover the following topics:
- Kmean Clustering algorithm
- Gaussian Mixture Model(GMM)
- Practical example in data generation and segmentation applications

What is Clustering Analysis?



Clustering (or cluster analysis) is a technique that allows us to find groups of similar objects that are more related to each other than to objects in other groups.

Examples of business- oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines.

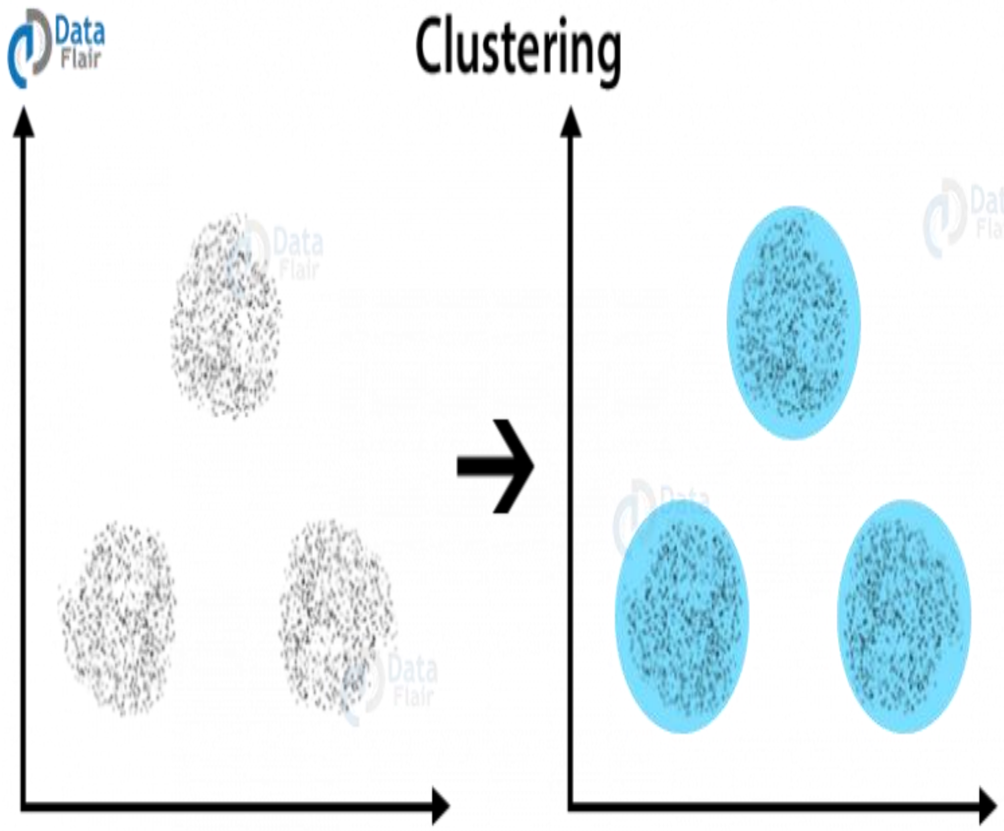
Types of Clustering Algorithms
In total, there are five distinct types of clustering algorithms.

- **Partitioning Based Clustering**
- **Hierarchical Clustering**
- **Model-Based Clustering**
- **Density-Based Clustering**
- **Fuzzy Clustering**

What is Clustering Analysis?

The goal of clustering is to find a natural grouping in data so that items in the same cluster are more similar to each other than to those from different clusters

Clustering in Real World
Data Flair
+
=
Identifying the potential customer base for selling the product
Implementing Clustering Algorithms to group the customer base
Selling product to the identified customer group

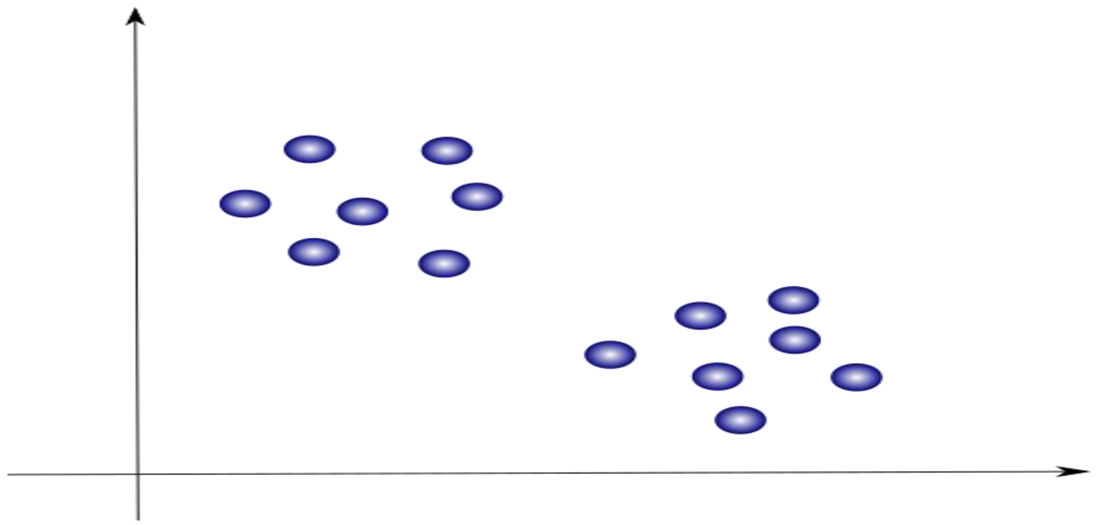


What is Clustering Analysis?

- **Working with Unlabeled Data – Clustering Analysis**
- k-means algorithm is extremely easy to implement, but it is also computationally very efficient compared to other clustering algorithms.
- The k-means algorithm belongs to the category of prototype-based clustering.
- We can discuss two other categories of clustering, hierarchical and density-based clustering
- Prototype-based clustering means that each cluster is represented by a prototype, which is usually either the centroid (average) of similar points with continuous features, or the medoid (the most representative or the point that minimizes the distance to all other points that belong to a particular cluster) in the case of categorical features. While k-means is very good at identifying clusters with a spherical shape, one of the drawbacks of this clustering algorithm is that we have to **specify the number of clusters, k, a priori.**
- An inappropriate choice for k can result in poor clustering performance. Later , we will discuss the **elbow method and silhouette plots**, which are useful techniques to evaluate the quality of a clustering to help us determine the optimal number of clusters, k.

What is Clustering Analysis?

In the world of Machine Learning, we can distinguish
two main areas: **Supervised and unsupervised learning.**
The main difference between both lies in the nature of the data as well as the approaches used to deal with it. Clustering is an unsupervised learning problem where we intend to find clusters of points in our dataset that share some common characteristics. Let's suppose we have a dataset that looks like this:



What is Clustering Analysis?

Our job is to find sets of points that appear close together. In this case, we can clearly identify two clusters of points which we will colour blue and red figure a and four cluster form in figure b.

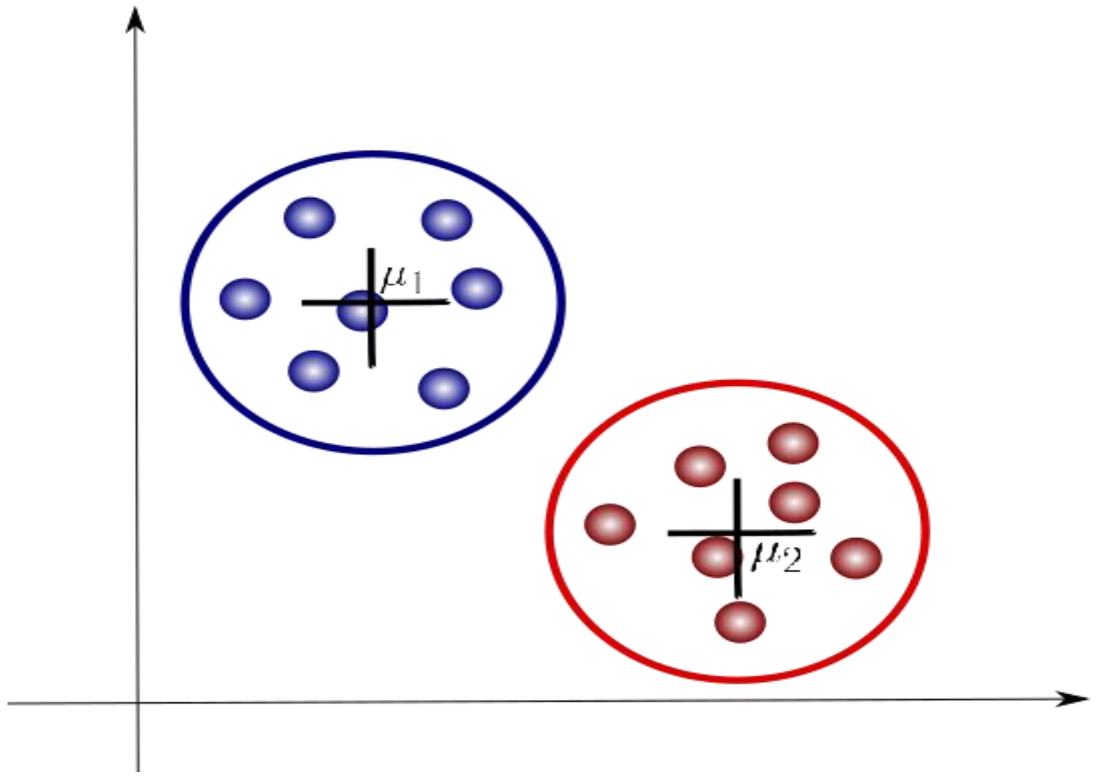


Figure a

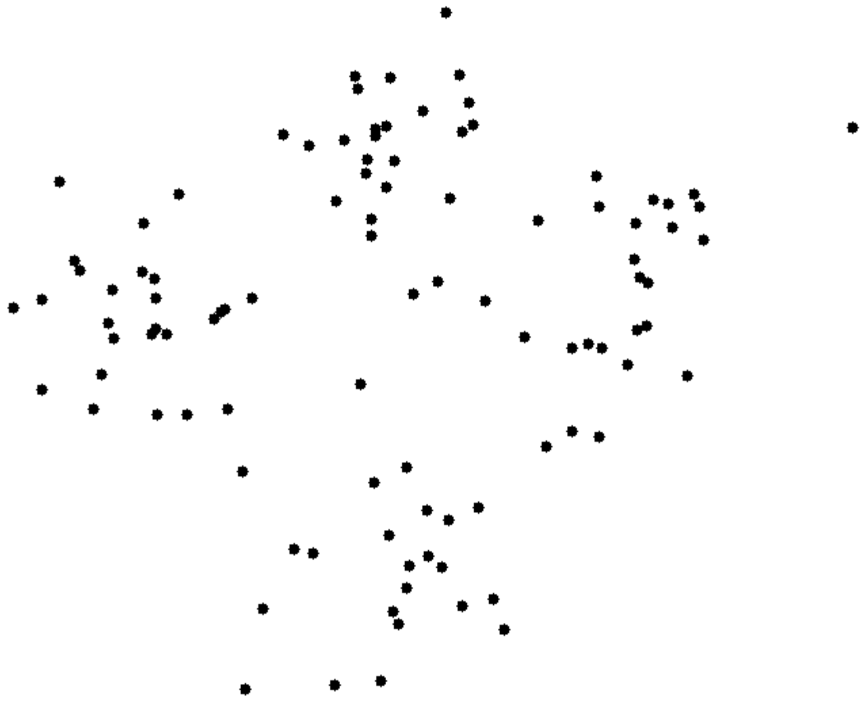
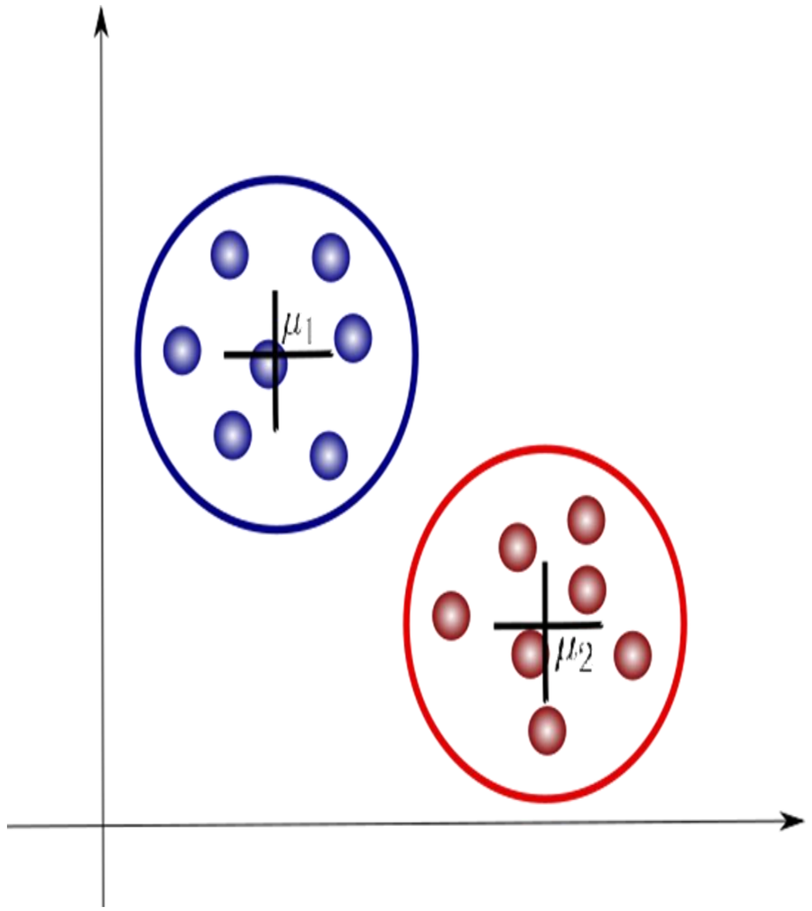


Figure b

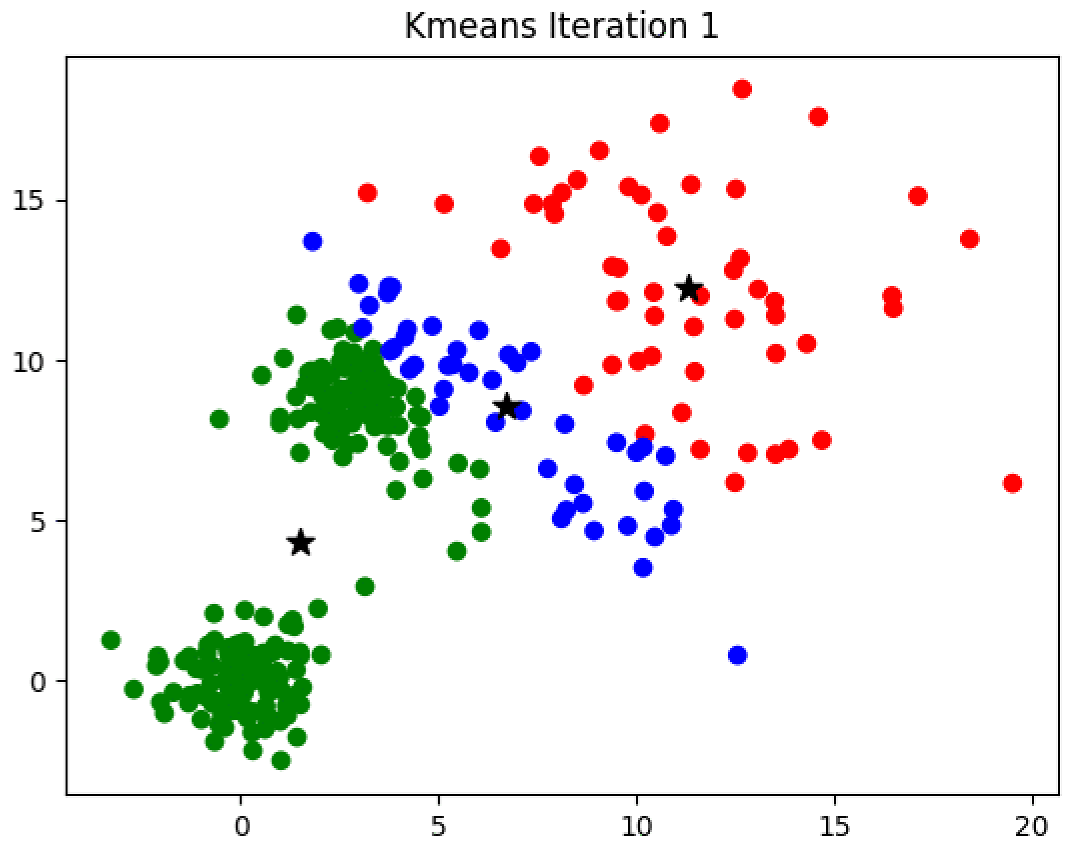
What is Clustering Analysis?

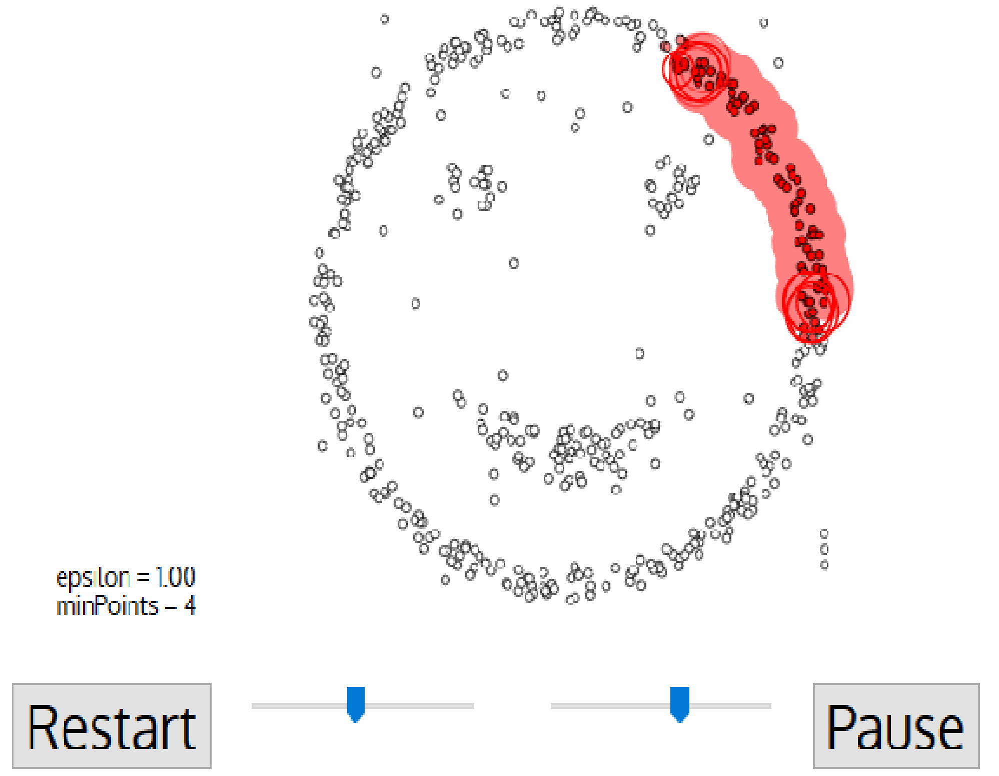
Please note that we are now introducing some additional notation. Here, μ1 and μ2 are the centroids of each cluster and are parameters that identify each of these. A popular clustering algorithm is known as K-means, which will follow an iterative approach to update the parameters of each clusters.

More specifically, what it will do is to compute the means (or centroids) of each cluster, and then calculate their distance to each of the data points. The latter are then labeled as part of the cluster that is identified by their closest centroid. This process is repeated until some convergence criterion is met, for example when we see no further changes in the cluster assignments.



What is Clustering Analysis?





What is Clustering Analysis?

One important characteristic of K-means is that it is a hard clustering method, which means that it will associate each point to one and only one cluster. A limitation to this approach is that there is no uncertainty measure or probability that tells us how much a data point is associated with a specific cluster. So what about using a soft clustering instead of a hard one? This is exactly what Gaussian Mixture Models, or simply GMMs, attempt to do.

What is Clustering Analysis?

Slide taken from Prof Fabrice course

- One of the key components of unsupervised learning is Clustering

  Clustering is a mathematical tool that attempts to discover structures or certain patterns in a data set, where the objects inside each cluster show a certain degree of similarity

- Hard clustering
  - Assign each sample to one class only
- Fuzzy clustering
  - Each sample has a degree of membership to all classes

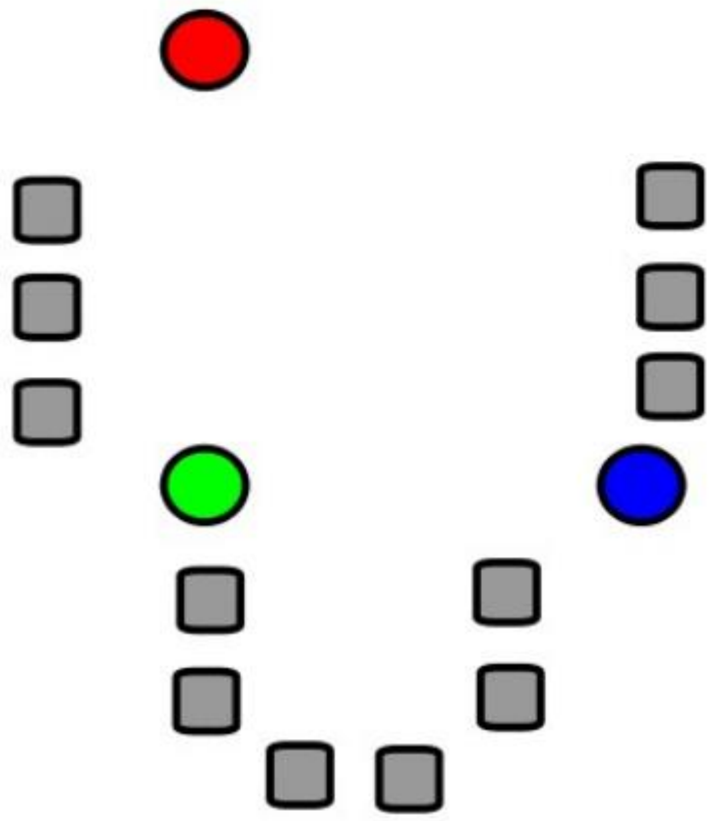
What is Clustering Analysis?

Slide taken from Prof Fabrice course

- The most known example of hard clustering
- Composed of three steps
  - Initialisation
    - The number of cluster and cluster centres are identified
  - Assignment step
    - Assign each observation to the cluster with the closest mean
  - Update step
    - Calculate the new means to be the centroid of the observations in the cluster
  - Repeat Assignment/Update steps until convergence is found

What is K-Means?

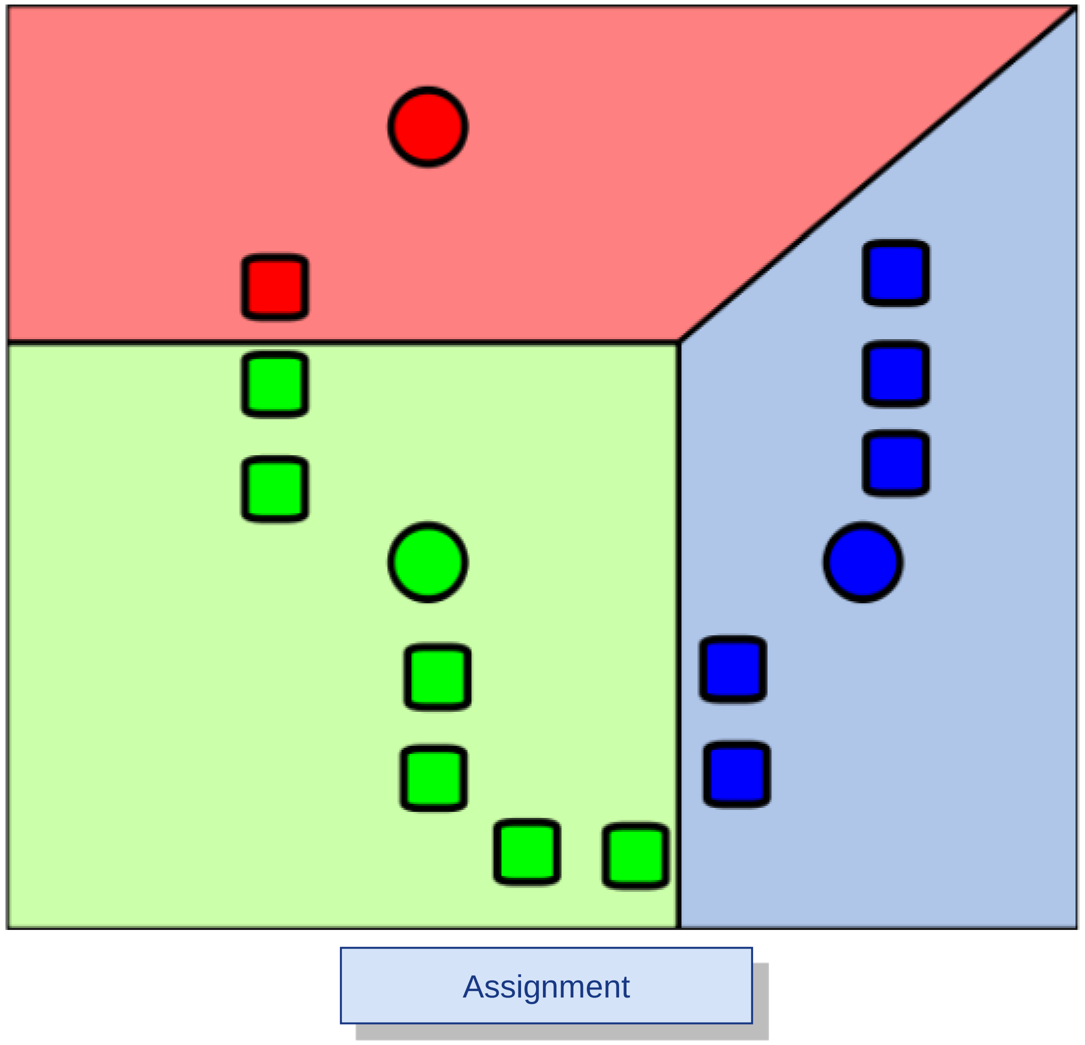
Slide taken from Prof Fabrice course



Initialisation

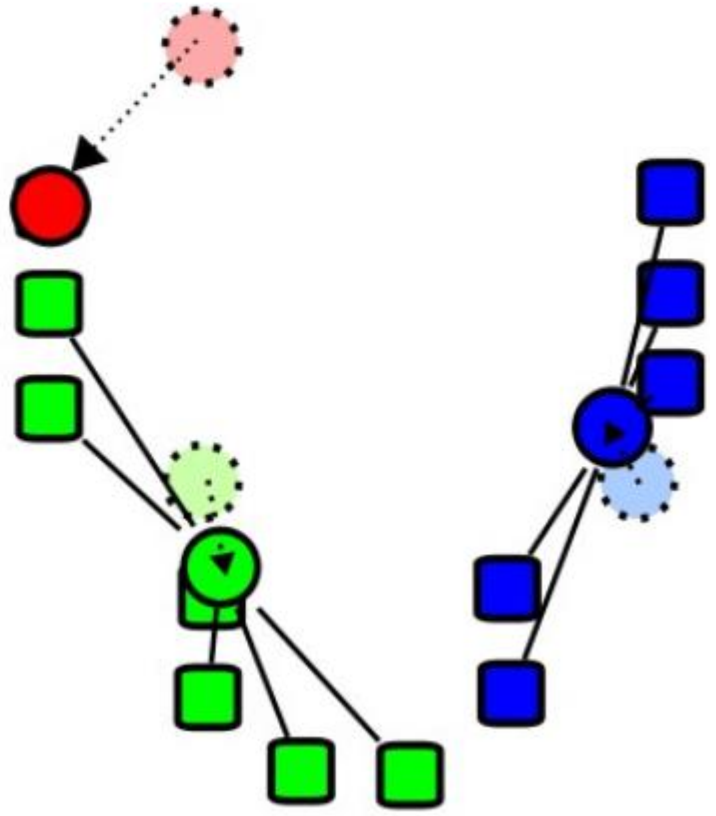
What is K-Means?

Slide taken from Prof Fabrice course



What is K-Means?

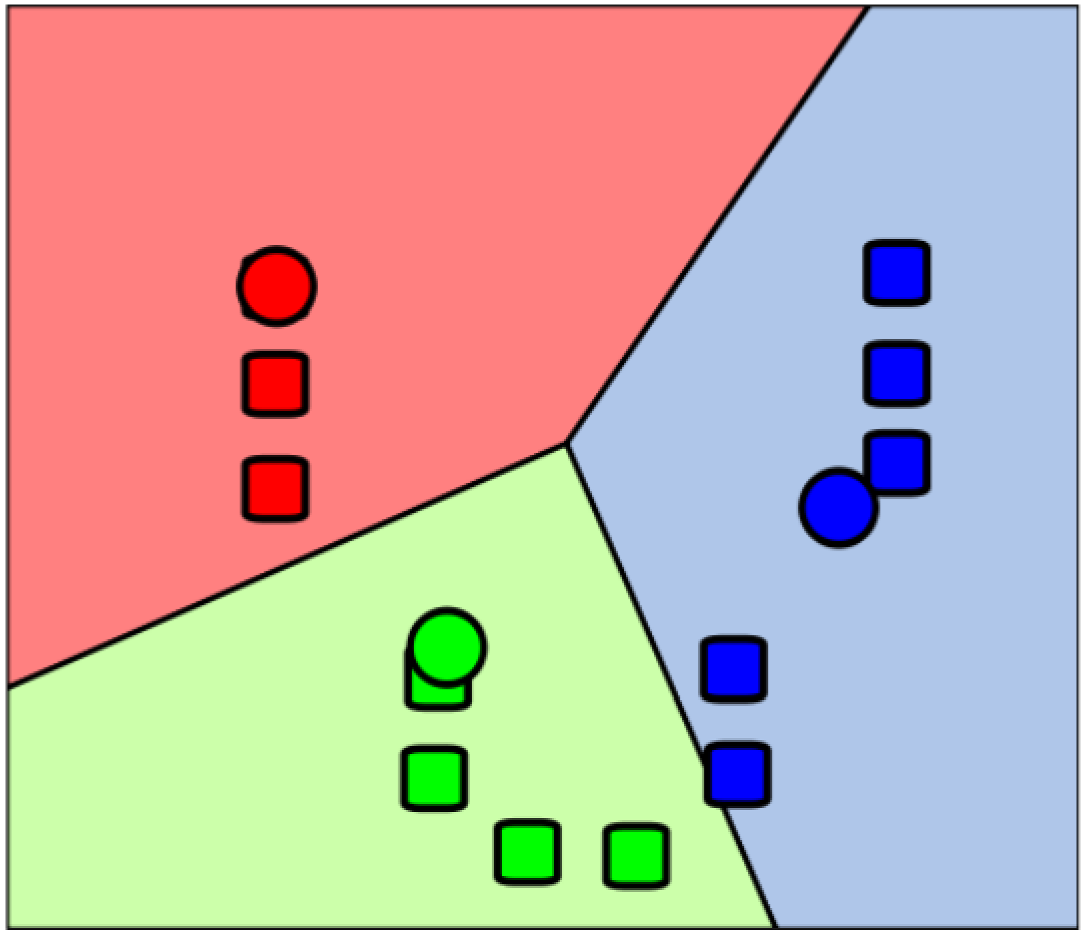
Slide taken from Prof Fabrice course



Update

What is K-Means?

Slide taken from Prof Fabrice course

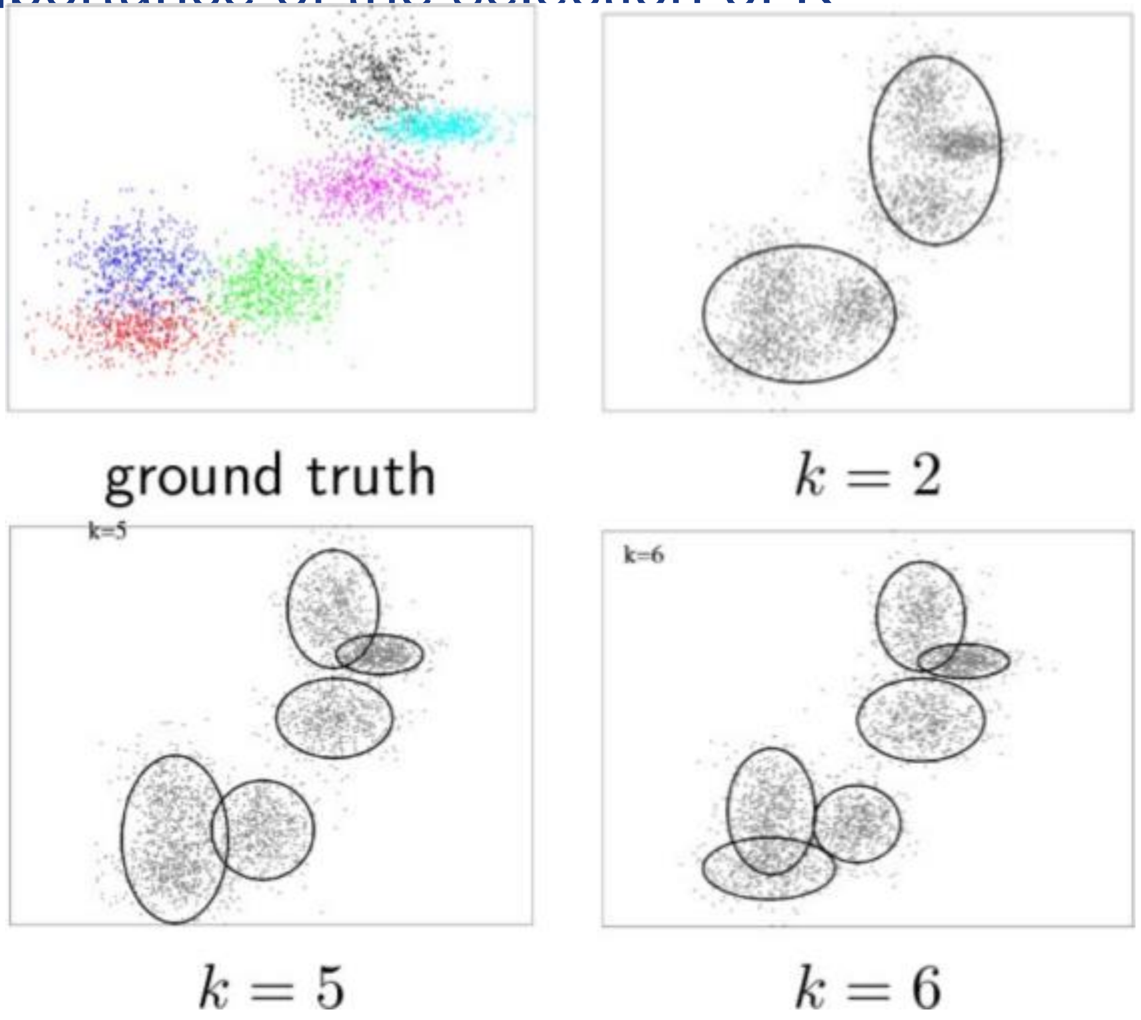


Check convergence

What is K-Means?

Slide taken from Prof Fabrice course

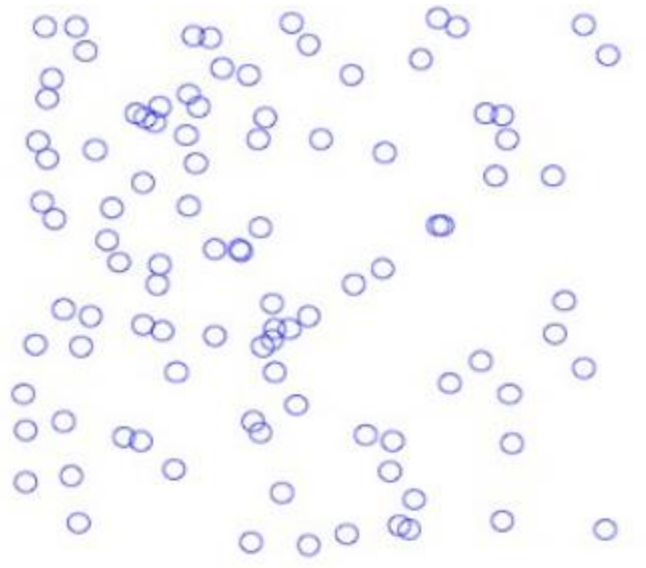
- Importance of the selection of K



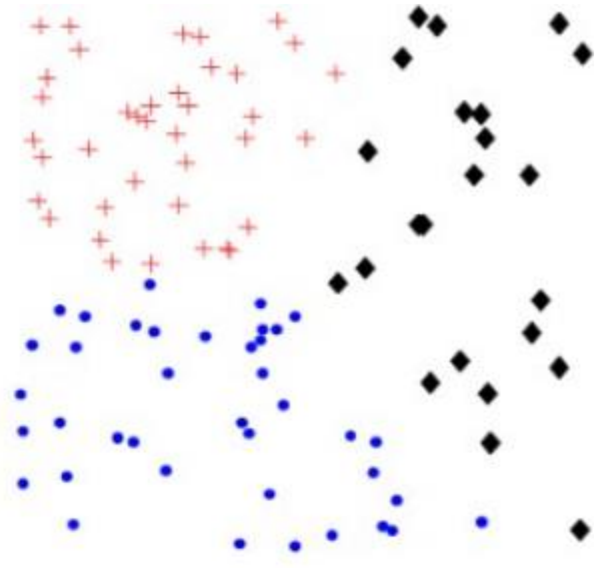
What is K-Means?

Slide taken from Prof Fabrice course

- Clustering algorithms find clusters, even if there are no natural clusters in the data



100 2D uniform data points



k-Means with k=3

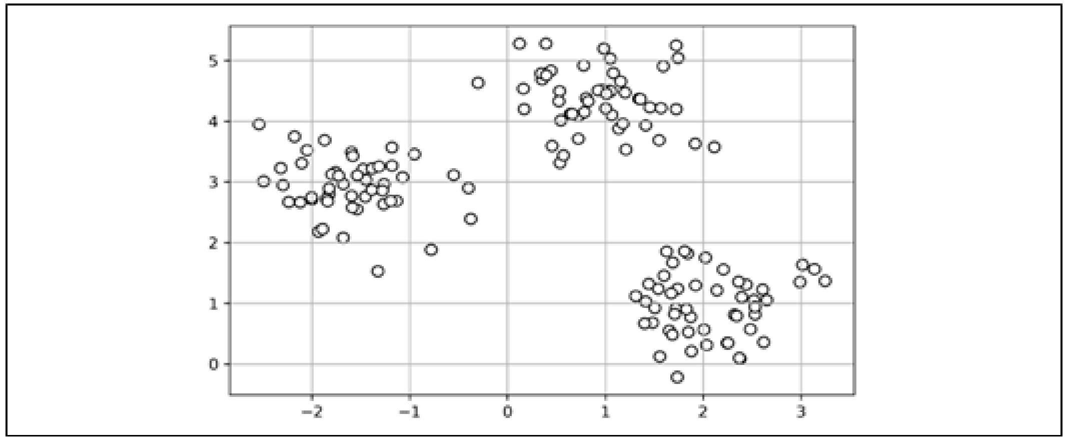
Clustering Analysis

- **Working with Unlabeled Data – Clustering Analysis**
- Although k-means clustering can be applied to data in higher dimensions,
- We will walk through the following examples using a simple two-dimensional dataset for the purpose of visualization:

```
>>> from sklearn.datasets import make_blobs
>>> X, y = make_blobs(n_samples=150,
...                 n_features=2,
...                 centers=3,
...                 cluster_std=0.5,
...                 shuffle=True,
...                 random_state=0)

>>> import matplotlib.pyplot as plt
>>> plt.scatter(X[:, 0],
...                 X[:, 1],
...                 c='white',
...                 marker='o',
...                 edgecolor='black',
...                 s=50)
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

Clustering Analysis

- **Working with Unlabeled Data – Clustering Analysis**
- The dataset that we just created consists of 150 randomly generated points that are roughly grouped into three regions with higher density, which is visualized via a two-dimensional scatterplot.
- In real-world applications of clustering, we do not have any ground truth category information (information provided as empirical evidence as opposed to inference) about those examples; **if we were given class labels, this task would fall into the category of supervised learning**. Thus, our goal is to group the examples based on their **feature similarities**, which can be achieved using the k-means algorithm



Clustering Analysis

- **Working with Unlabeled Data – Clustering Analysis**
- The k-means algorithm, as summarized by the following four steps:
- 1. Randomly pick k centroids from the examples as initial cluster centers.
- 2. Assign each example to the nearest centroid, $\mu^{(j)}, j \in \{1, \ldots, k\}$
- 3. Move the centroids to the center of the examples that were assigned to it.
- 4. Repeat steps 2 and 3 until the cluster assignments do not change or a user-defined tolerance or maximum number of iterations is reached.
- Now, **the next question is, how do we measure similarity between objects?** We can define similarity as the opposite of distance, and a commonly used distance for clustering examples with **continuous features** is the **squared Euclidean distance between two points, x and y, in m-dimensional space**:
- $d(x,y)^2 = \sum_{j=1}^{m}\left(x_j - y_j\right)^2 = \|x - y\|_2^2$
- Note that, in the preceding equation, the index j refers to the jth dimension (feature column) of the example inputs, x and y. We will use the superscripts i and j to refer to the index of the example (data record) and cluster index, respectively.

Clustering Analysis

- **Working with Unlabeled Data – Clustering Analysis**
- Based on this Euclidean distance metric, we can describe the k-means algorithm as a simple optimization problem, an iterative approach for minimizing the within-cluster sum of squared errors (SSE), which is sometimes also called **cluster inertia**:
- $SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)} \left\| x^{(i)} - \mu^{(j)} \right\|_2^2$
- Here, $\mu^{(j)}$ is the representative point (centroid) for cluster j. $w^{(i,j)}$ = 1 if the example,
- $x^{(i)}$, is in cluster j, or 0 otherwise.
- $w^{(i,j)} = \begin{cases} 1 & if\, x^{(i)} \in j \\ 0 & otherwise \end{cases}$

- Now that you have learned how the simple k-means algorithm works, let's apply it to our example dataset using the KMeans class from scikit-learn's cluster module:

Clustering Analysis

- **Working with Unlabeled Data – Clustering Analysis**
- Now that you have learned how the simple k-means algorithm works, let's apply it to our example dataset using the KMeans class from scikit-learn's cluster module:
- We set the number of desired clusters to 3; specifying the number of clusters a priori is one of the limitations of k-means. We set n_init=10 to run the k-means clustering algorithms 10 times independently, with different random centroids to choose the final model as the one with the lowest SSE. Via the max_iter parameter, we specify the maximum number of iterations for each single run (here, 300). One way to deal with convergence problems is to choose larger values for tol, which is a parameter that controls the tolerance with regard to the changes in the within-cluster SSE to declare convergence. We chose a tolerance of 1e-04 (=0.0001).

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
...             init='random',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
```

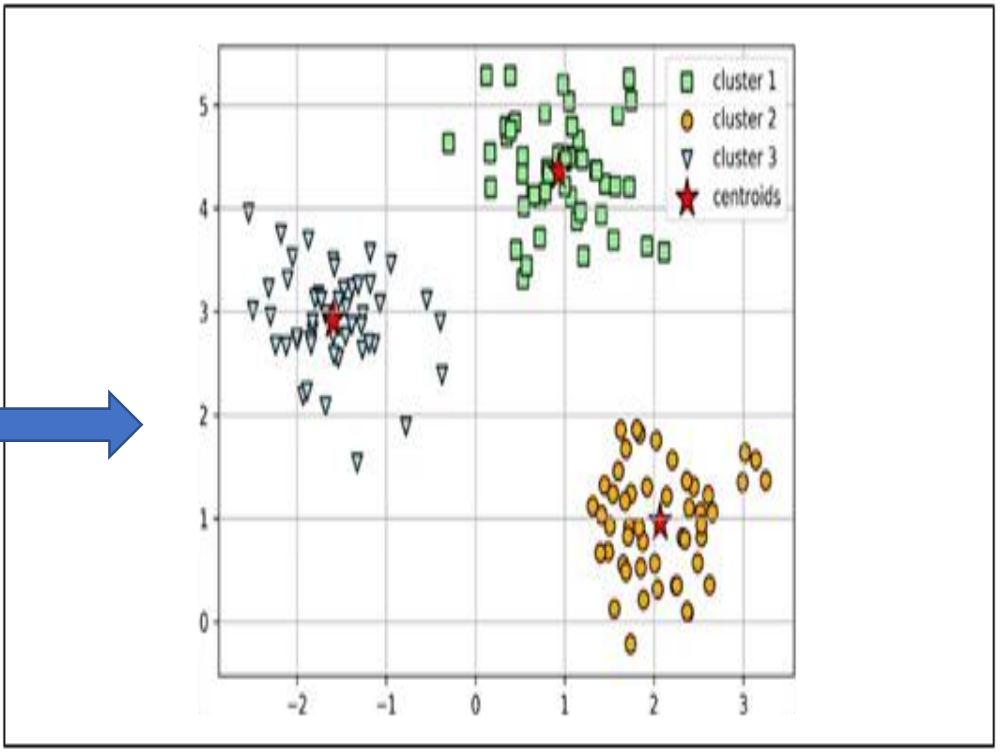
Clustering Analysis

- **Working with Unlabeled Data – Clustering Analysis**
- A problem with k-means is that one or more clusters can be empty. However, this problem is accounted for in the current k-means implementation in scikit-learn. If a cluster is empty, the algorithm will search for the example that is farthest away from the centroid of the empty cluster. Then it will reassign the centroid to be this farthest point.
- Note that this problem does not exist for **k-medoids or fuzzy C-means, an algorithm h**aving predicted the cluster labels, y_km, and discussed some of the challenges of the k-means algorithm, let's now visualize the clusters that k-means identified in the dataset together with the cluster centroids. These are stored under the cluster_centers_ attribute of the fitted KMeans object:

Clustering Analysis

- **Working with Unlabeled Data – Clustering Analysis**
- Having predicted the cluster labels, y_km, and discussed some of the challenges of the k-means algorithm, let's now visualize the clusters that k-means identified in the dataset together with the cluster centroids. These are stored under the cluster_centers_ attribute of the fitted KMeans object:

```
>>> plt.scatter(X[y_km == 0, 0],
...             X[y_km == 0, 1],
...             s=50, c='lightgreen',
...             marker='s', edgecolor='black',
...             label='Cluster 1')
>>> plt.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             s=50, c='orange',
...             marker='o', edgecolor='black',
...             label='Cluster 2')
>>> plt.scatter(X[y_km == 2, 0],
...             X[y_km == 2, 1],
...             s=50, c='lightblue',
...             marker='v', edgecolor='black',
...             label='Cluster 3')
>>> plt.scatter(km.cluster_centers_[:, 0],
...             km.cluster_centers_[:, 1],
...             s=250, marker='*',
...             c='red', edgecolor='black',
...             label='Centroids')
>>> plt.legend(scatterpoints=1)
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```



Clustering Analysis

- **Working with Unlabeled Data – Clustering Analysis**
- Drawback of k-means: we have to specify the number of clusters, k, a priori. The number of clusters to choose may not always be so obvious in real-world applications, especially if we are working with a higher dimensional dataset that cannot be visualized.
- The other properties of k-means are that clusters do not overlap and are not hierarchical, and we also assume that there is at least one item in each cluster.
- The different types of clustering algorithms, **hierarchical and density-based clustering**. These type of algorithm does not require to specify the number of clusters upfront or assume spherical structures in our dataset.
- **A smarter way of placing the initial cluster centroids using k-means++**
- It can greatly improve the clustering results through more clever seeding of the initial cluster centers.

Clustering Analysis

- **Working with Unlabeled Data – Clustering Analysis**
- **A smarter way of placing the initial cluster centroids using k-means++**
- So far, we have discussed the classic k-means algorithm, which uses a random seed to place the initial centroids, which can sometimes result in **bad clusterings or slow convergence if the initial centroids are chosen poorly**.
- One way to address this issue is to run the k-means algorithm multiple times on a dataset and choose the best performing model in terms of the SSE.
- Another strategy is to place the initial centroids far away from each other via the k-means++ algorithm, which leads to better and more consistent results than the classic k-means (k-means++: The Advantages of Careful Seeding, D. Arthur and S. Vassilvitskii in Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, pages 1027-1035. Society for Industrial and Applied Mathematics, 2007).

Clustering Analysis

- **A smarter way of placing the initial cluster centroids using k-means++**
- **The initialization in k-means++ can be summarized as follows:**
- 1. Initialize an empty set, **M**, to store the k centroids being selected.
- 2. Randomly choose the first centroid, $\boldsymbol{\mu}^{(j)}$, from the input examples and assign it to **M**.
- 3. For each example, $\boldsymbol{x}^{(i)}$,that is not in **M**, find the minimum squared distance, $d(\boldsymbol{x}^{(i)} - \boldsymbol{M})^2$, to any of the centroids in **M**.
- 4. To randomly select the next centroid, $\boldsymbol{\mu}^{(p)}$, use a weighted probability distribution equal to $\frac{d(\boldsymbol{\mu}^{(p)},\boldsymbol{M})^2}{\sum_i d(\boldsymbol{x}^{(i)},\boldsymbol{M})^2}$
- 5. Repeat steps 2 and 3 until k centroids are chosen.
- 6. Proceed with the classic k-means algorithm.

To use k-means++ with scikit-learn's KMeans object, we just need to set the init parameter to 'k-means++'. In fact, 'k-means++' is the default argument to the init parameter, which is strongly recommended in practice. You are encouraged to experiment more with the two different approaches (classic k-means via init='random' versus k-means++ via init='k-means++') for placing the initial cluster centroids.

Clustering Analysis

- **Hard versus soft clustering**

Hard clustering describes a family of algorithms where each example in a dataset is assigned to exactly one cluster, as in the k-means and k-means++ algorithms
In contrast, algorithms for soft clustering (sometimes also called fuzzy clustering) assign an example to one or more clusters. A popular example of soft clustering is the fuzzy C-means (FCM) algorithm (also called soft k-means or fuzzy k-means).

The original idea goes back to the 1970s, when Joseph C. Dunn first proposed an early version of fuzzy clustering to improve k-means (A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters, J. C. Dunn, 1973). Almost a decade later, James C. Bedzek published his work on the improvement of the fuzzy clustering algorithm, which is now known as the FCM algorithm (Pattern Recognition with Fuzzy Objective Function Algorithms, J. C. Bezdek, Springer Science+Business Media, 2013).

Clustering Analysis

- **Hard versus soft clustering**
- The FCM procedure is very similar to k-means. However, we replace the hard cluster assignment with probabilities for each point belonging to each cluster. In k-means, we could express the cluster membership of an example, x, with a sparse vector of binary values:

$$\begin{bmatrix} x \in \mu^{(1)} \rightarrow w^{(i,j)} = 0 \\ x \in \mu^{(2)} \rightarrow w^{(i,j)} = 1 \\ x \in \mu^{(3)} \rightarrow w^{(i,j)} = 0 \end{bmatrix}$$

- Here, the index position with value 1 indicates the cluster centroid, $\mu^{(j)}$, that the example is assigned to (assuming k= 3, $j \in \{1, 2, 3\}$).
- In contrast, a membership vector in FCM could be represented as follows:

$$\begin{bmatrix} x \in \mu^{(1)} \rightarrow w^{(i,j)} = 0.1 \\ x \in \mu^{(2)} \rightarrow w^{(i,j)} = 0.85 \\ x \in \mu^{(3)} \rightarrow w^{(i,j)} = 0.05 \end{bmatrix}$$

- Here, each value falls in the range [0, 1] and represents a probability of membership of the respective cluster centroid. The sum of the memberships for a given example is equal to 1.

Clustering Analysis

- **FCM clustering**
- 1. Specify the number of k centroids and randomly assign the cluster memberships for each point.
- 2. Compute the cluster centroids, $\mu^{(j)}, j \in \{1, \dots, k\}$.
- 3. Update the cluster memberships for each point.
- 4. Repeat steps 2 and 3 until the membership coefficients do not change or a user-defined tolerance or maximum number of iterations is reached.
- The objective function of FCM—we abbreviate it as j_m—looks very similar to the within-cluster SSE that we minimize in k-means:
- $j_m = \sum_{i=1}^{n} \sum_{j=1}^{k} \left(w^{(i,j)}\right)^m \left\| x^{(i)} - \mu^{(j)} \right\|_2^2$
- However, note that the membership indicator, $w^{(i,j)}$, is not a binary value as in k-means ($w^{(i,j)} \in \{0, 1\}$), but a real value that denotes the cluster membership probability ($w^{(i,j)}) \in [0, 1]$). You also may have noticed that we added an additional exponent to $w^{(i,j)}$; the exponent m, any number greater than or equal to one (typically m = 2), is the so-called fuzziness coefficient (or simply fuzzifier), which controls the degree of fuzziness. The larger the value of m, the smaller the cluster membership, $w^{(i,j)}$, becomes, which leads to fuzzier clusters.

Clustering Analysis

- **FCM clustering**
- The cluster membership probability itself is calculated as follows:
- $w^{(i,j)} = \left[\sum_{c=1}^{k} \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| x^{(i)} - \boldsymbol{\mu}^{(c)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$
- For example, if we chose three cluster centers, as in the previous k-means example, we could calculate the membership of $x^{(i)}$ belonging to the $\mu^{(j)}$ cluster as follows:
- $w^{(i,j)} = \left[\left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| x^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$
- The center, $\boldsymbol{\mu}^{(j)}$, of a cluster itself is calculated as the mean of all examples weighted by the degree to which each example belongs to that cluster ($\left(w^{(i,j)}\right)^m$):
- $\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^{n} \left(w^{(i,j)}\right)^m \boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} \left(w^{(i,j)}\right)^m}$

Clustering Analysis

- **FCM clustering**

$$w^{(i,j)} = \left[\sum_{c=1}^{k} \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(c)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

$$\Downarrow$$

Just by looking at the equation to calculate the cluster memberships, we can say that each iteration in FCM is more expensive than an iteration in k-means. On the other hand, FCM typically requires fewer iterations overall to reach convergence. **Unfortunately, the FCM algorithm is currently not implemented in scikit-learn.** However, it has been found, in practice, that both k-means and FCM produce very similar clustering outputs, as described in a study (Comparative Analysis of k-means and Fuzzy C-Means Algorithms, S. Ghosh, and S. K. Dubey, IJACSA, 4: 35–38, 2013).

Clustering Analysis

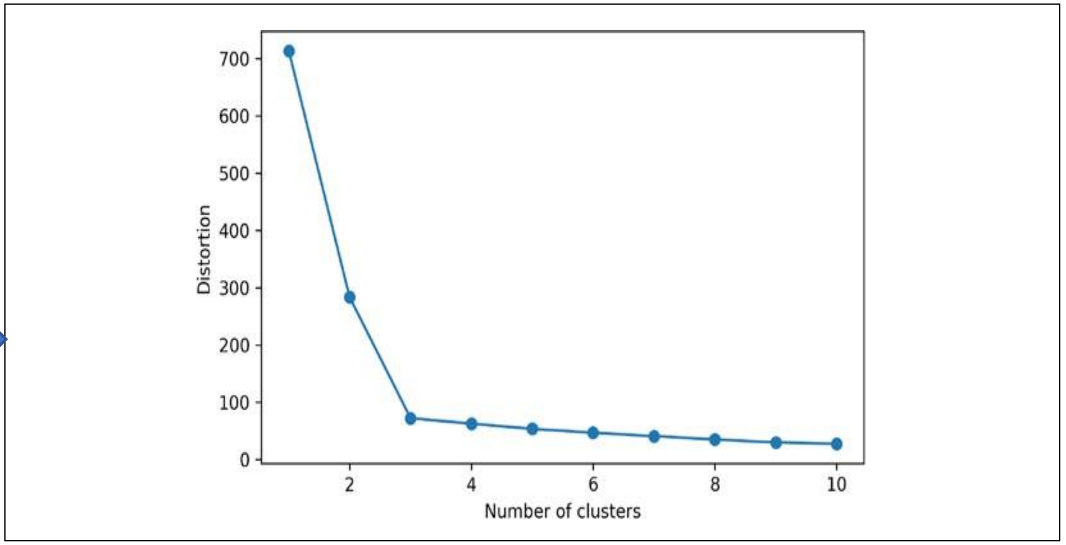
- **Using the elbow method to find the optimal number of clusters**
- One of the main challenges in unsupervised learning is that we do not know the definitive answer. We don't have the ground truth class labels in our dataset. In order to evaluate the performance of a supervised model.
- Thus, to quantify the quality of clustering, we need to use intrinsic metrics—**such as the within-cluster SSE (distortion)—to compare the performance of different k-means clusterings.**
- Conveniently, we don't need to compute the within-cluster SSE explicitly when we are using scikit-learn, as it is already accessible via the inertia_ attribute after fitting a KMeans model:

```
>>> print('Distortion: %.2f' % km.inertia_) Distortion: 72.48
```

Clustering Analysis

- **Using the elbow method to find the optimal number of clusters**
- Based on the within-cluster SSE, we can use a graphical tool, the so-called elbow method, to estimate the optimal number of clusters, k, for a given task.
- We can say that if k increases, the distortion will decrease. This is because the examples will be closer to the centroids they are assigned to.
- The idea behind the elbow method is to identify the value of k where the distortion begins to increase most rapidly, which will become clearer if we plot the distortion for different values of k:

```
>>> distortions = []
>>> for i in range(1, 11):
...         km = KMeans(n_clusters=i,
...         init='k-means++',
...         n_init=10,
...         max_iter=300,
...         random_state=0)
...         km.fit(X)
...         distortions.append(km.inertia_)
>>> plt.plot(range(1,11), distortions, marker='o')
>>> plt.xlabel('Number of clusters')
>>> plt.ylabel('Distortion')
>>> plt.tight_layout()
>>> plt.show()
```



As you can see in the plot, the elbow is located at k = 3, so this is evidence that k = 3 is indeed a good choice for this dataset:

Clustering Analysis

- **Quantifying the quality of clustering via silhouette plots**
- Another intrinsic metric to evaluate the quality of a clustering is silhouette analysis, which can also be applied to clustering algorithms other than k-means.
- Silhouette analysis can be used as a graphical tool to plot a measure of how tightly grouped the examples in the clusters are. To calculate the silhouette coefficient of a single example in our dataset, we can apply the following three steps:
- 1. Calculate the cluster cohesion, $a^{(i)}$, as the average distance between an example, $\boldsymbol{x}^{(i)}$ and all other points in the same cluster.
- 2. Calculate the cluster separation, $b^{(i)}$,from the next closest cluster as the average distance between the example, $\boldsymbol{x}^{(i)}$, and all examples in the nearest cluster.
- 3. Calculate the silhouette, $\boldsymbol{s}^{(i)}$, as the difference between cluster cohesion and separation divided by the greater of the two, as shown here:
- $$\boldsymbol{x}^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\{b^{(i)}, a^{(i)}\}}$$

Clustering Analysis

- **Quantifying the quality of clustering via silhouette plots**
- The silhouette coefficient is bounded in the range –1 to 1. Based on the preceding equation, we can see that the silhouette coefficient is 0 if the cluster separation and cohesion are equal ($b^{(i)} = a^{(i)}$). Furthermore, we get close to an ideal silhouette coefficient of 1 if $b^{(i)} > a^{(i)}$, since $b^{(i)}$ quantifies how dissimilar an example is from other clusters, and $a^{(i)}$ tells us how similar it is to the other examples in its own cluster.
-

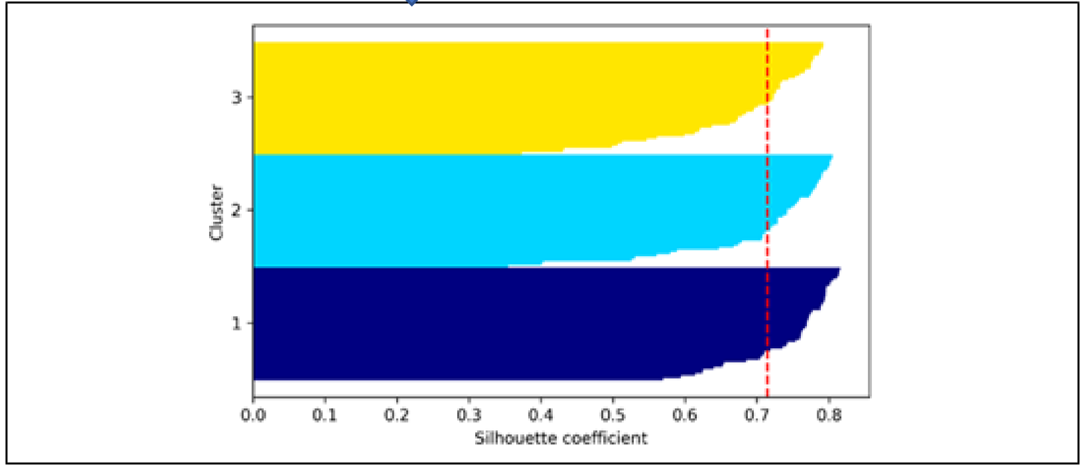
The silhouette coefficient is available as silhouette_samples from scikit-learn's metric module, and optionally, the silhouette_scores function can be imported for convenience. The silhouette_scores function calculates the average silhouette coefficient across all examples, which is equivalent to numpy.mean(silhouette_ samples(...)). By executing the following code, we will now create a plot of the silhouette coefficients for a k-means clustering with k = 3:

Clustering Analysis

- **Quantifying the quality of clustering via silhouette plots**

```
>>> km = KMeans(n_clusters=3,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> import numpy as np
 >>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...             y_km,
...             metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...             c_silhouette_vals = silhouette_vals[y_km == c]
...             c_silhouette_vals.sort()
...             y_ax_upper += len(c_silhouette_vals)
...             color = cm.jet(float(i) / n_clusters)
...             plt.barh(range(y_ax_lower, y_ax_upper),
...             c_silhouette_vals,
...             height=1.0,
...             edgecolor='none',
...             color=color)
...             yticks.append((y_ax_lower + y_ax_upper) / 2.)
...             y_ax_lower += len(c_silhouette_vals)
```

```
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...             color="red",
...             linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.tight_layout()
>>> plt.show()
```

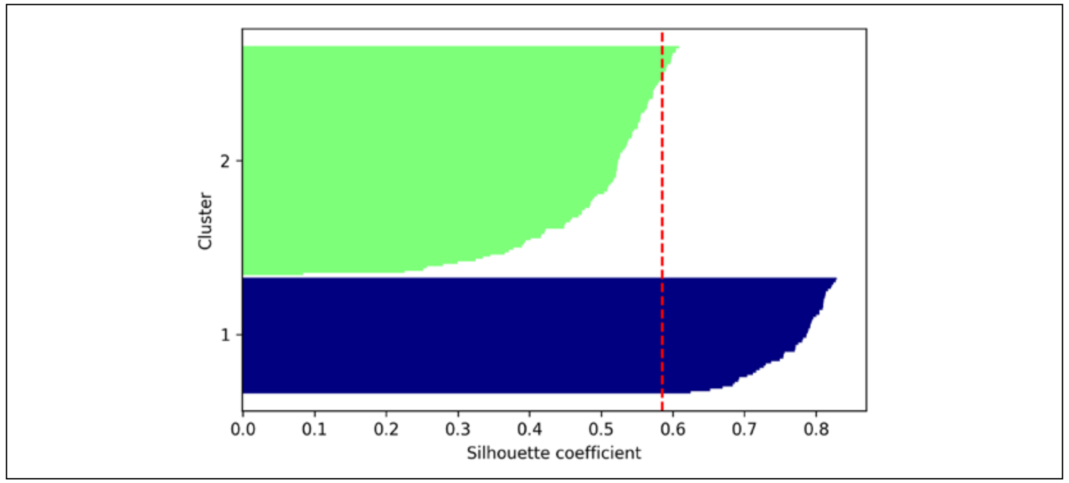


However, as you can see in the silhouette plot, the silhouette coefficients are not even close to 0, which is, in this case, an indicator of a good clustering. Furthermore, to summarize the goodness of our clustering, we added the average silhouette coefficient to the plot (dotted line).

Clustering Analysis

- **Quantifying the quality of clustering via silhouette plots**

```
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                 y_km,
...                 metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...                 c_silhouette_vals = silhouette_vals[y_km == c]
...                 c_silhouette_vals.sort()
...                 y_ax_upper += len(c_silhouette_vals)
...                 color = cm.jet(float(i) / n_clusters)
...                 plt.barh(range(y_ax_lower, y_ax_upper),
...                 c_silhouette_vals,
...                 height=1.0,
...                 edgecolor='none',
...                 color=color)
...                 yticks.append((y_ax_lower + y_ax_upper) / 2.)
...                 y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
 >>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.tight_layout()
>>> plt.show()
```

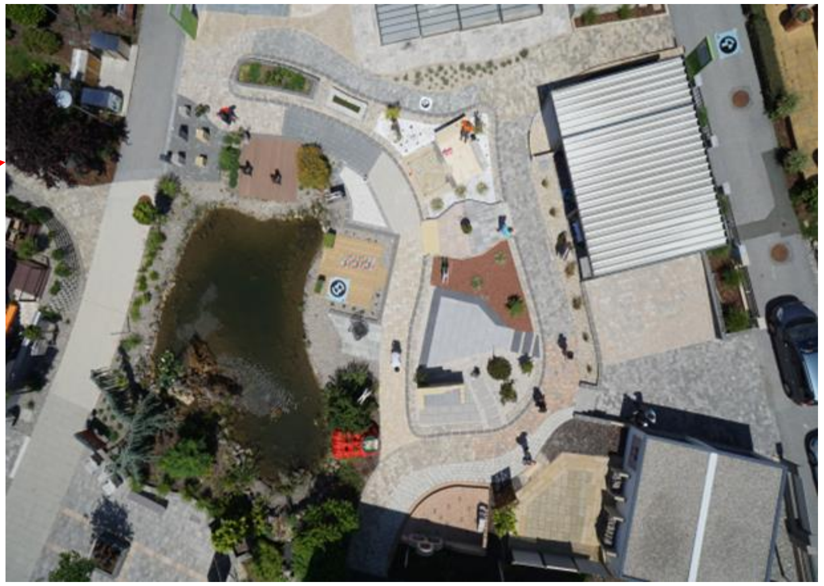


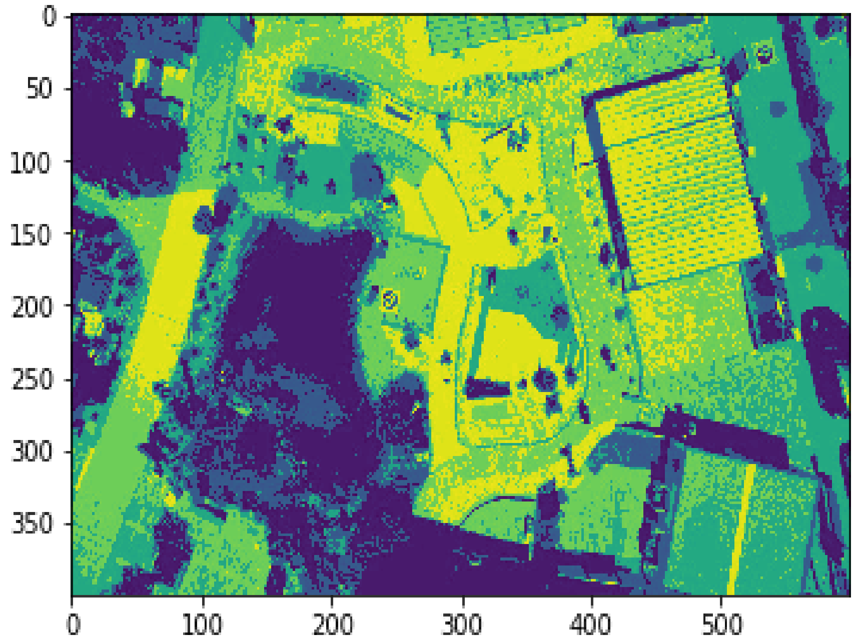
As you can see in the resulting plot, the silhouettes now have visibly different lengths and widths, which is evidence of a relatively *bad* or at least *suboptimal* clustering:

Clustering Segmentation

- **Segmentation using Kmeans clustering**

```
path='D:\\LAB4Segmentation\\LAB6\\12.png'
img = cv2.imread(path)

# Convert MxNx3 image into Kx3 where K=MxN
img2 = img.reshape((-1,3))  #-1 reshape means, in this case MxN

#We convert the unit8 values to float as it is a requirement of the k-means method of
OpenCV
img2 = np.float32(img2)

criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)

# Number of clusters
k = 5

attempts = 10

ret,label,center=cv2.kmeans(img2, k, None, criteria, attempts,
cv2.KMEANS_PP_CENTERS)

center = np.uint8(center)

#Next, we have to access the labels to regenerate the clustered image
res = center[label.flatten()]
res2 = res.reshape((img.shape)) #Reshape labels to the size of original image
cv2.imwrite("kmeansegmented.jpg", res2)
```

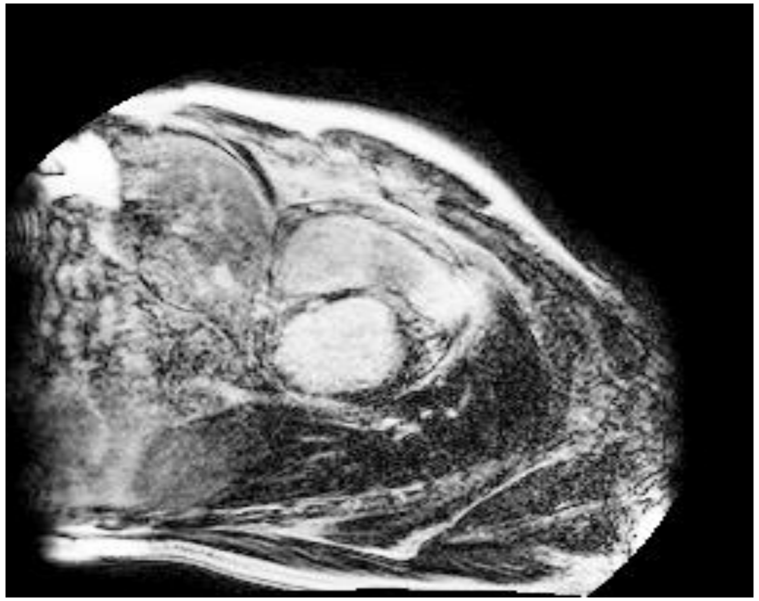


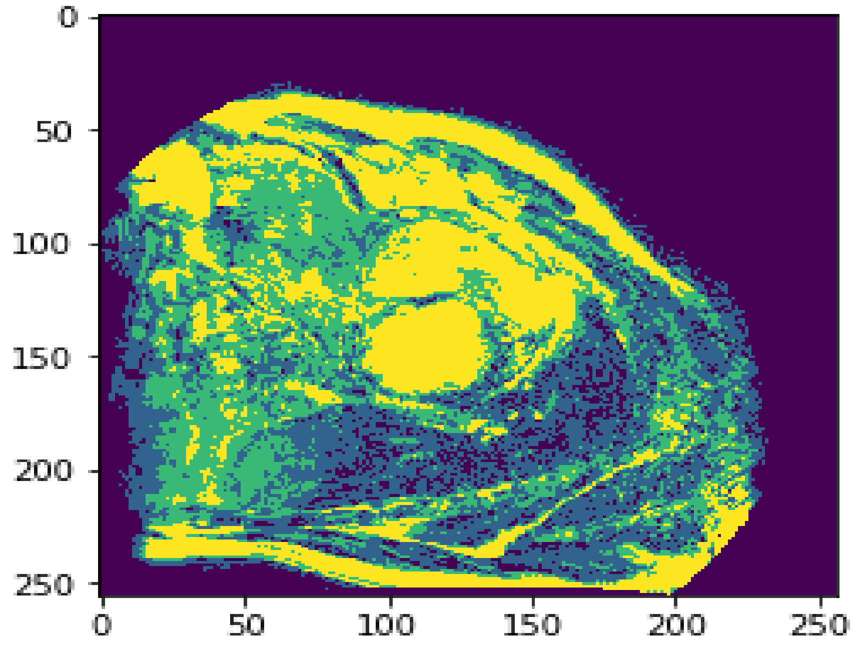


Clustering Segmentation

- **Segmentation using Kmeans clustering**

```
path='D:\\LAB4Segmentation\\LAB6\\12.png'
img = cv2.imread(path)

# Convert MxNx3 image into Kx3 where K=MxN
img2 = img.reshape((-1,3))  #-1 reshape means, in this case MxN

#We convert the unit8 values to float as it is a requirement of the k-means method of
OpenCV
img2 = np.float32(img2)

criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)

# Number of clusters
k = 5

attempts = 10

ret,label,center=cv2.kmeans(img2, k, None, criteria, attempts,
cv2.KMEANS_PP_CENTERS)

center = np.uint8(center)

#Next, we have to access the labels to regenerate the clustered image
res = center[label.flatten()]
res2 = res.reshape((img.shape)) #Reshape labels to the size of original image
cv2.imwrite("kmeansegmented.jpg", res2)
```





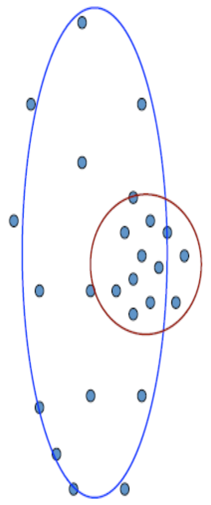
What is GMM?

- **GMM model**
- The k-means clustering model explored in the previous section is simple and relatively easy to understand, but its simplicity leads to practical challenges in its application.
- In particular, 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), which can be viewed as an extension of the ideas behind k-means, but can also be a powerful tool for estimation beyond simple clustering.
- As we saw in the previous section, given simple, well-separated data, k-means finds suitable clustering results.

What is GMM?

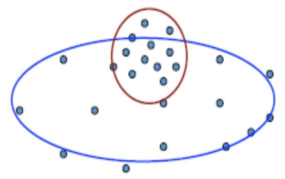
https://www.naftaliharris.com/blog/visualizing-k-means-clustering/

The Evils of "**Hard Assignments**"?



- Clusters may overlap
- Some clusters may be "wider" than others
- Distances can be deceiving!

Probabilistic Clustering



- Try a probabilistic model!
  - allows overlaps, clusters of different size, etc.
- Can tell a *generative story* for data
  - P(X|Y) P(Y)
- Challenge: we need to estimate model parameters without labeled Ys

Y	X_1	X_2
??	0.1	2.1
??	0.5	-1.1
??	0.0	3.0
??	-0.1	-2.0
??	0.2	1.5
…	…	…

What is GMM?

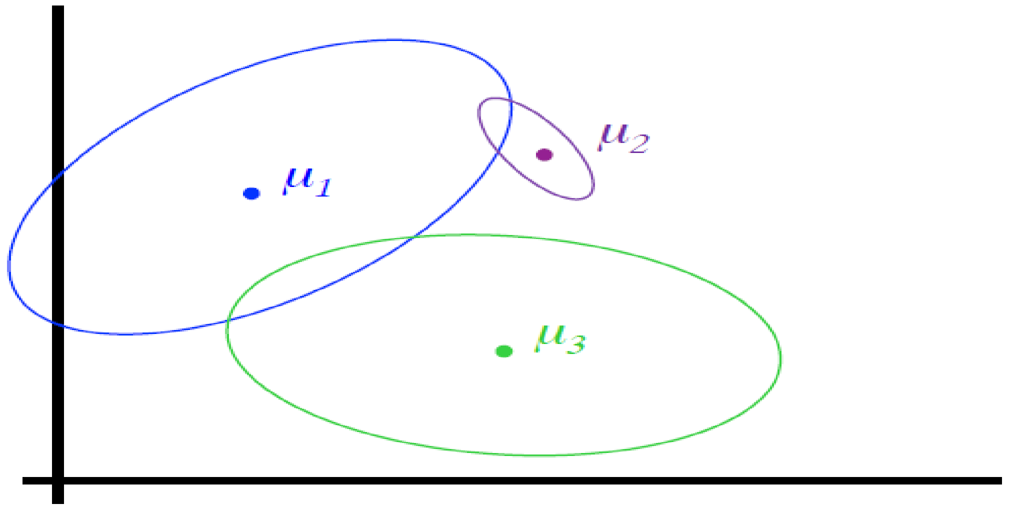
The General GMM assumption

- P(Y): There are k components
- P(X|Y): Each component generates data from a **multivariate Gaussian** with mean μ_i and covariance matrix Σ_i

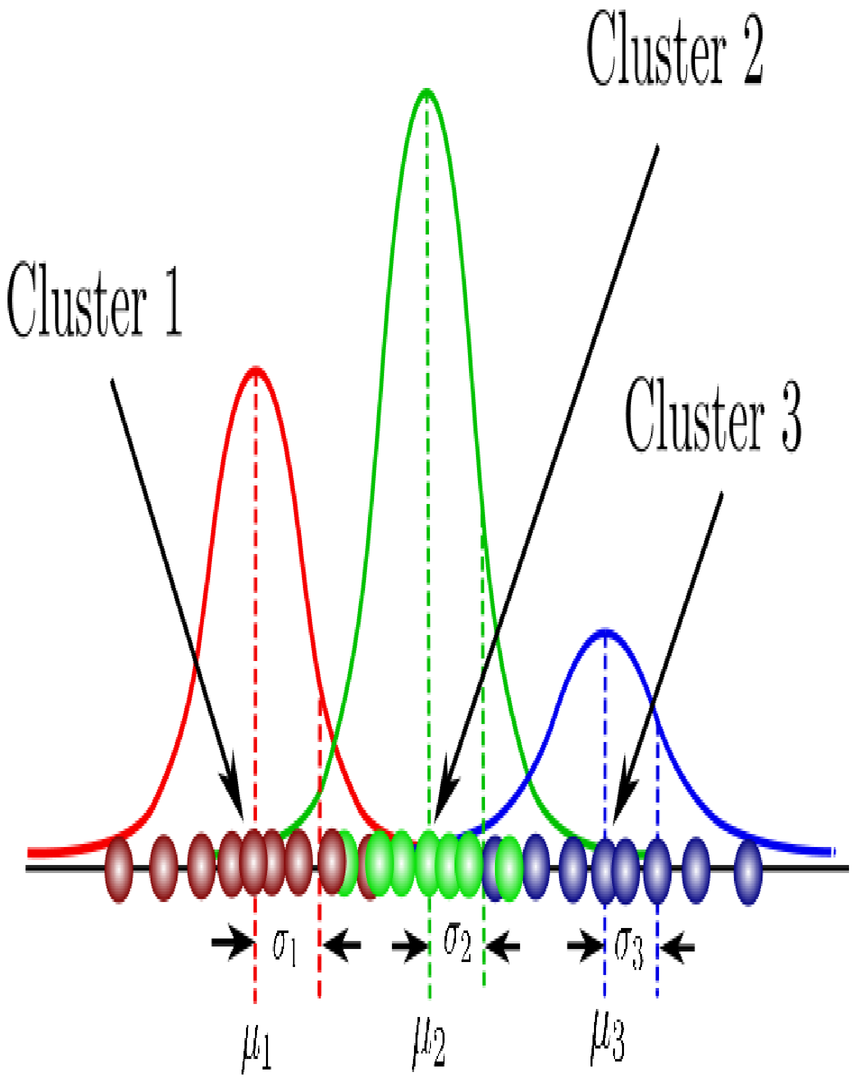
Each data point is sampled from a ***generative process***:

1. Choose component i with probability *P(y=i)*
2. Generate datapoint ~ N(m_i, Σ_i)

Gaussian mixture model (GMM)



What is GMM?



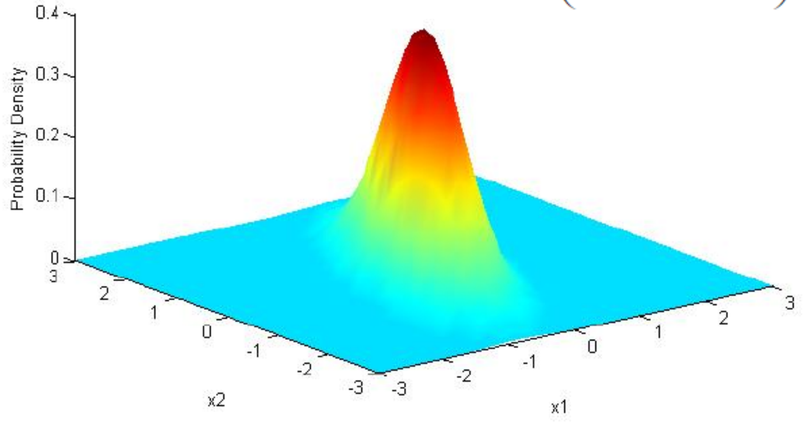
What is a Gaussian?

For ***d* dimensions**, the Gaussian distribution of a vector $x = (x^1, x^2, \ldots, x^d)^T$ is defined by:

$$N(x \mid \mu, \Sigma) = \frac{1}{(2\pi)^{d/2}\sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right)$$

where μ is the mean and Σ is the covariance matrix of the Gaussian.

Example: $\mu = (0,0)^T$ $\Sigma = \begin{pmatrix} 0.25 & 0.30 \\ 0.30 & 1.00 \end{pmatrix}$



What is GMM?

What is a Gaussian mixture model?

The probability given in a mixture of *K* Gaussians is:

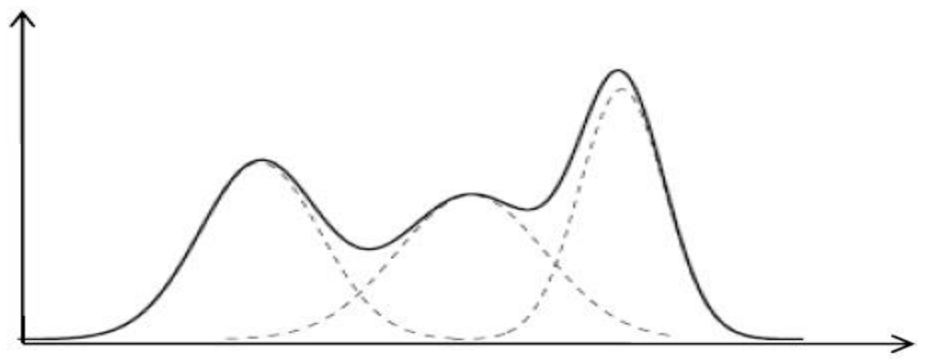
$$p(x) = \sum_{j=1}^{K} w_j \cdot N(x \mid \mu_j, \Sigma_j)$$

where w_j is the prior probability (weight) of the jth Gaussian.

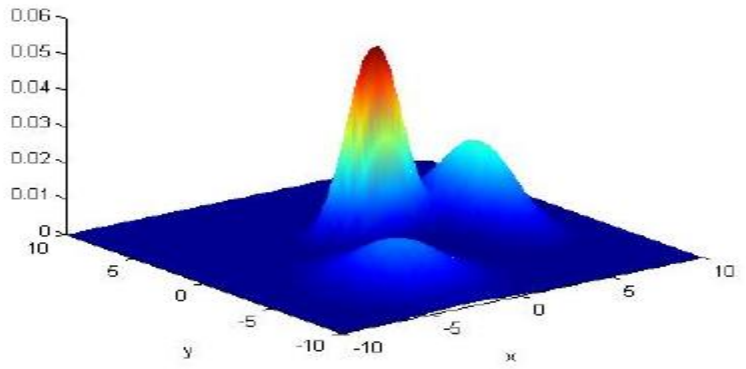
$$\sum_{j=1}^{K} w_j = 1 \qquad \text{and} \qquad 0 \le w_j \le 1$$

Examples:

d=1:



d=2:



What is GMM?

What is a Gaussian mixture model?

- **Problem:**

  Given a set of data $X = \{x_1, x_2, \ldots, x_N\}$ drawn from an unknown distribution (probably a GMM), estimate the parameters θ of the GMM model that fits the data.

- **Solution:**

  Maximize the likelihood $p(X \mid \theta)$ of the data with regard to the model parameters?

$$\boxed{\theta^* = \arg\max_{\theta} p(X \mid \theta) = \arg\max_{\theta} \prod_{i=1}^{N} p(x_i \mid \theta)}$$

What is GMM?

μ_k: mean
σ_k: covariance

A linear superposition of K-Gaussians

$$p(\mathrm{x}_i) = \sum_{k=1}^{K} \underbrace{\pi_k}_{p(k)} \underbrace{\mathcal{N}(\mathrm{x}_i|\mu_k, \sigma_k)}_{p(\mathrm{x}_i|k)}, \quad i = 1, \ldots, N$$

is called a **Gaussian mixture (GM)**. The mixture coefficient π_k satisfies

$$\sum_{k=1}^{K} \pi_k = 1, \qquad 0 \leq \pi_k \leq 1$$

Interpretation: The density $p(\mathrm{x}|k) = \mathcal{N}(\mathrm{x}|\mu_k, \sigma_k)$ is the probability of x, given that component k was chosen. The probability of choosing component k is given by the prior probability $p(k)$.

What is GMM?

The form of the GM distribution is governed by the parameters $\boldsymbol{\pi}, \boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. One way to get them is by **maximum likelihood**.

Given N observations $\{\mathrm{x}_n\}_{n=1}^{N}$, the log-likelihood function is

$$\ln p\left(X; \pi_{1:K}, \mu_{1:K}, \sigma_{1:K}\right) = \sum_{n=1}^{N} \ln\left(\sum_{k=1}^{K} \pi_k \mathcal{N}\left(\mathrm{x}_n | \mu_k, \sigma_k\right)\right)$$

There is **no closed-form solution** available (due to the sum inside the logarithm).

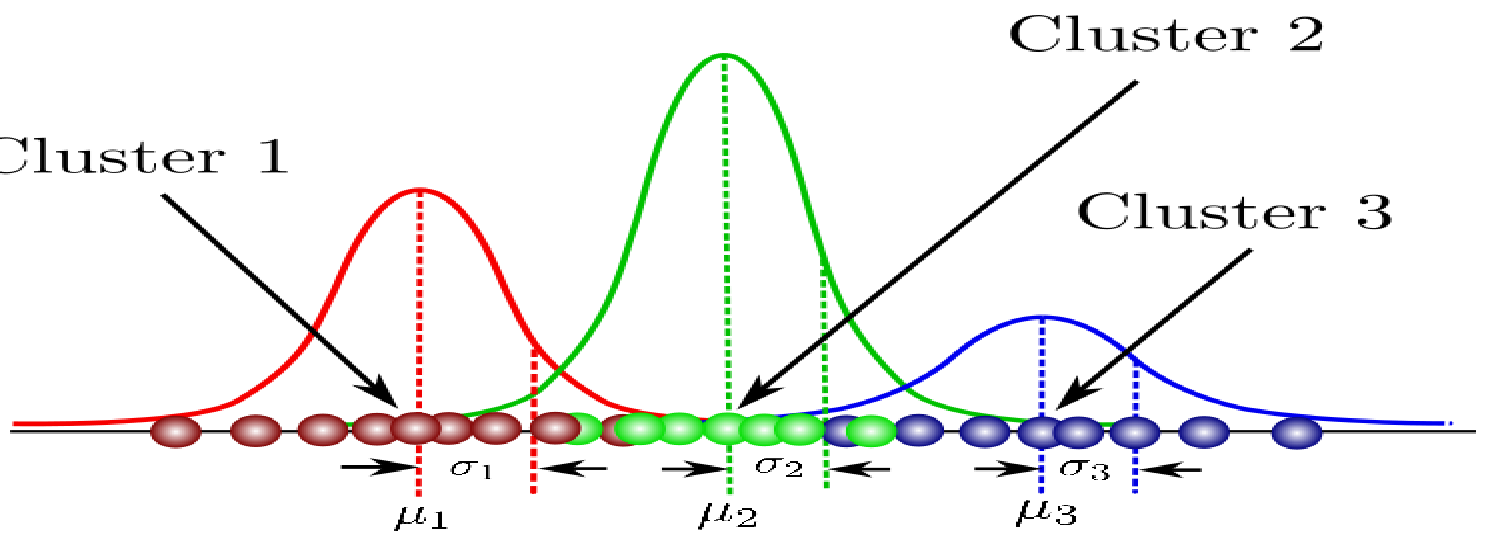
This problem can be separated into two simple problems using the *expectation-maximization (EM)* algorithm.

What is GMM?

A Gaussian Mixture is a function that is comprised of several Gaussians, each identified by k ∈ {1,…, K}, where K is the number of clusters of our dataset.

Each Gaussian k in the mixture is comprised of the following parameters:

- A mean μ that defines its centre.
- A covariance Σ that defines its width. This would be equivalent to the dimensions of an ellipsoid in a multivariate scenario. A mixing probability π that defines how big or small the Gaussian function will be.

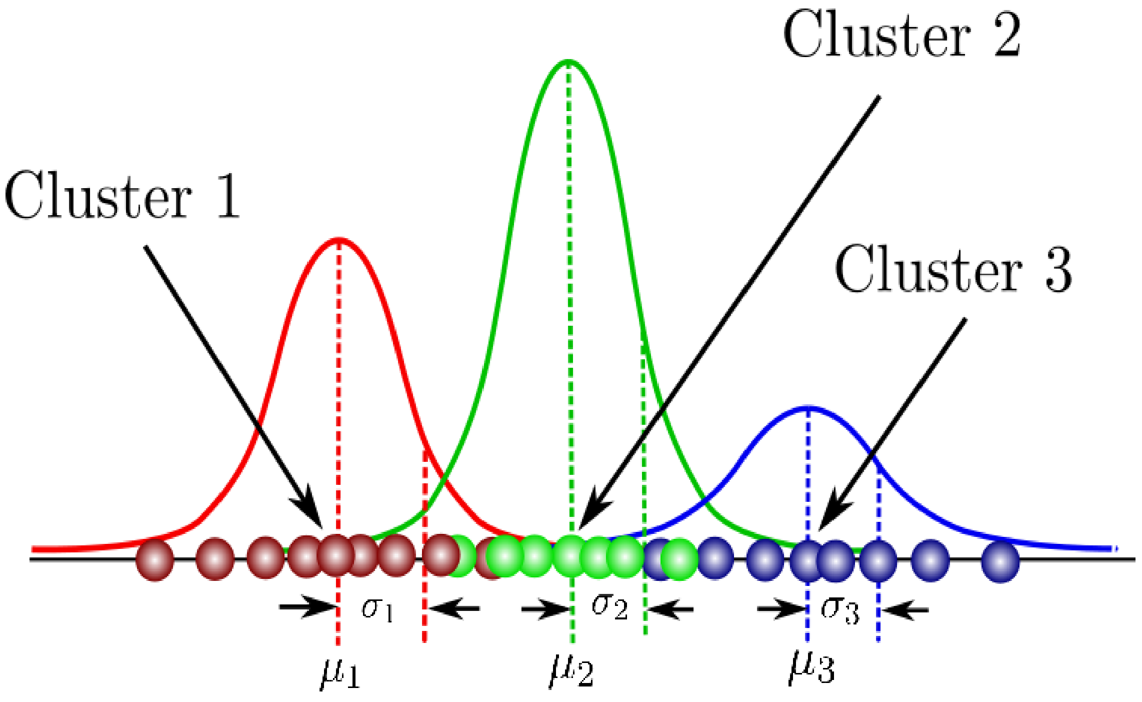


What is GMM?

Here, we can see that there are three Gaussian functions, hence K = 3. Each Gaussian explains the data contained in each of the three clusters available.
The mixing coefficients are themselves probabilities
and must meet this condition:

$$\sum_{k=1}^{K} \pi_k = 1 \qquad (1)$$

Now how do we determine the optimal values for these parameters? To achieve this we must ensure that each Gaussian fits the data points belonging to each cluster. This is exactly what maximum likelihood does.



What is GMM?

Now how do we determine the optimal values for these parameters? To achieve this we must ensure that each Gaussian fits the data points belonging to each cluster. This is exactly what maximum likelihood does.

In general, the Gaussian density function is given by:

$$\mathcal{N}(\mathbf{x}|\mu,\Sigma) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right)$$

Where x represents our data points, D is the number of dimensions of each data point. μ and Σ are the mean and covariance, respectively. If we have a dataset comprised of N = 1000 three-dimensional points (D = 3), then x will be a 1000 × 3 matrix. μ will be a 1 × 3 vector, and Σ will be a 3 × 3 matrix. For later purposes, we will also find it useful to take the log of this equation, which is given by:

$$\ln\mathcal{N}(\mathbf{x}|\mu,\Sigma) = -\frac{D}{2}\ln 2\pi - \frac{1}{2}\ln\Sigma - \frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu) \qquad (2)$$

What is GMM?

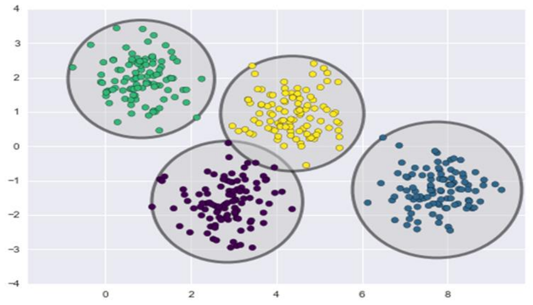
$$\mathcal{N}(\mathbf{x}|\mu,\Sigma) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right)$$

For later purposes, we will also find it useful to take the log of this equation, which is given by:

$$\ln\mathcal{N}(\mathbf{x}|\mu,\Sigma) = -\frac{D}{2}\ln 2\pi - \frac{1}{2}\ln\Sigma - \frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu) \qquad (2)$$

If we differentiate this equation with respect to the mean and covariance and then equate it to zero, then we will be able to find the optimal values for these parameters, and the solutions will correspond to the Maximum Likelihood Estimates (MLE) for this setting. However, because we are dealing with not just one, but many Gaussians, things will get a bit complicated when time comes for us to find the parameters for the whole mixture.

What is GMM?



Initial derivations

First, let’s suppose we want to know what is the probability that a data point xn comes from Gaussian k. We can express this as:

$$p(z_{nk} = 1|\mathbf{x}_n)$$

Which reads “given a data point x, what is the probability it came from Gaussian k?” In this case, z is a latent variable that takes only two possible values. It is one when x came from Gaussian k, and zero otherwise. Actually, we don’t get to see this z variable in reality, but knowing its probability of occurrence will be useful in helping us determine the Gaussian mixture parameters, as we discuss later.

What is GMM?

Likewise, we can state the following:

$$\pi_k = p(z_k = 1)$$

Which means that the overall probability of observing a point that comes from Gaussian k is actually equivalent to the mixing coefficient for that Gaussian. This makes sense, because the bigger the Gaussian is, the higher we would expect this probability to be.

For example, consider the following GMM:

$$p(x) = \underbrace{0.3}_{\pi_1} \mathcal{N}\left(x \mid \underbrace{\begin{pmatrix} 4 \\ 4.5 \end{pmatrix}}_{\mu_1}, \underbrace{\begin{pmatrix} 1.2 & 0.6 \\ 0.6 & 0.5 \end{pmatrix}}_{\Sigma_1}\right) + \underbrace{0.5}_{\pi_2} \mathcal{N}\left(x \mid \underbrace{\begin{pmatrix} 8 \\ 1 \end{pmatrix}}_{\mu_2}, \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}}_{\Sigma_2}\right) + \underbrace{0.2}_{\pi_3} \mathcal{N}\left(x \mid \underbrace{\begin{pmatrix} 9 \\ 8 \end{pmatrix}}_{\mu_3}, \underbrace{\begin{pmatrix} 0.6 & 0.5 \\ 0.5 & 1.5 \end{pmatrix}}_{\Sigma_3}\right)$$

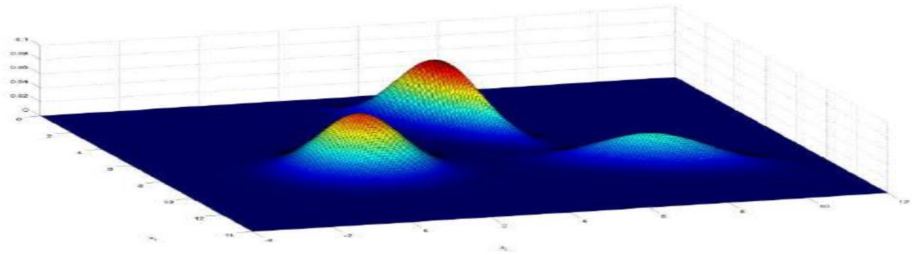


Figure: Probability density function.

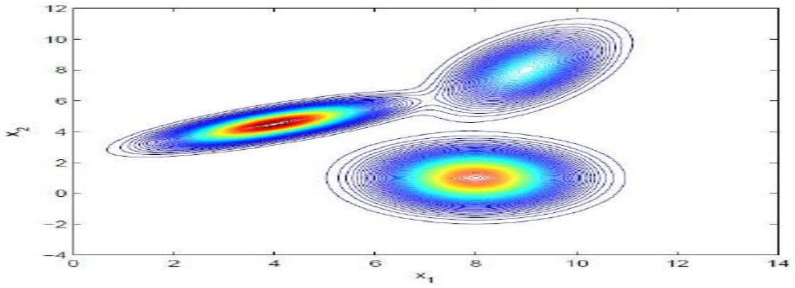


Figure: Contour plot.

What is GMM?

https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95

Now let z be the set of all possible latent variables z, hence:

$$\mathbf{z} = \{z_1, ..., z_K\}$$

We know beforehand that each z occurs independently of others and that they can only take the value of one when k is equal to the cluster the point comes from. Therefore:

$$p(\mathbf{z}) = p(z_1 = 1)^{z_1} p(z_2 = 1)^{z_2} ... p(z_K = 1)^{z_K} = \prod_{k=1}^{K} \pi_k^{z_k}$$

Now, what about finding the probability of observing our data given that it came from Gaussian k? Turns out to be that it is actually the Gaussian function itself! Following the same logic we used to define p(z), we can state:

$$p(\mathbf{x}_n|\mathbf{z}) = \prod_{k=1}^{K} \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k)^{z_k}$$

What is GMM?

https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95

Remember our initial aim was to determine what the probability of z given our observation x? Well, it turns out to be that the equations we have just derived, along with the Bayes rule, will help us determine this probability. From the product rule of probabilities, we know that

$$p(\mathbf{x}_n, \mathbf{z}) = p(\mathbf{x}_n|\mathbf{z})p(\mathbf{z})$$

$$p(\mathbf{x}_n) = \sum_{k=1}^{K} p(\mathbf{x}_n|\mathbf{z})p(\mathbf{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)$$

What is GMM?

https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95

This is the equation that defines a Gaussian Mixture, and you can clearly see that it depends on all parameters.

To determine the optimal values for these we need to determine the maximum likelihood of the model. We can find the likelihood as the joint probability of all observations xn, defined by:

$$p(\mathbf{x}_n) = \sum_{k=1}^{K} p(\mathbf{x}_n|\mathbf{z})p(\mathbf{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)$$

$$p(\mathbf{X}) = \prod_{n=1}^{N} p(\mathbf{x}_n) = \prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)$$

What is GMM?

https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95

Like we did for the original Gaussian density function, let's apply the log to each side of the equation:

$$\ln p(\mathbf{X}) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n | \mu_k, \Sigma_k) \qquad (3)$$

But first, remember we were supposed to find the probability of z given x? Well, let's do that since at this point we already have everything in place to define what this probability will look like.

$$p(z_k = 1 | \mathbf{x}_n) = \frac{p(\mathbf{x}_n | z_k = 1) p(z_k = 1)}{\sum_{j=1}^{K} p(\mathbf{x}_n | z_j = 1) p(z_j = 1)}$$

What is GMM?

https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95

But first, remember we were supposed to find the probability of z given x? Well, let's do that since at this point we already have everything in place to define what this probability will look like.

$$p(z_k = 1|\mathbf{x}_n) = \frac{p(\mathbf{x}_n|z_k = 1)p(z_k = 1)}{\sum_{j=1}^{K} p(\mathbf{x}_n|z_j = 1)p(z_j = 1)}$$

From our earlier derivations we learned that:

$$p(z_k = 1) = \pi_k, \qquad p(\mathbf{x}_n|z_k = 1) = \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k)$$

$$p(z_k = 1|\mathbf{x}_n) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\mu_j, \Sigma_j)} = \gamma(z_{nk}) \qquad (4)$$

Expectation Maximization in GMM?

Algorithm 1 EM for Gaussian mixtures

```
1: Initialize $\mu_k^1, \sigma_k^1, \pi_k^1$ and set $i = 1$.
2: while not converged do
3:     Compute $\gamma(z_{nk})$.                          ▷ Expectation step
4:     Compute $\mu_k^{i+1}; \pi_k^{i+1}; N_k; \sigma_k^{i+1}$.       ▷ Maximization step
5:     $i \leftarrow i + 1$.
6: end while
```

$$\gamma(z_{nk}) = \frac{\pi_k^i \mathcal{N}(\mathrm{x}_n | \mu_k^i, \sigma_k^i)}{\sum_{j=1}^{K} \pi_j^i \mathcal{N}(\mathrm{x}_n | \mu_j^i, \sigma_j^i)}, n = 1, \ldots, N; k = 1, \ldots, K$$

$$\mu_k^{i+1} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathrm{x}_n,$$

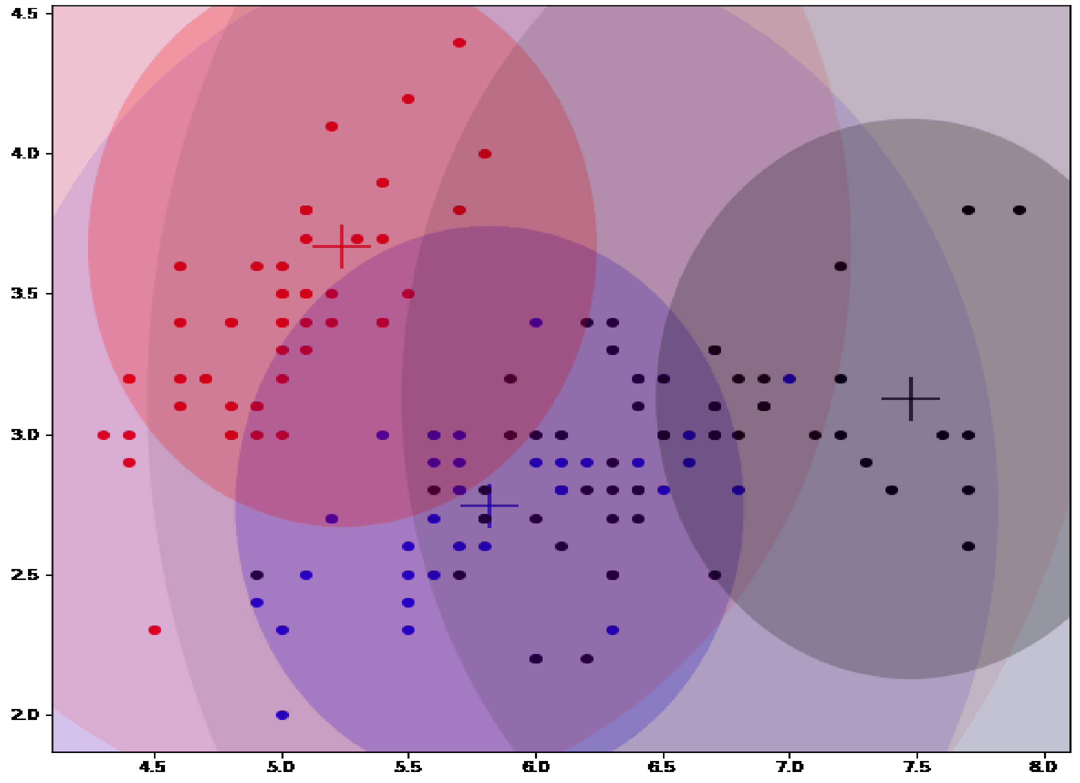
$$\pi_k^{i+1} = \frac{N_k}{N}, \quad N_k = \sum_{n=1}^{N} \gamma(z_{nk}),$$

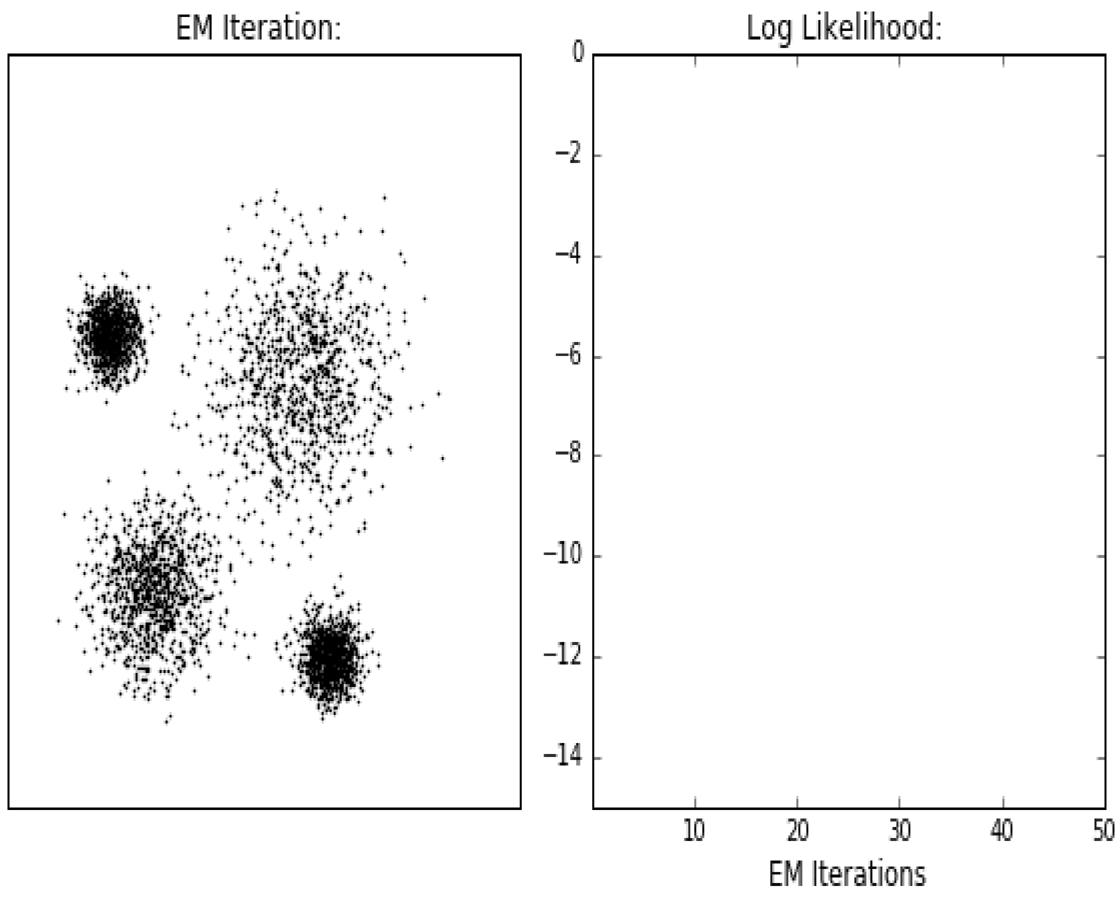
$$\sigma_k^{i+1} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \left(\mathrm{x}_n - \mu_k^{i+1}\right) \left(\mathrm{x}_n - \mu_k^{i+1}\right)^T.$$

Expectation Maximization in GMM?

Expectation — Maximization algorithm

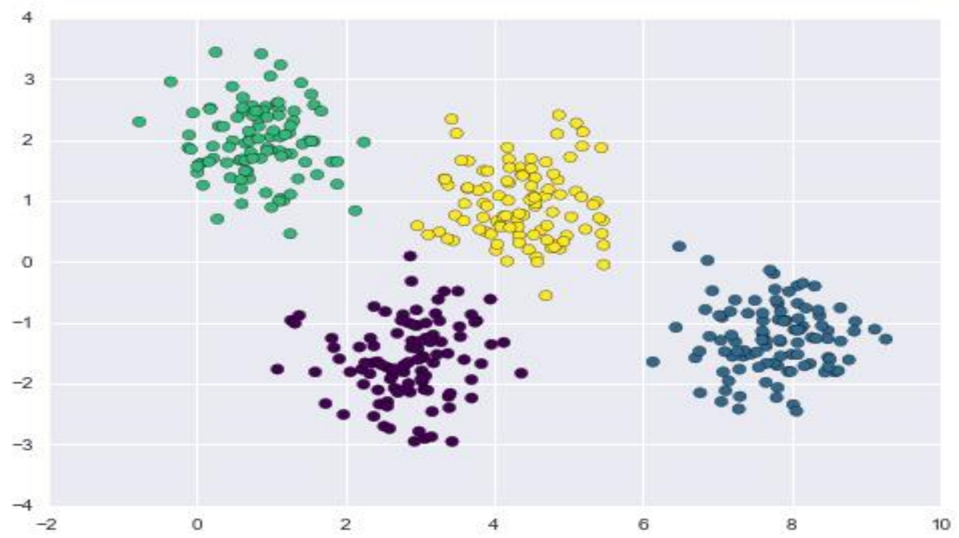
https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95





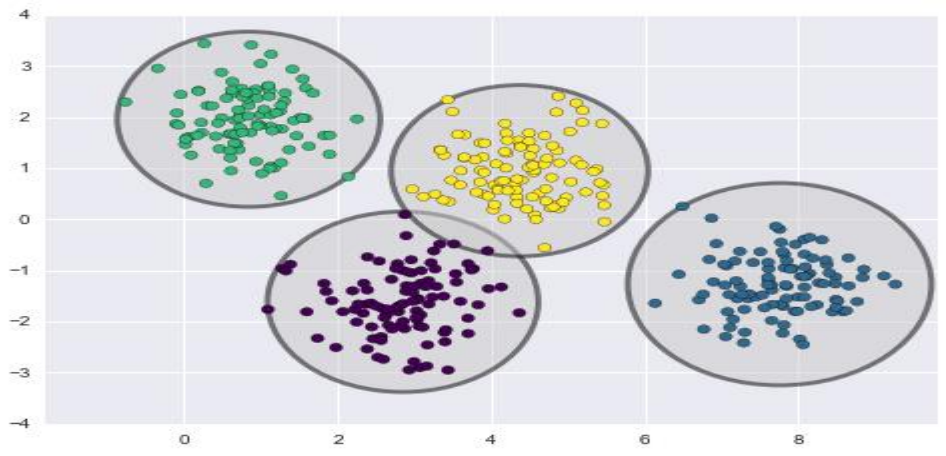
GMM Implementation

- **GMM model**
- From an intuitive standpoint, we might 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 assigment of points between them.
- Unfortunately, the k-means model has no intrinsic measure of probability or uncertainty of cluster assignments (although it may be possible to use a bootstrap approach to estimate this uncertainty).
- For this, we must think about generalizing the model.



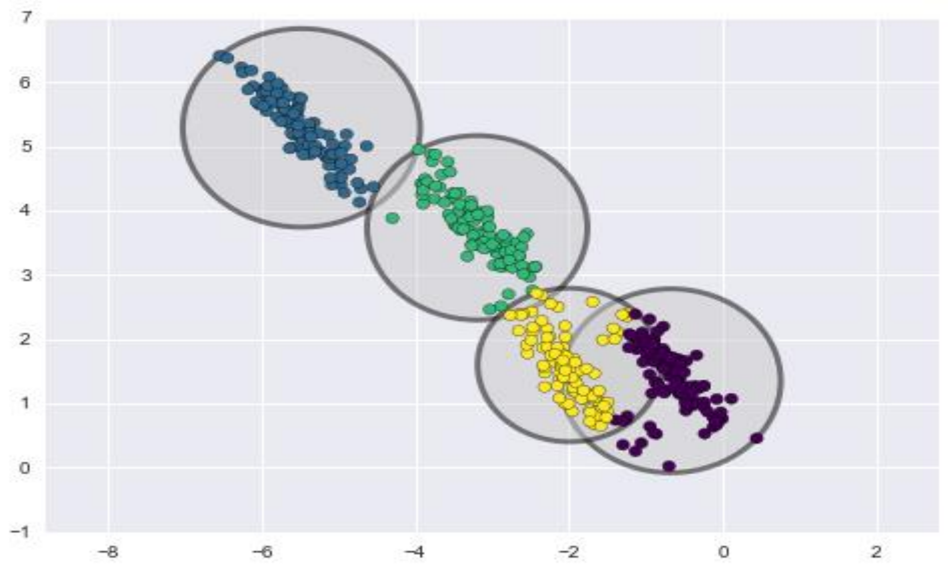
GMM Implementation

- **GMM model**
- 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.**
- 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:



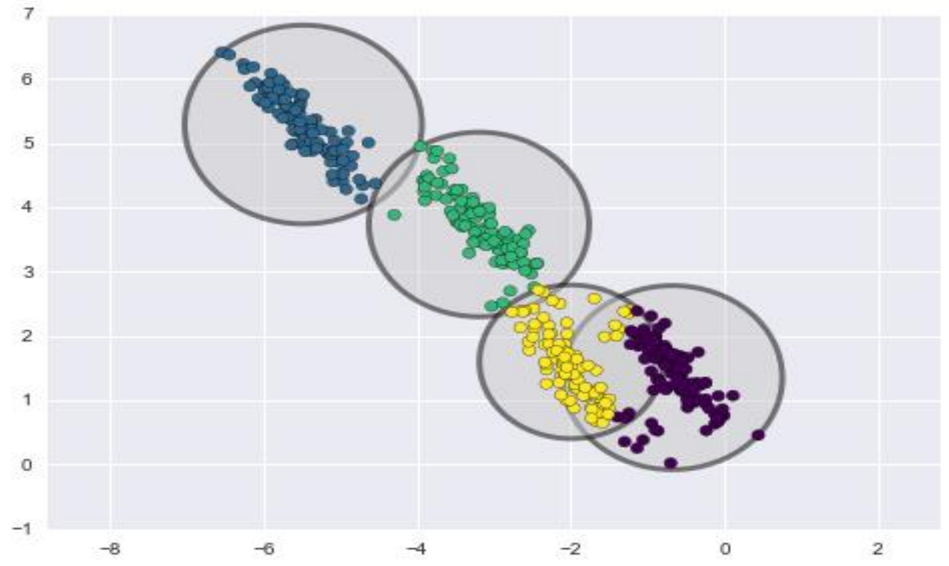
GMM Implementation

- **GMM model**
- 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.**
- These two disadvantages of k-means—its lack of flexibility in **cluster shape and lack of probabilistic cluster assignment**—mean that for many datasets (especially low-dimensional datasets) it may not perform as well as you might hope.



GMM Implementation

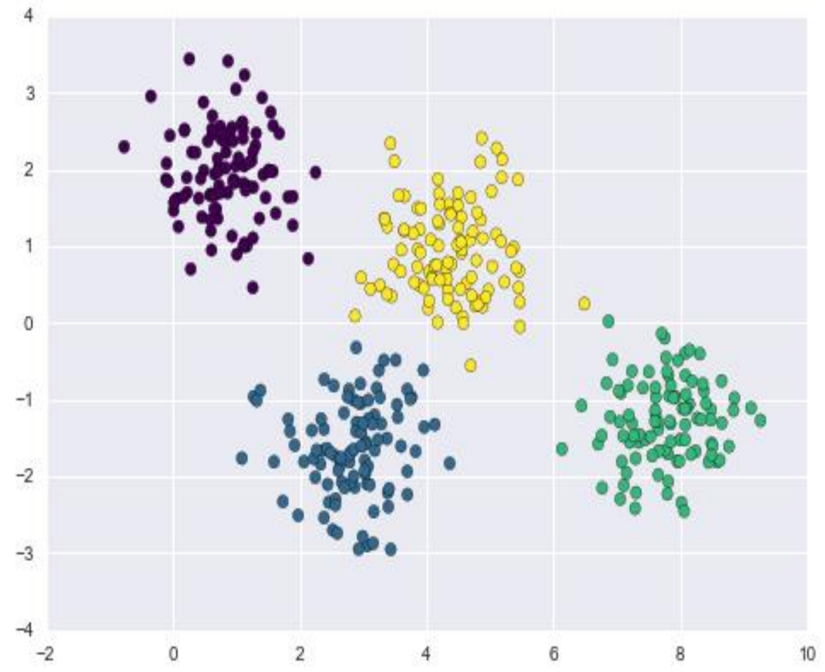
- **GMM model**
- You might imagine addressing these weaknesses by generalizing the k-means model: for example, you could measure uncertainty in cluster assignment by comparing the distances of each point to all cluster centers, rather than focusing on just the closest. You might also imagine allowing 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.**



GMM Implementation

- **Generalizing E–M: Gaussian Mixture Models**
- A Gaussian mixture model (GMM) 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:.

```
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');
```



GMM Implementation

- **Generalizing E–M: Gaussian Mixture Models**
- GMM contains a probabilistic model under the hood, 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.   ]]
```

```
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');
```

Generalizing E–M: Gaussian Mixture Models

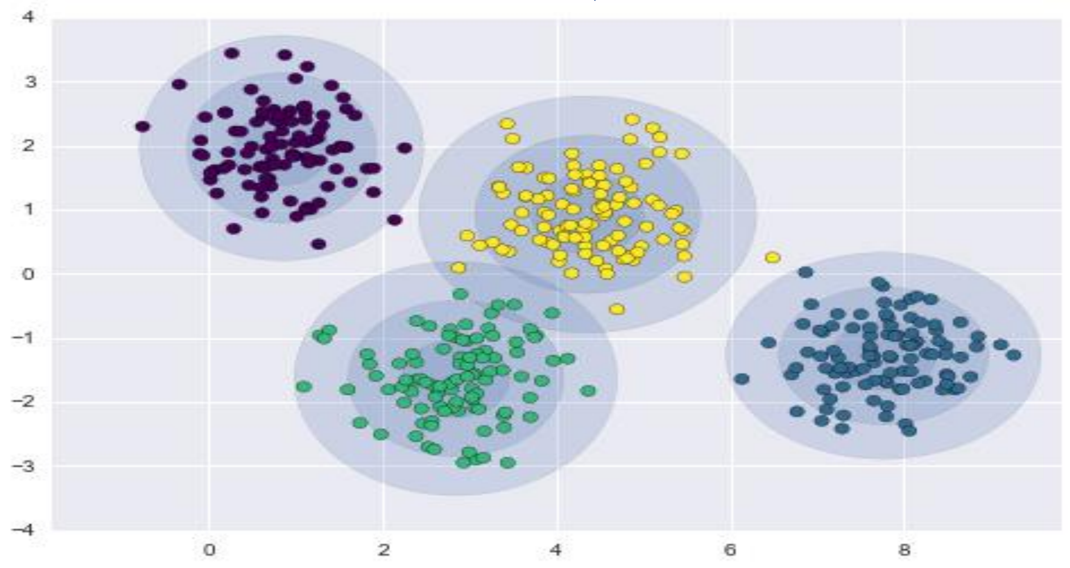
- **Generalizing E–M: Gaussian Mixture Models**
- Under the hood, a Gaussian mixture model is very similar to k-means: it uses an expectation–maximization approach which qualitatively does the following:
- Choose starting guesses for the location and shape
- Repeat until converged:
- **E-step:** for 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.**
- 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:

GMM Implementation

- **Generalizing E–M: Gaussian Mixture Models**
- 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:

```
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)
```

```
gmm = GMM(n_components=4, random_state=42)
plot_gmm(gmm, X)
```

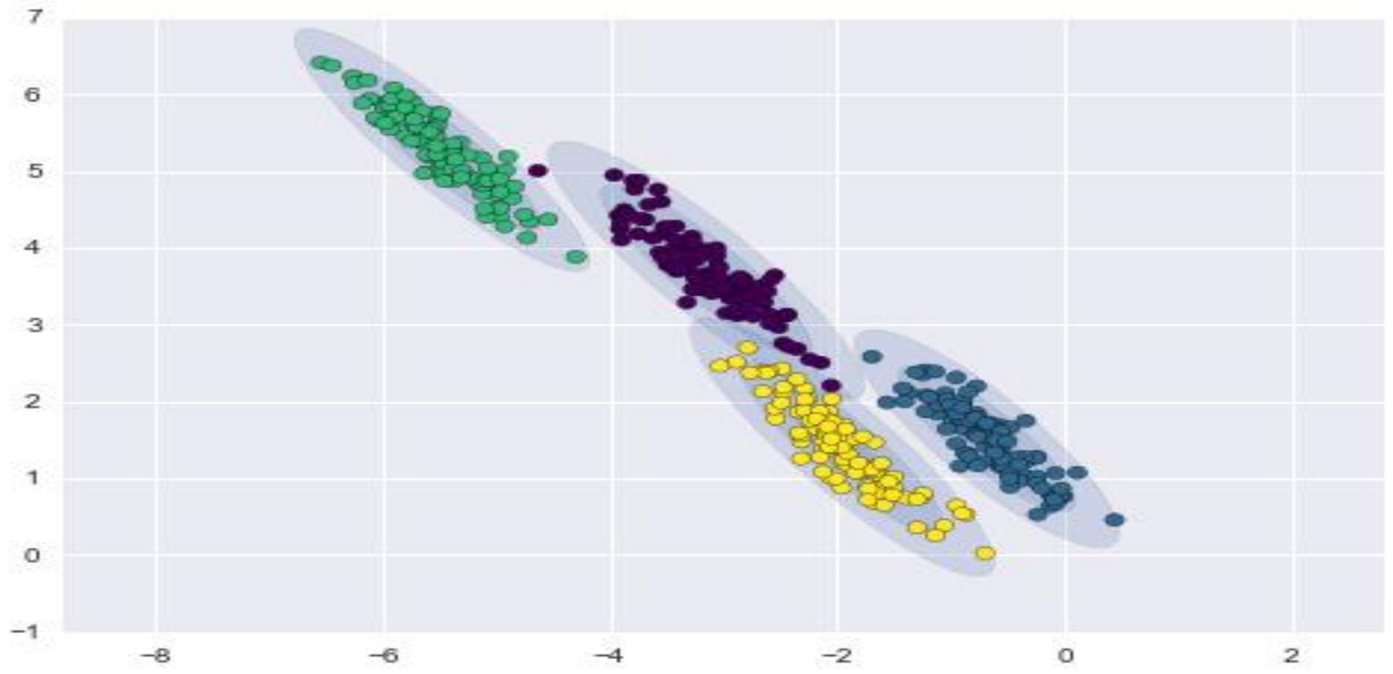


With this in place, we can take a look at what the four-component GMM gives us for our initial data

GMM Implementation

- **Generalizing E–M: Gaussian Mixture Models**
- 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:

```
gmm = GMM(n_components=4, covariance_type='full',
random_state=42)
plot_gmm(gmm, X_stretched)
```



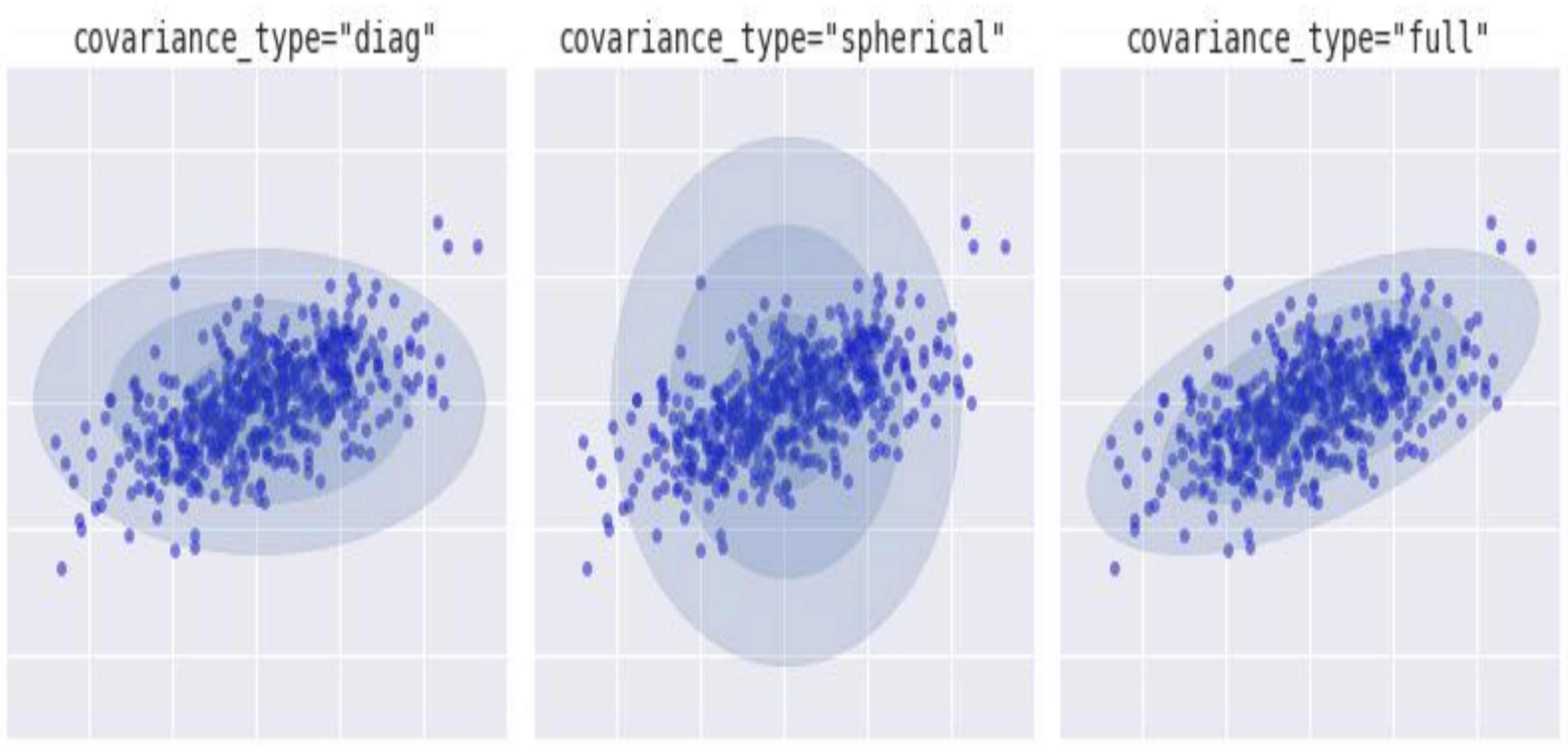
Choosing the covariance type in GMM

- **Generalizing E–M: Gaussian Mixture Models**
- 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.

Choosing the covariance type in GMM

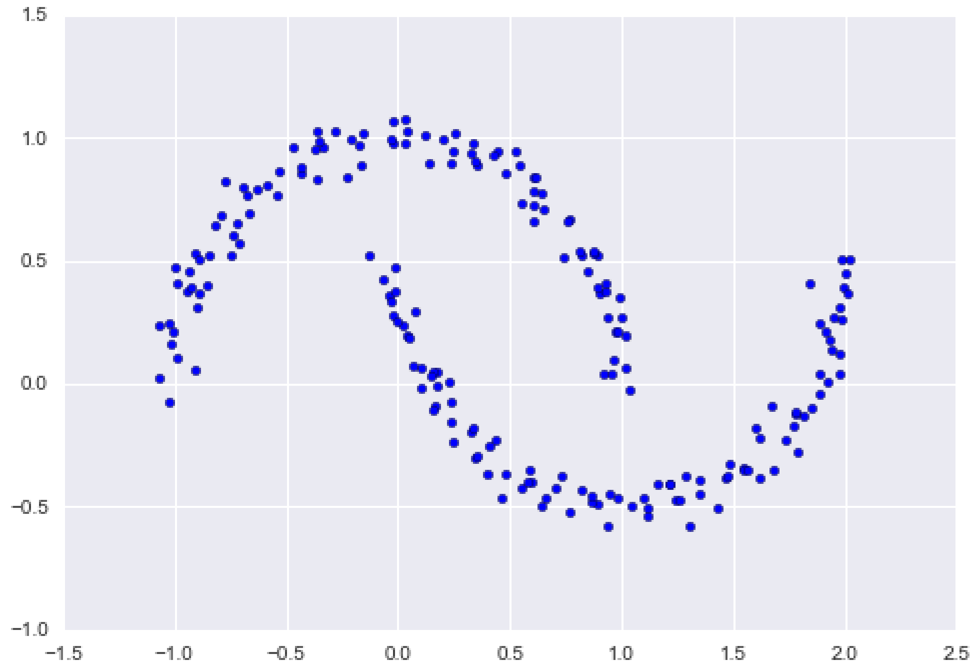
- **Generalizing E–M: Gaussian Mixture Models**
- Choosing the covariance type
- We can see a visual representation of these three choices for a single cluster within the following figure:



GMM as Density Estimation

- **GMM as Density Estimation**
- 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.

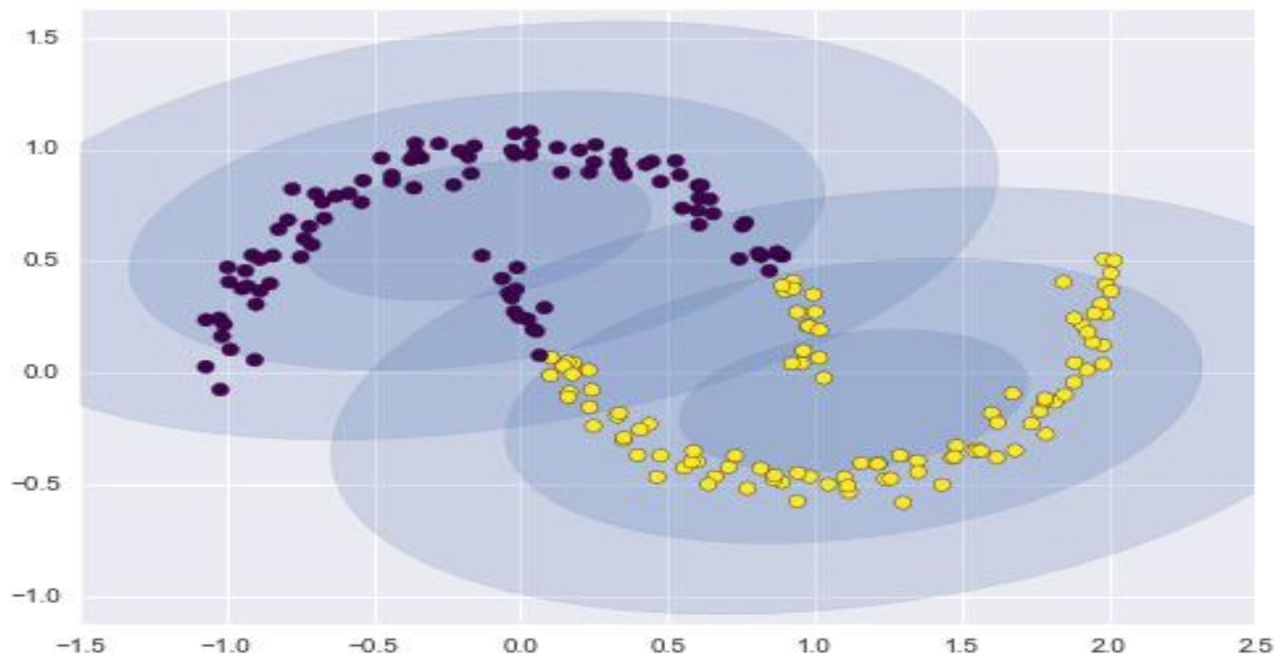
```
from sklearn.datasets import make_moons
Xmoon, ymoon = make_moons(200, noise=.05,
random_state=0)
plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);
```



GMM as Density Estimation

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

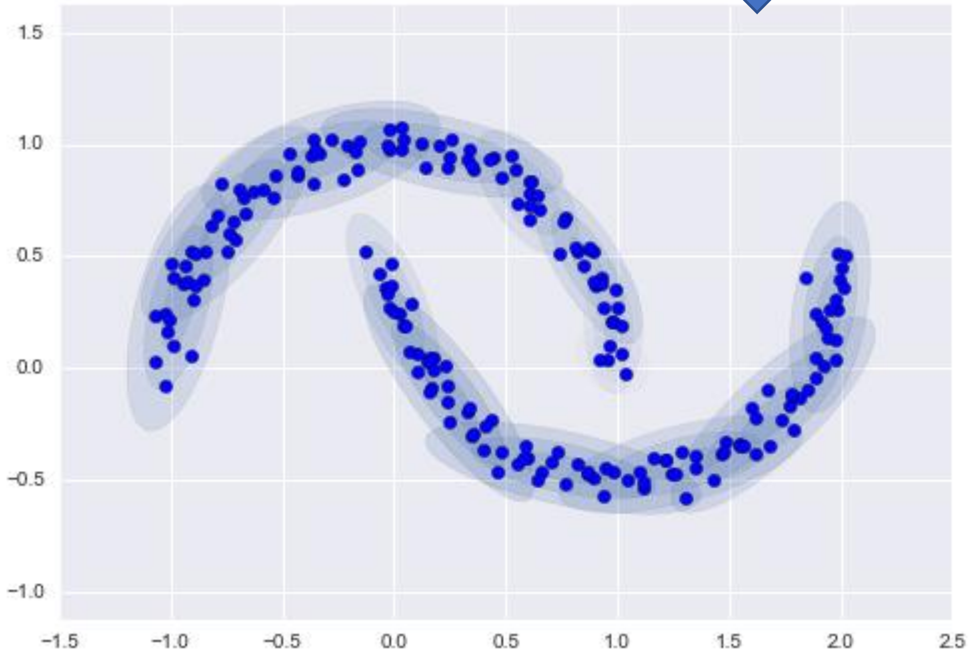
```
gmm2 = GMM(n_components=2, covariance_type='full',
random_state=0)
plot_gmm(gmm2, Xmoon)
```



GMM as Density Estimation

- **GMM as Density Estimation**
- GMM as Density Estimation
- But if we instead use many more components and ignore the cluster labels, we find a fit that is much closer to the input data:

```
gmm16 = GMM(n_components=16, covariance_type='full',
random_state=0)
plot_gmm(gmm16, Xmoon, label=False)
```

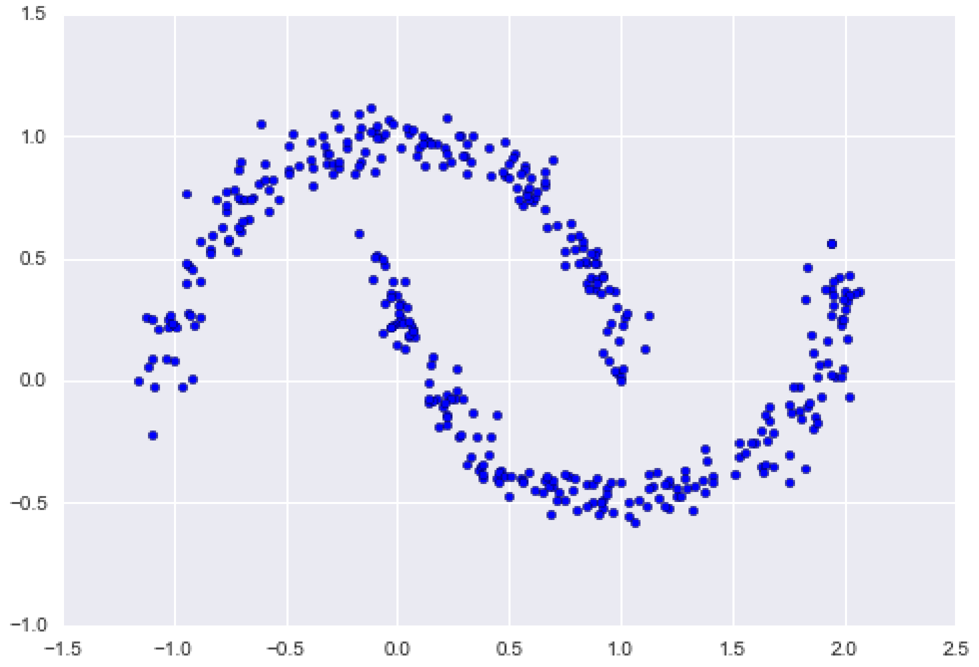


Here the mixture of 16 Gaussians serves not to find separated clusters of data, but rather to model the overall distribution of the input data. This is a generative model of the distribution, meaning that the GMM gives us the recipe to generate new random data distributed similarly to our input.

GMM as Density Estimation

- **GMM as Density Estimation**
- GMM as Density Estimation
- For example, here are 400 new points drawn from this 16-component GMM fit to our original data:

```
Xnew = gmm16.sample(400, random_state=42)
plt.scatter(Xnew[:, 0], Xnew[:, 1]);
```



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

How many components in GMM?

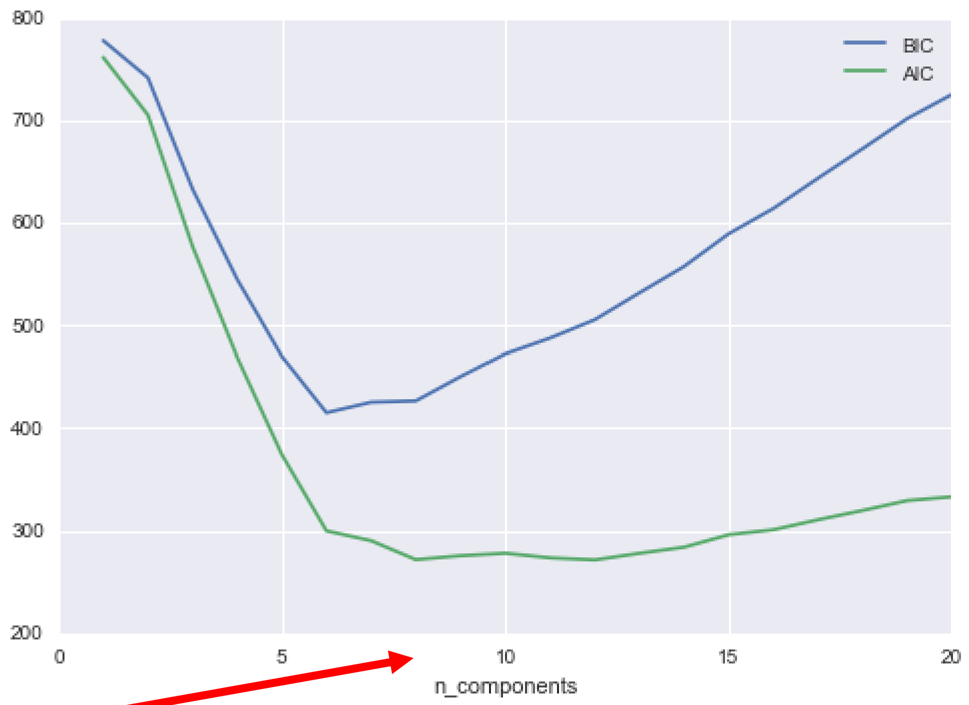
- **How many components?**
- The fact that GMM is a generative model gives us a natural means of determining the optimal number of components for a given dataset. A generative model is inherently a probability distribution for the dataset, and so we can simply evaluate the likelihood of the data under 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 our moon dataset:

GMM for Generating New Data

- **How many components?**
- Let's look at the AIC and BIC as a function as the number of GMM components for our moon dataset:

```
n_components = np.arange(1, 21)
models = [GMM(n, covariance_type='full',
random_state=0).fit(Xmoon)
          for n in n_components]

plt.plot(n_components, [m.bic(Xmoon) for m in models],
label='BIC')
plt.plot(n_components, [m.aic(Xmoon) for m in models],
label='AIC')
plt.legend(loc='best')
plt.xlabel('n_components');
```



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.

What is GMM?

- **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.
- To start with, let's load the digits data using Scikit-Learn's data tools:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
(1797, 64)
```

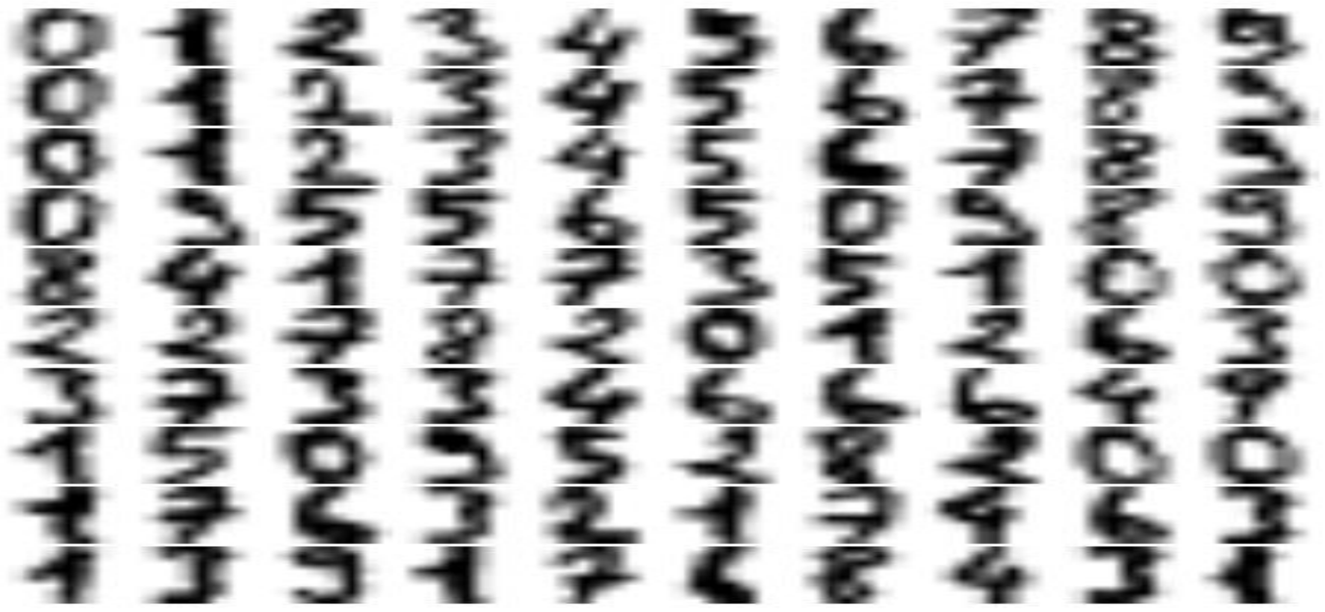
Next let's plot the first 100 of these to recall exactly what we're looking at:

```
def plot_digits(data):
    fig, ax = plt.subplots(10, 10, figsize=(8, 8),
                           subplot_kw=dict(xticks=[], yticks=[]))
    fig.subplots_adjust(hspace=0.05, wspace=0.05)
    for i, axi in enumerate(ax.flat):
        im = axi.imshow(data[i].reshape(8, 8), cmap='binary')
        im.set_clim(0, 16)
plot_digits(digits.data)
```

GMM for Generating New Data

- **Example: GMM for Generating New Data**
- Next let's plot the first 100 of these to recall exactly what we're looking at:

```
def plot_digits(data):
    fig, ax = plt.subplots(10, 10, figsize=(8, 8),
                           subplot_kw=dict(xticks=[], yticks=[]))
    fig.subplots_adjust(hspace=0.05, wspace=0.05)
    for i, axi in enumerate(ax.flat):
        im = axi.imshow(data[i].reshape(8, 8), cmap='binary')
        im.set_clim(0, 16)
plot_digits(digits.data)
```



GMM for Generating New Data

- **Example: GMM for Generating New Data**
- 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:

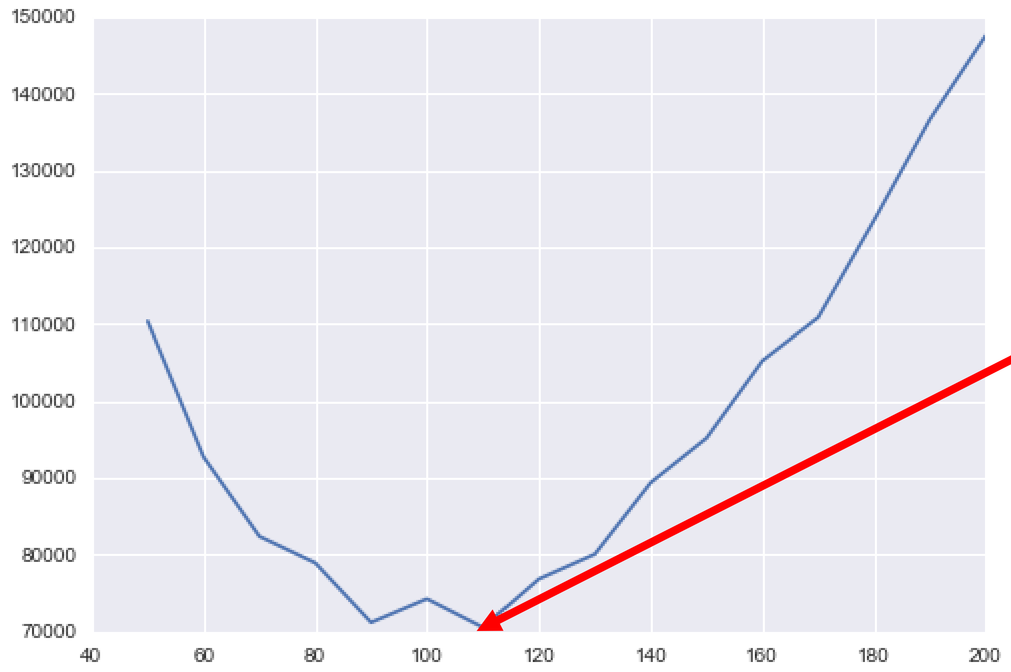
```
from sklearn.decomposition import PCA
pca = PCA(0.99, whiten=True)
data = pca.fit_transform(digits.data)
data.shape
(1797, 41)
```

The result is 41 dimensions, a reduction of nearly 1/3 with almost no information loss. Given this projected data.

GMM for Generating New Data

- **Example: GMM for Generating New Data**
- let's use the AIC to get a gauge for the number of GMM components we should use:

```
n_components = np.arange(50, 210, 10)
models = [GMM(n, covariance_type='full', random_state=0)
          for n in n_components]
aics = [model.fit(data).aic(data) for model in models]
plt.plot(n_components, aics);
```



It appears that around 110 components minimizes the AIC; we will use this model.

GMM for Generating New Data

- **Example: GMM for Generating New Data**
- Let's quickly fit this to the data and confirm that it has converged:

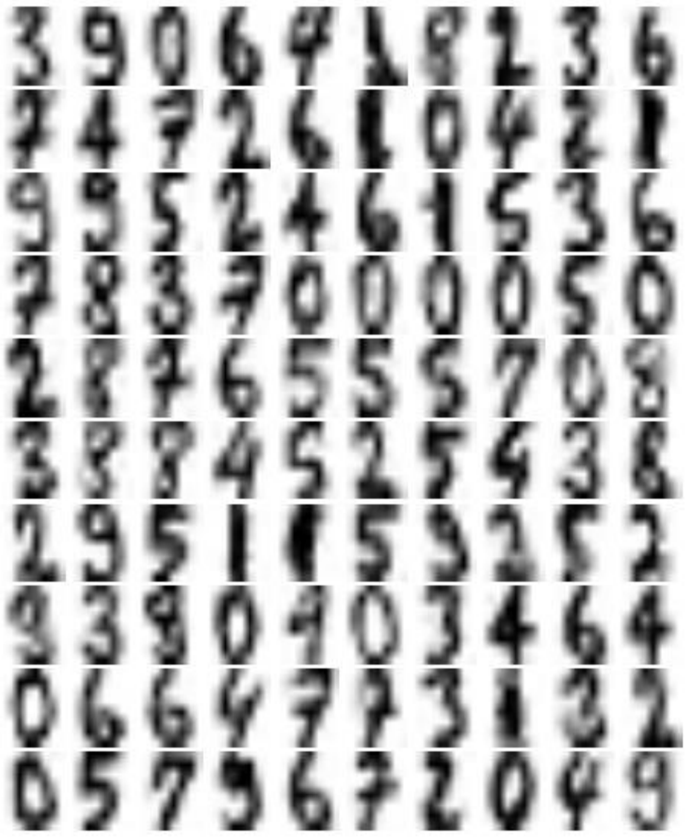
```
gmm = GMM(110, covariance_type='full', random_state=0)
gmm.fit(data)
print(gmm.converged_)
True
```

Now we can draw samples of 100 new points within this 41-dimensional projected space, using the GMM as a generative model:

```
data_new = gmm.sample(100, random_state=0)
data_new.shape
(100, 41)
```

Finally, we can use the inverse transform of the PCA object to construct the new digits:

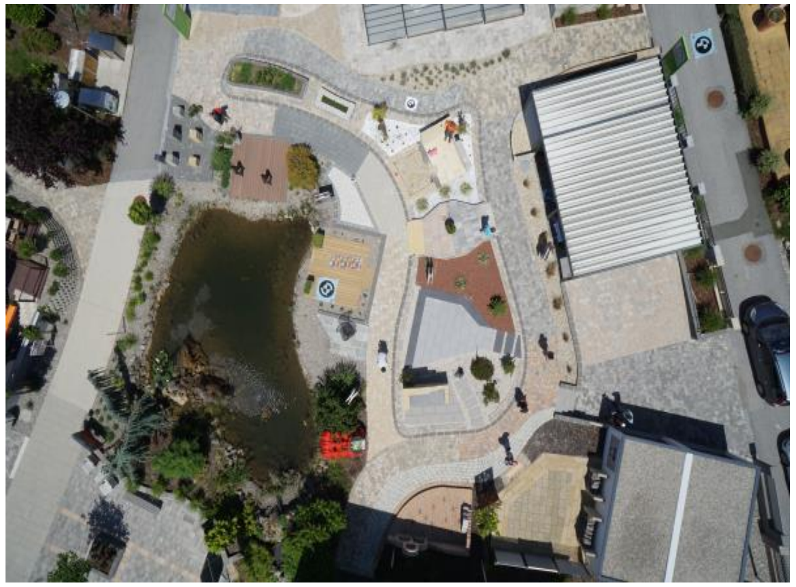
```
digits_new = pca.inverse_transform(data_new)
plot_digits(digits_new)
```

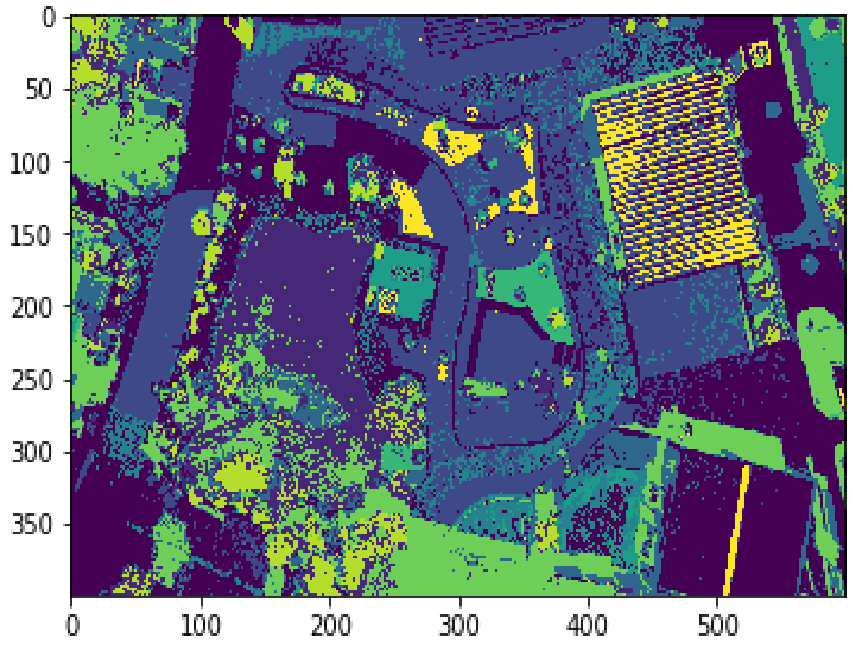


GMM for Segmentation

- **Example: GMM for Segmentation**
- GMM model could be used as a segmentation

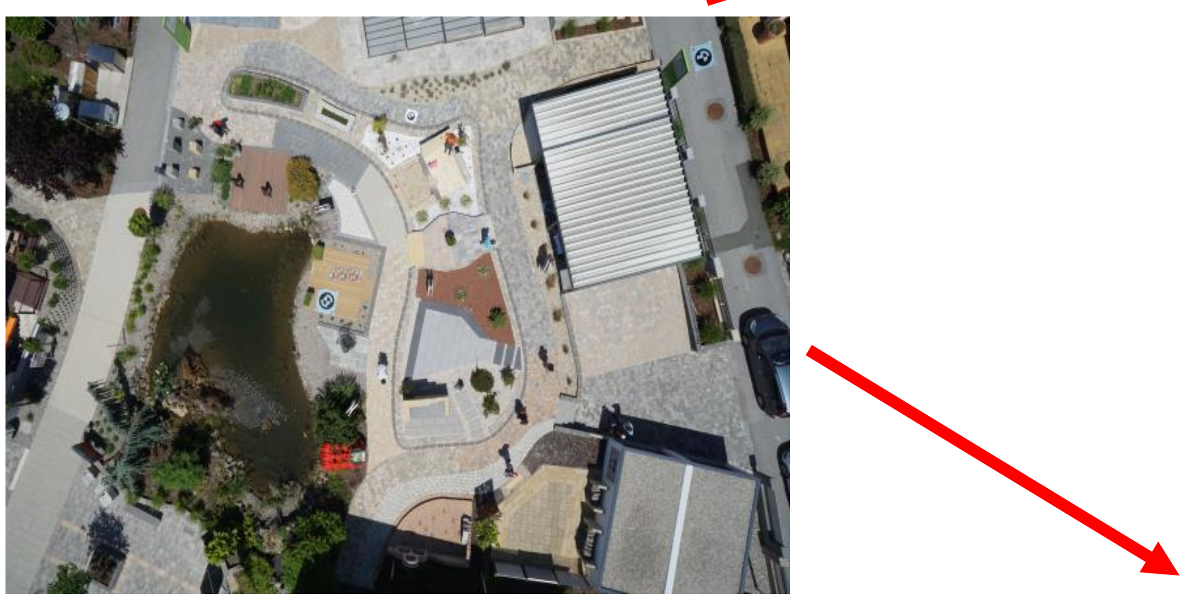
```
import numpy as np
import cv2
path='D:\\LAB4Segmentation\\LAB6\\12.png'
img = cv2.imread(path)
# Convert MxNx3 image into Kx3 where K=MxN
img2 = img.reshape((-1,3))  #-1 reshape means, in this case MxN

from sklearn.mixture import GaussianMixture as GMM

#covariance choices, full, tied, diag, spherical
gmm_model = GMM(n_components=10, covariance_type='tied').fit(img2)  #tied works
better than full
gmm_labels = gmm_model.predict(img2)

#Put numbers back to original shape so we can reconstruct segmented image
original_shape = img.shape
segmented = gmm_labels.reshape(original_shape[0], original_shape[1])
cv2.imwrite("segmented1.jpg", segmented*(255/9))
```

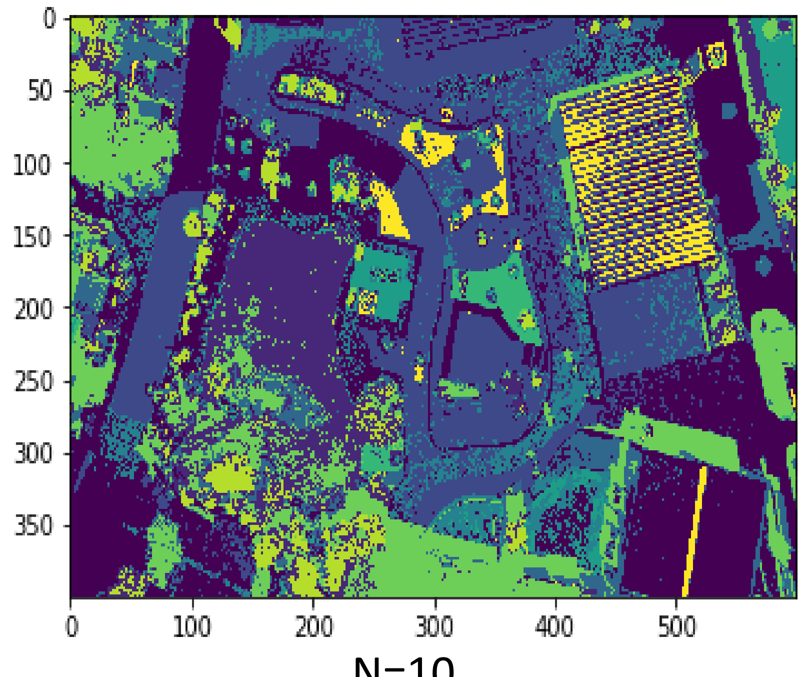




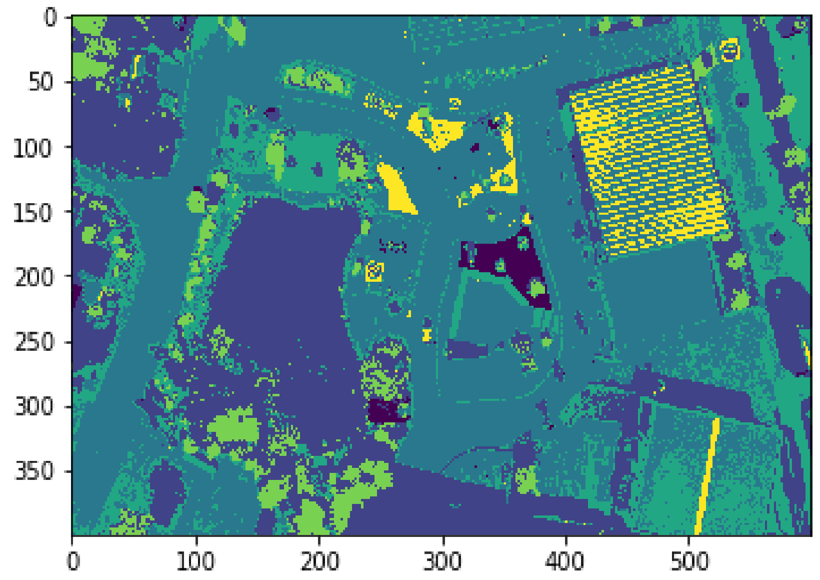
GMM for Segmentation

- **Example: GMM for Segmentation**
- GMM model could be used as a segmentation





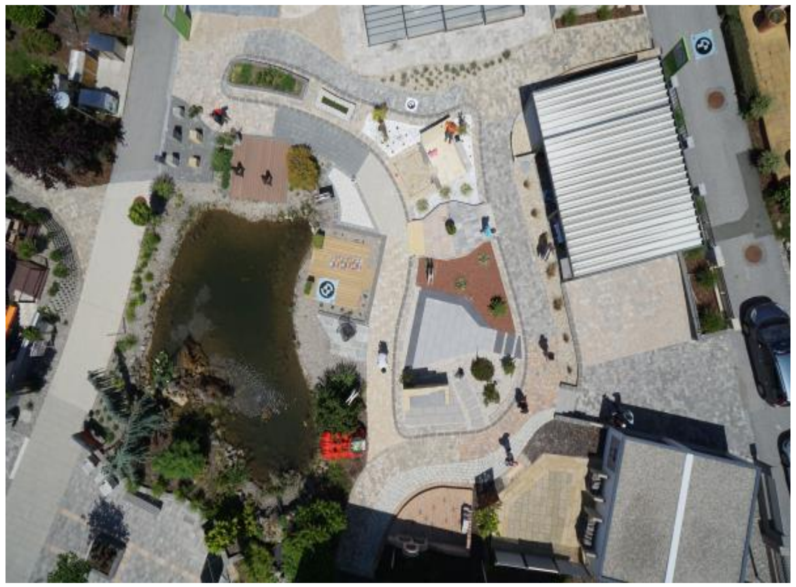
N=10

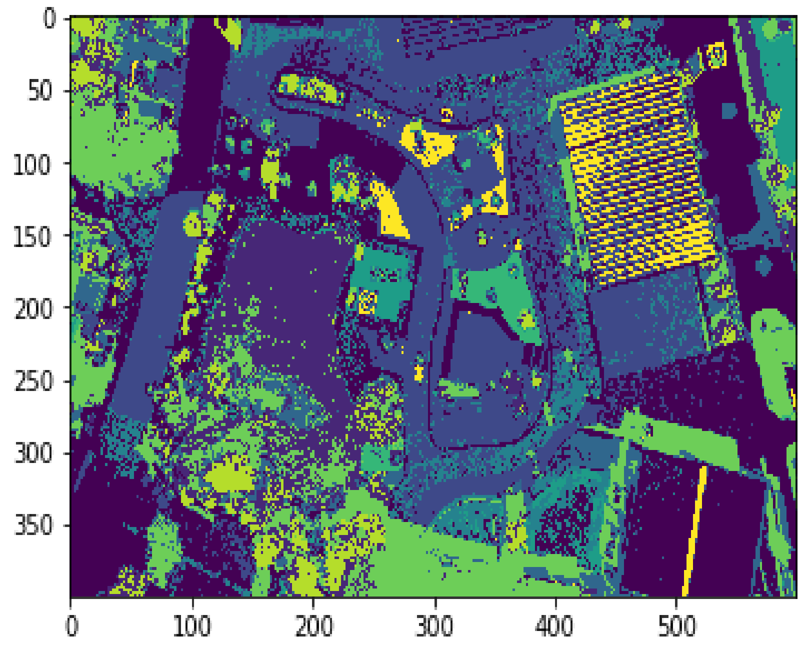


N=5

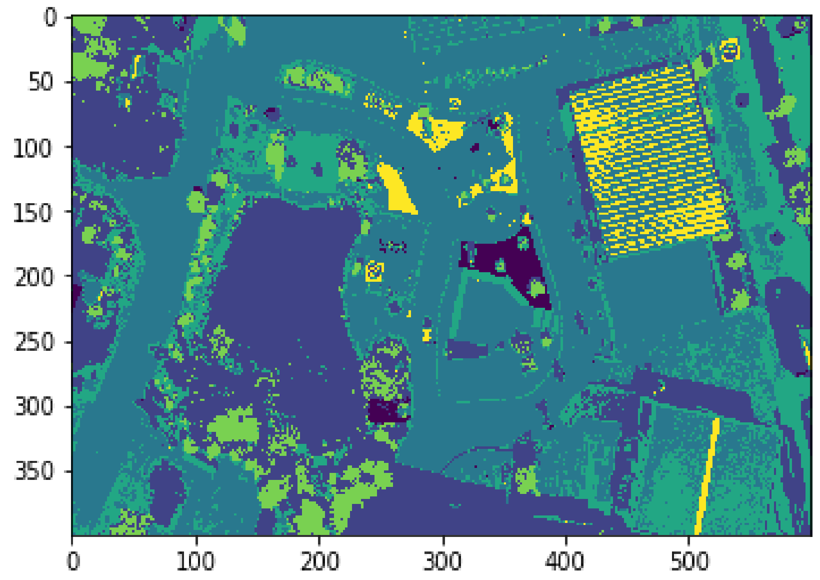
What is GMM?

- **Example: GMM for Segmentation**
- GMM model could be used as a segmentation





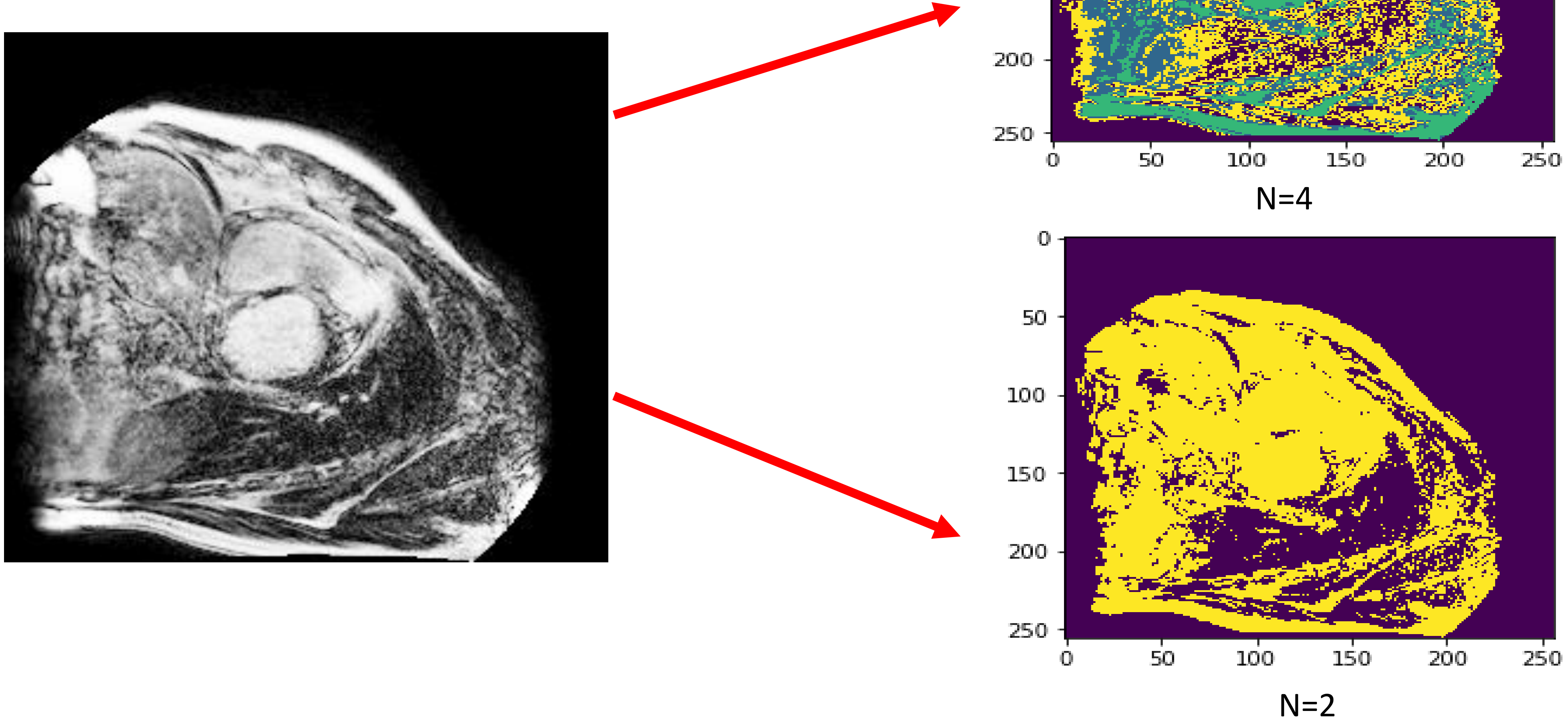
N=10



N=5

GMM for Segmentation

- **Example: GMM for Segmentation**
- GMM model could be used as a segmentation



Q&A