Lecture 6
Clustering

Wonjun Lee

Contents

- Finding centers of similarity using the **prototype-based clustering**
  - ✓K-means Clustering
  - ✓K-means++ Clustering
  - ✓Soft (fuzzy) Clustering
- Taking a bottom-up approach to building **hierarchical** clustering trees
- Identifying arbitrary shapes of objects using a **density-based clustering** approach
- Gaussian Mixture Model

K-means Clustering

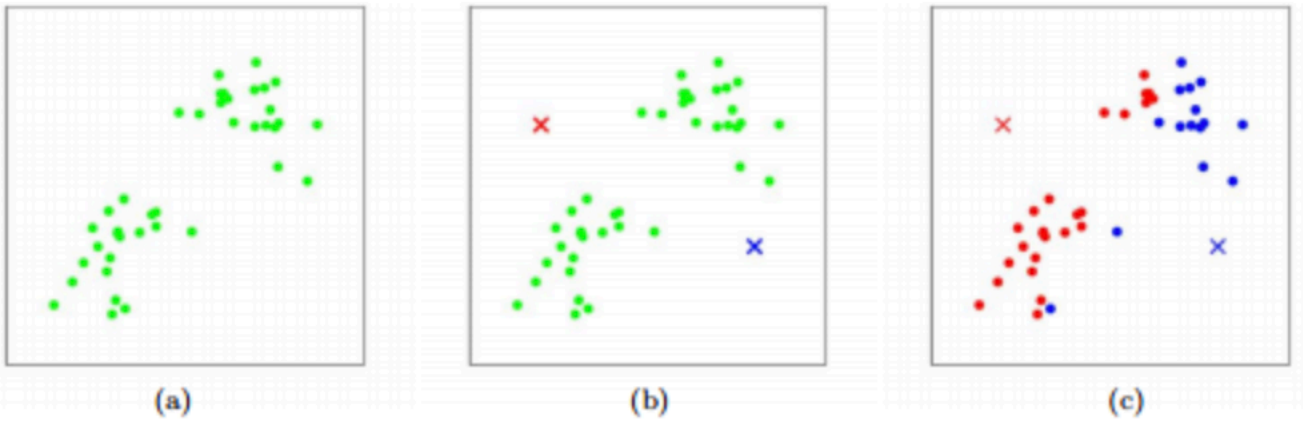
Unsupervised Learning

- Unsupervised Learning – Discover hidden structures in data where we do not know the right answer upfront
- Clustering – 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
  - ✓Grouping of documents, music, and movies by topics
  - ✓Finding customers that share similar interests based on common purchase behaviors

K-means Clustering

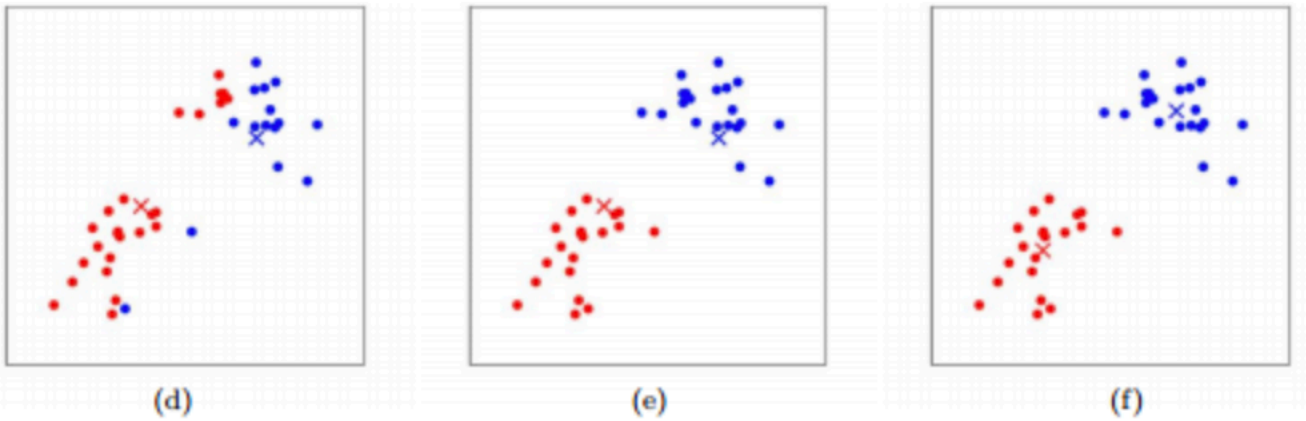
- Prototype-based clustering – each cluster is represented by a prototype
  - ✓Centroid – average of similar points with continuous features
  - ✓Medoid – representative
- (+) Good at identifying clusters with a spherical shape
- (–) Have to specify number of clusters, k, a priori
  - ✓Inappropriate choice for k can result in poor clustering performance
  - ✓Elbow, silhouette plots to find optimal number of clusters, k

K-means Clustering



- Training examples are shown as dots, and cluster centroids are shown as crosses
  - ✓(a) Original dataset
  - ✓(b) Random initial cluster centroids
  - ✓(c-f) Illustration of running two iterations of k-means

K-means Clustering



- In each iteration, we assign each training example to the closest cluster centroid (shown by "painting" the training examples the same color as the cluster centroid to which is assigned)
- Then we move each cluster centroid to the mean of the points assigned to it

K-means Clustering Using scikit-learn

- How do we measure similarity between objects?
- Squared Euclidean Distance between x and y, in m-dim space:

$$d(\boldsymbol{x}, \boldsymbol{y})^2 = \sum_{j=1}^{m} (x_j - y_j)^2 = \|\boldsymbol{x} - \boldsymbol{y}\|_2^2$$

- Sum of Squared Errors (SSE) or Cluster Inertia:

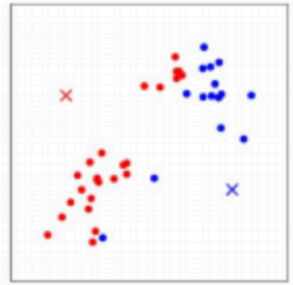
$$SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)} \left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)}\right\|_2^2$$

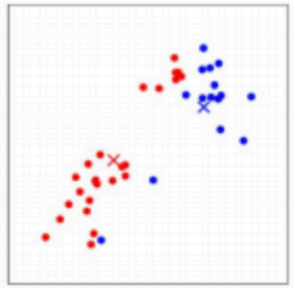
✓ $\mu^{(j)}$= representative point (centroid) for cluster j

✓ $w^{(i,j)} = 1$ if example $x^{(i)}$ is in cluster j, or 0 otherwise

K-means Clustering Algorithm

1. Randomly pick *k* centroids from the examples as initial cluster centers
2. Assign each example to the nearest centroid, $\mu^{(j)}$, $j \in \{1, \dots, k\}$
   - ✓For every i, set $c_j^{(i)} = \arg\min_j ||x^{(i)} - \mu^{(j)}||^2$
3. Move the centroids to the center of the examples that were assigned to it
   - ✓Find the $\mu^{(j)}$ that satisfies SSE minimum $SSE = \sum_{i=1}^{n}\sum_{j=1}^{k} w^{(i,j)} \|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)}\|_2^2$
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





K-means Clustering Using scikit-learn

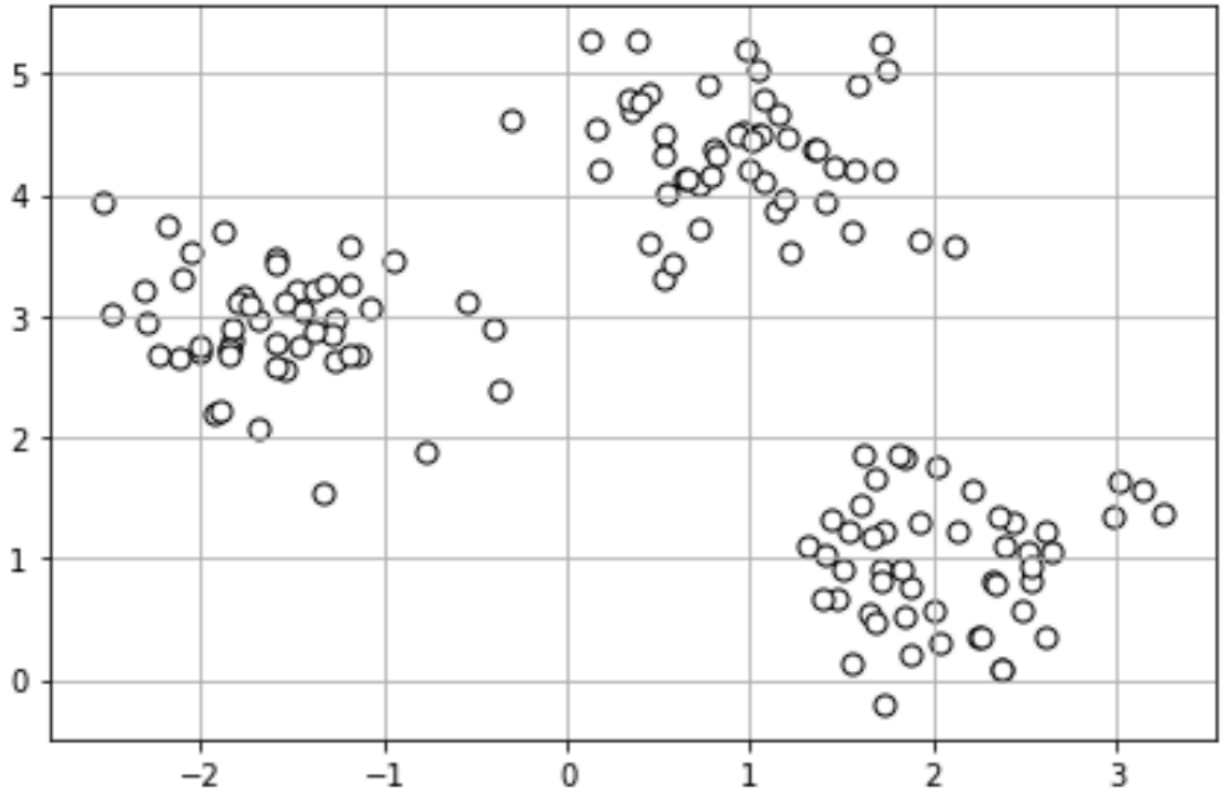
- Ex) Two-dimensional dataset

```
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.savefig('images/11_01.png', dpi=300)
plt.show()
```

K-means Clustering Using scikit-learn

- 150 randomly generated points – roughly grouped into three regions with higher density in 2-d



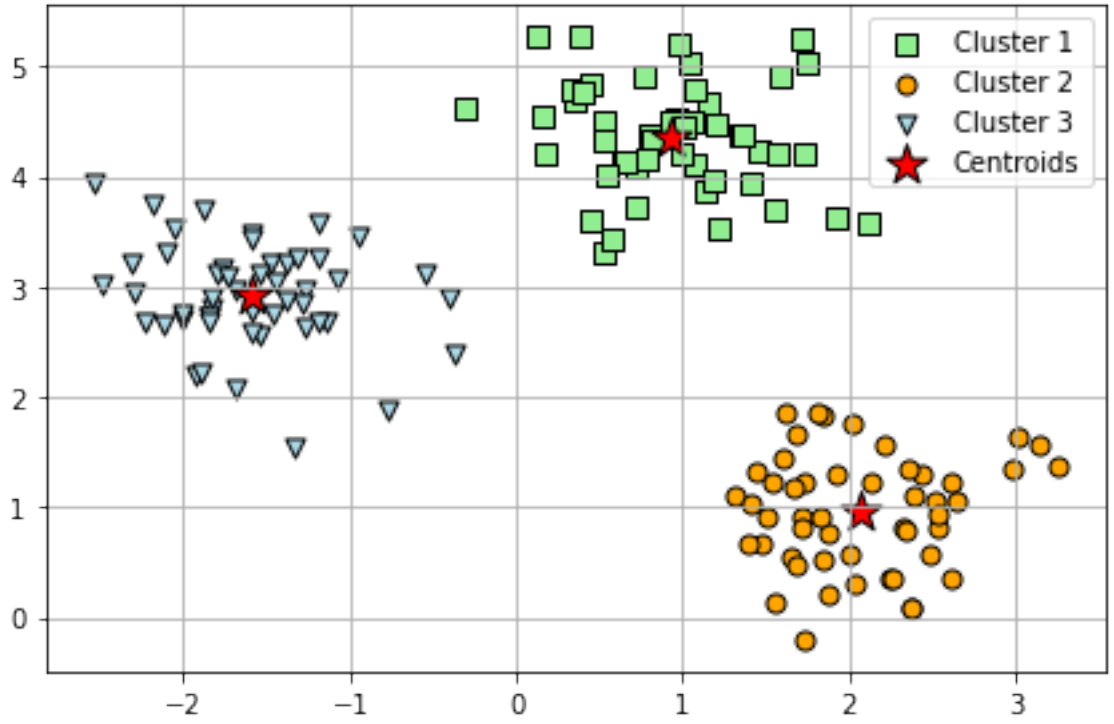
K-means Clustering Using scikit-learn

```
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)
```

- *n_clusters* = # of desired clusters
- *n_init* – Run the k-means clustering algorithm 10 times with different random centroids to choose the final model as the one with the lowest SSE
- *max_iter* = stops early if it converges before the maximum number of iterations is reached
- *tol* = when it does not reach convergence for a particular run (due to computational limit) if choose larger values for *max_iter* → declare convergence with large number of *tol*

K-means Clustering Using scikit-learn

- k-means placed the three centroids at the center of each sphere, which looks like a reasonable grouping given this dataset



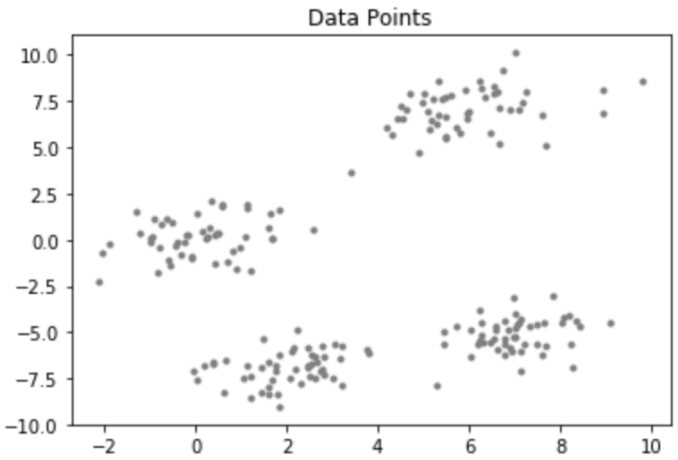
A Smarter Way of Placing the Initial Cluster Centroids Using k-means++

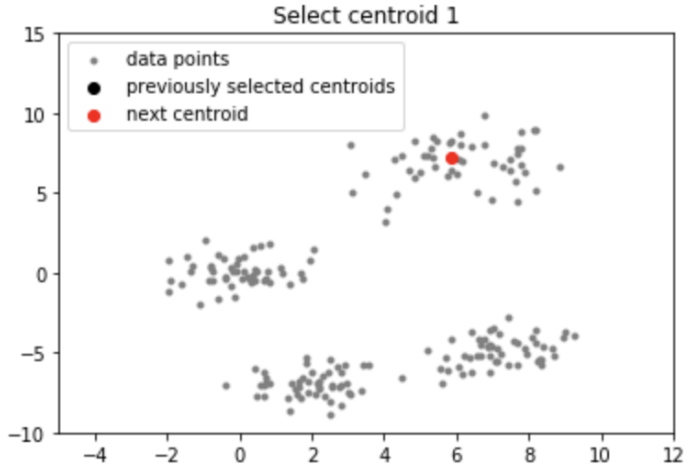
- Random seed to place the initial centroids → sometimes results in bad clustering or slow convergence

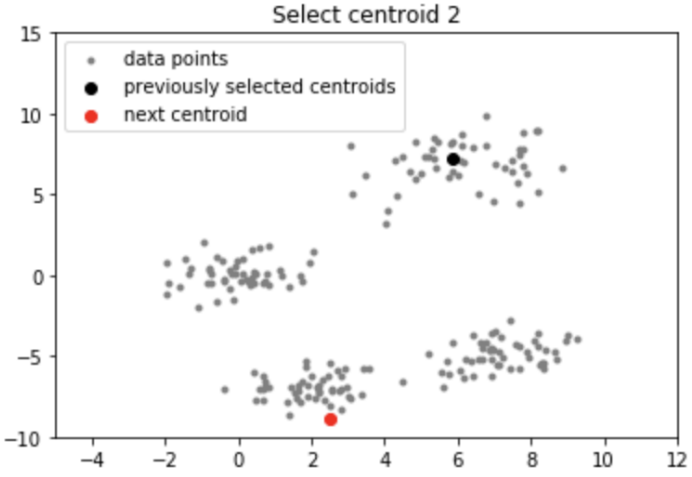
A Smarter Way of Placing the Initial Cluster Centroids Using k-means++

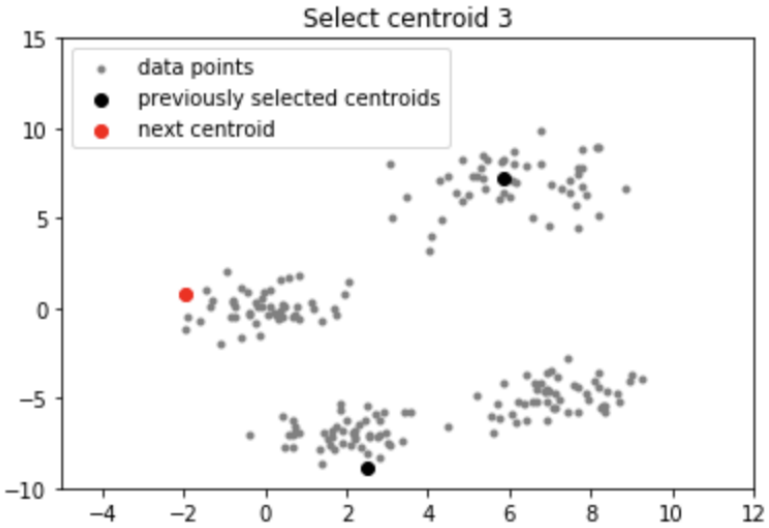
- Place the initial centroids far away from each other
  1. Randomly select the first centroid from the data points
  2. For each data point compute its distance from the nearest, previously chosen centroid
  3. Select the point having maximum distance from the nearest centroid as the next centroid
  4. Repeat steps 2 and 3 until k centroids have been sampled

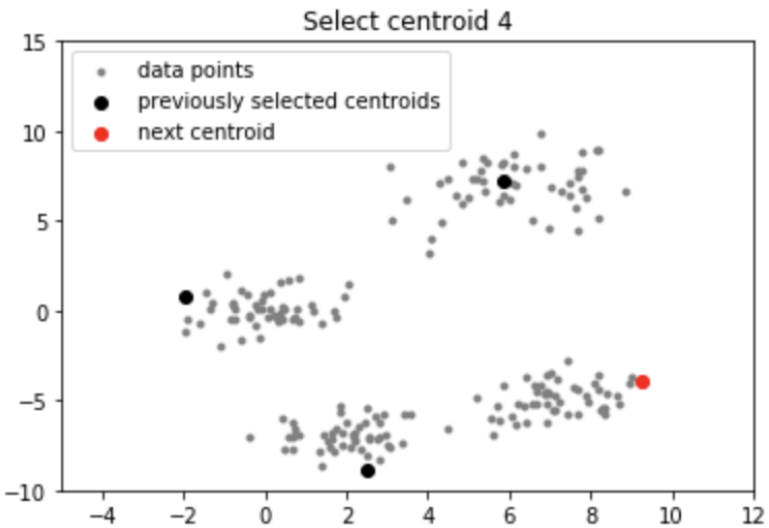
A Smarter Way of Placing the Initial Cluster Centroids Using k-means++











k-means++ in scikit-learn

- Use *init='k-means++'*

Hard vs. Soft Clustering

- Hard Clustering – a dataset is assigned to exactly one cluster
  - ✓Ex) K-means and k-means++
- Soft (fuzzy) Clustering – an example to one or more clusters
  - ✓Ex) fuzzy C-means (FCM)
- In K-means $\begin{bmatrix} x \in \mu^{(1)} & \rightarrow & w^{(i,j)} = 0 \\ x \in \mu^{(2)} & \rightarrow & w^{(i,j)} = 1 \\ x \in \mu^{(3)} & \rightarrow & w^{(i,j)} = 0 \end{bmatrix}$

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

FCM Algorithm

1. Specify the number of *k* centroids and randomly assign the cluster memberships for each point
2. Compute the cluster centroids, $\boldsymbol{\mu}^{(j)}$, $j \in \{1, \ldots, k\}$
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

FCM Algorithm with Example

1. Specify the number of *k* centroids and randomly assign the cluster memberships for each point

   Ex) X = {(1,2), (2,3), (9,4), (10,1)}, k = 2, m = 2

x_1	1	2
x_2	2	3
x_3	9	4
x_4	10	1

	w(1,1)	w(1,2)
x_1	.4	.6
x_2	.88	.12
x_3	.41	.59
x_4	.27	.73

FCM Algorithm

2. Compute the cluster centroids, $\boldsymbol{\mu}^{(j)}$, $j \in \{1, \ldots, k\}$ – calculate as the mean of all examples weighted by the degree to which each example belongs to that cluster

$$\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^{n} w^{(i,j)^m} \boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} w^{(i,j)^m}}$$

x_1	1	2
x_2	2	3
x_3	9	4
x_4	10	1

	$w^{(1,1)}$	$w^{(1,2)}$
x_1	.4	.6
x_2	.88	.12
x_3	.41	.59
x_4	.27	.73

$\sum_{i=1}^{n} w^{(i,j)^m}$

$\sum_{i=1}^{n} w^{(i,1)^2} = 0.4^2 + 0.88^2 + 0.41^2 + 0.27^2 = 1.18$

$\sum_{i=1}^{n} w^{(i,2)^2} = 0.6^2 + 0.12^2 + 0.59^2 + 0.73^2 = 1.2$

$\boldsymbol{\mu}^{(1)} = [\boldsymbol{\mu}^{(1,1)}, \boldsymbol{\mu}^{(1,2)}]$

$\boldsymbol{\mu}^{(1,1)} = \frac{0.4^2 \times 1 + 0.88^2 \times 2 + 0.41^2 \times 9 + 0.27^2 \times 10}{1.18} = 3.38,$

$\boldsymbol{\mu}^{(1,2)} = \frac{0.4^2 \times 2 + 0.88^2 \times 3 + 0.41^2 \times 4 + 0.27^2 \times 1}{1.18} = 2.88$

$\boldsymbol{\mu}^{(1)} = [3.38, 2.88]$. Likewise, $\boldsymbol{\mu}^{(2)} = [7.02, 2.14]$

FCM Algorithm

3. Update the cluster memberships for each point

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

x_1	1	2
x_2	2	3
x_3	9	4
x_4	10	1

$\boldsymbol{\mu}^{(1)}$	3.38	2.88
$\boldsymbol{\mu}^{(2)}$	7.02	2.14

	Euclidean Distance to $\boldsymbol{\mu}^{(1)}$	Euclidean Distance to $\boldsymbol{\mu}^{(2)}$
x_1	2.54	6.03
x_2	1.38	5.10
x_3	5.73	2.71
x_4	6.88	3.19

$$w^{(1,1)} = \frac{\frac{1}{2.54}}{\frac{1}{2.54}+\frac{1}{6.03}} = 0.7,\ w^{(1,2)} = \frac{\frac{1}{6.03}}{\frac{1}{2.54}+\frac{1}{6.03}} = 0.3$$

$w^{(2,1)} = 0.79,\ w^{(2,2)} = 0.21$

$w^{(3,1)} = 0.32,\ w^{(3,2)} = 0.68$

$w^{(4,1)} = 0.32,\ w^{(4,2)} = 0.68$

	$w^{(1,1)}$	$w^{(1,2)}$
x_1	.7	.3
x_2	.79	.21
x_3	.32	.68
x_4	.32	.68

	$w^{(1,1)}$	$w^{(1,2)}$	
x_1	.88	.12	→ Cluster 1
x_2	.94	.06	→ Cluster 1
x_3	.17	.83	→ Cluster 2
x_4	.18	.82	→ Cluster 2

Using the Elbow Method to Find the Optimal Number of Clusters

- Within Cluster **Sum of Squared Error** (**SSE**) in k-means (**Cluster Inertia**):

$$SSE = \sum_{i=1}^{n}\sum_{j=1}^{k} w^{(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

- SSE in FCM:

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

→ Use *inertia_* attribute after fitting a *Kmeans* model

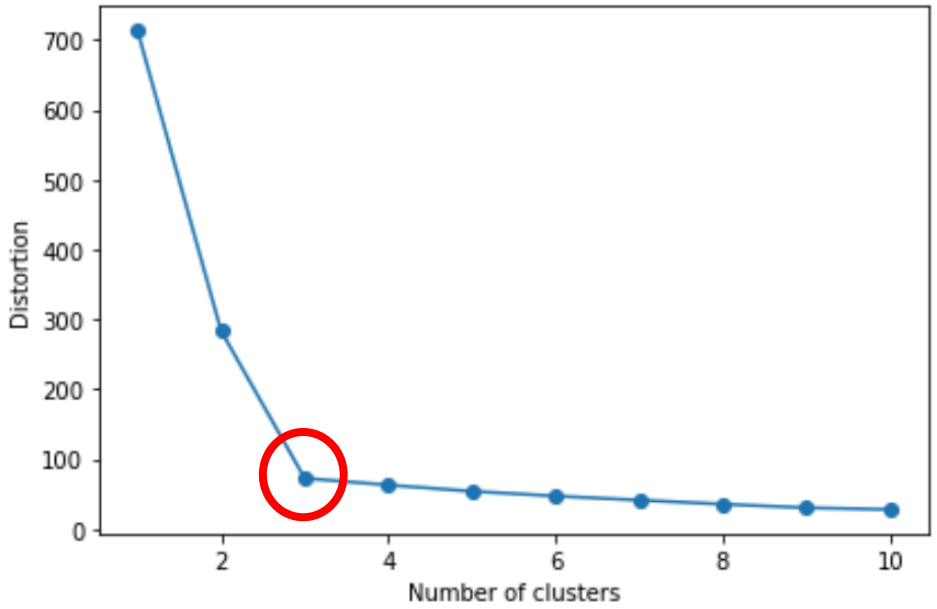
```
print('Distortion: %.2f' % km.inertia_)
```

```
Distortion: 72.48
```

Using the Elbow Method to Find the Optimal Number of Clusters

- Elbow Method – Estimate the optimal number of clusters, k, for a given task
  - ✓If k increases, the distortion will decrease
  - ✓Idea 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.savefig('images/11_03.png', dpi=300)
plt.show()
```



Quantifying the Quality of Clustering Via Silhouette Plots

- Silhouette Analysis – graphical tool to plot a measure of how tightly grouped the examples in the clusters are

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 → how similar
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 → how dissimilar
3. Calculate the silhouette, $s^{(i)}$, as the difference between cluster cohesion and separation divided by the greater of the two, as shown here:

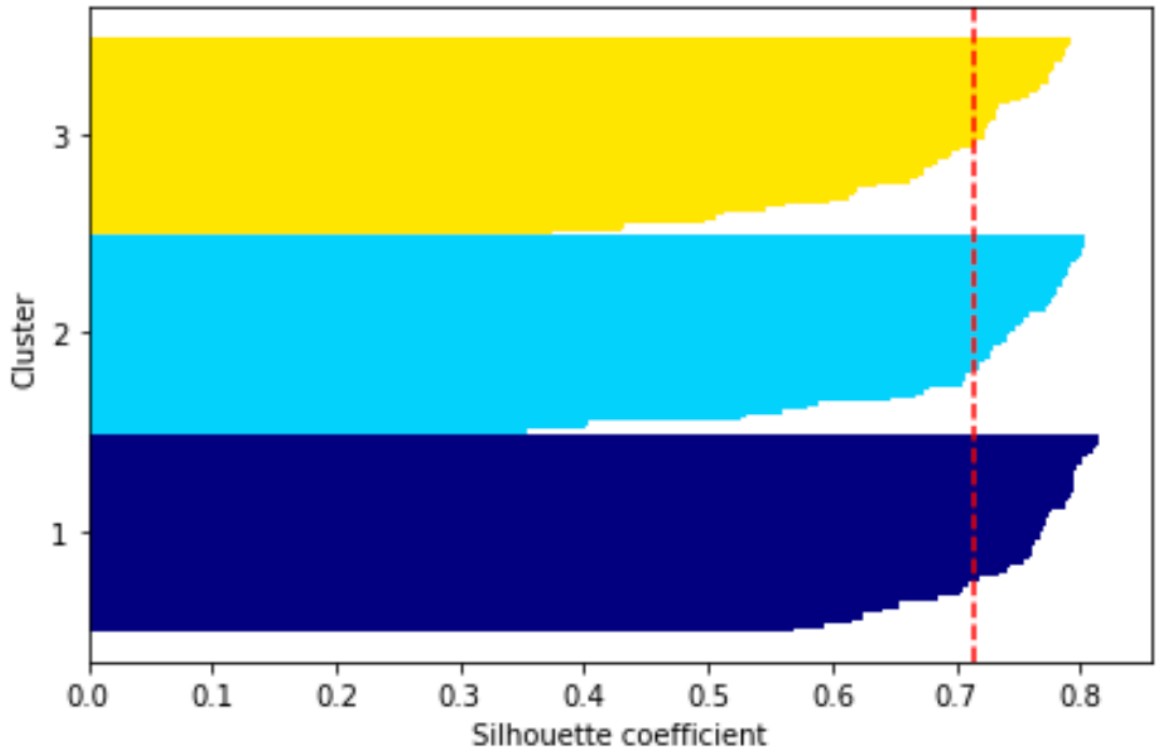
$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{max\{b^{(i)}, a^{(i)}\}}$$

Quantifying the Quality of Clustering Via Silhouette Plots

- [-1, 1] : coefficient range
- 0 : cluster separation = cohesion
- 1 : ideal silhouette if $b^{(i)} >> a^{(i)}$
- *silhouette_samples* from scikit-learn *metric* module
- *silhouette_scores* function – calculates average silhouette coefficient (= *numpy.mean(silhouette_ samples(...))*)

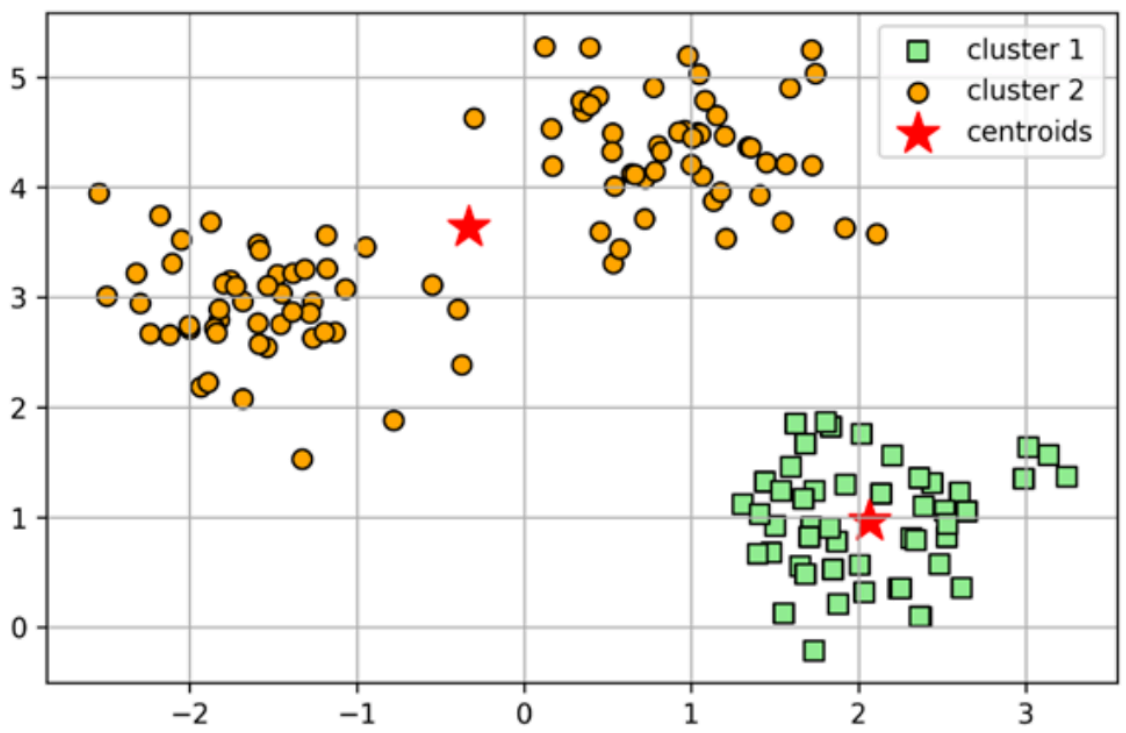
Quantifying the Quality of Clustering Via Silhouette Plots

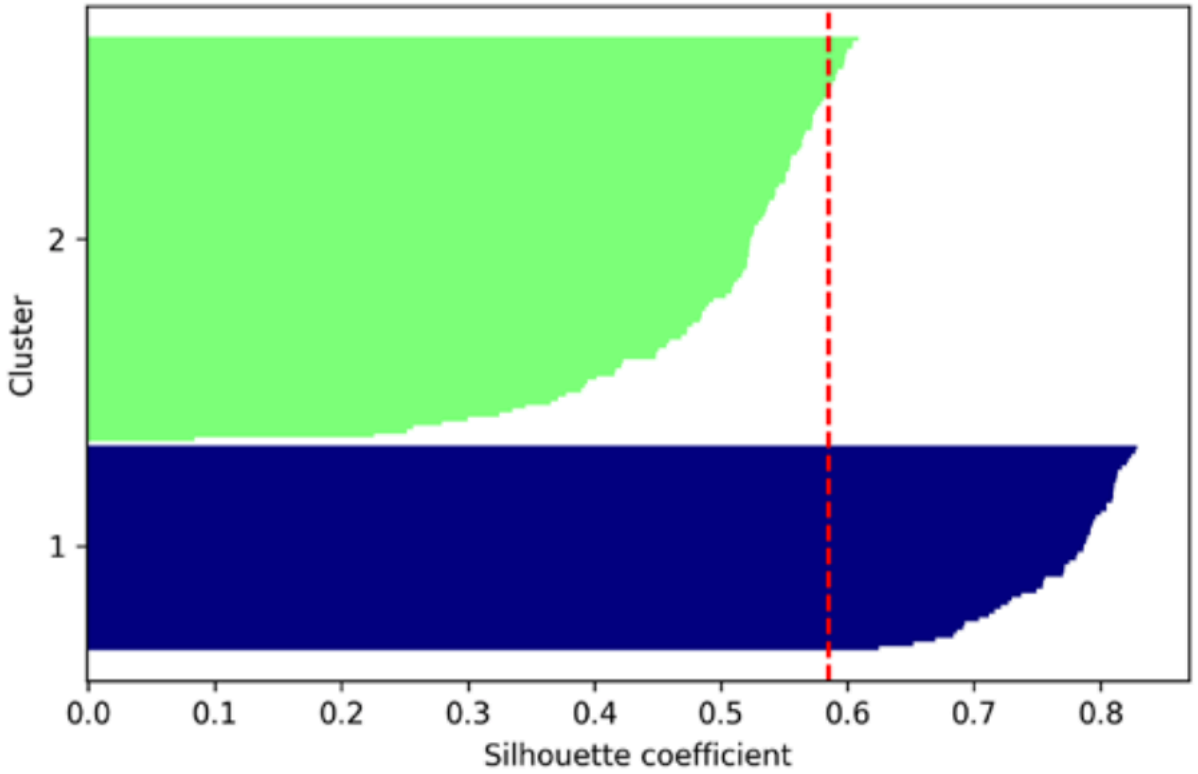
```
from sklearn.metrics import silhouette_samples

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)

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)
```



- The silhouette coefficients are not even close to **0** → an indicator of a *good* clustering
- To summarize the goodness of the clustering → added the average silhouette coefficient (dotted line)

Quantifying the Quality of Clustering Via Silhouette Plots





- The silhouettes have visibly different lengths and widths → evidence of a relatively *bad* or at least *suboptimal* clustering

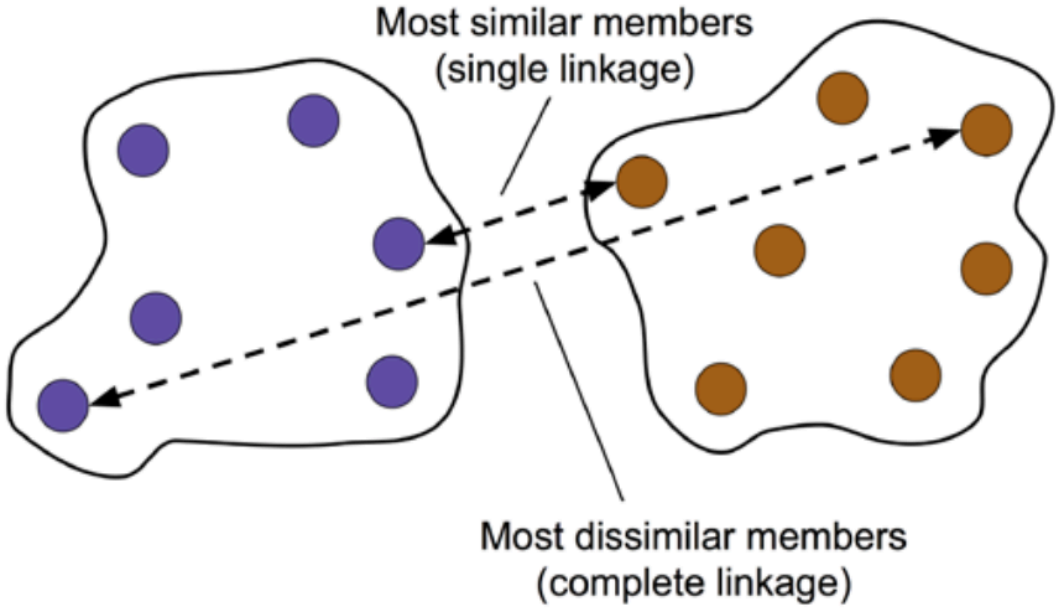
Hierarchical Clustering

Organizing Clusters as a Hierarchical Tree

- Advantage
  - ✓Allows us to plot dendrograms (visualizations of a binary hierarchical clustering)
  - ✓Does not need to specify number of clusters
- Two main approaches
  - ✓**Agglomerative** hierarchical clustering – start with each example as an individual cluster and merge the closest pairs of clusters until only one cluster remains
  - ✓**Divisive** hierarchical clustering – start with one cluster that encompasses the complete dataset, and iteratively split the cluster into smaller clusters until each cluster only contains one example

Grouping Clusters in Bottom-up Fashion

- Agglomerative hierarchical clustering
  - ✓**Single linkage** – Compute the distances between the most similar members for each pair of clusters and merge the two clusters for which the distance between the most similar members is the smallest
  - ✓**Complete linkage** – Compare the most dissimilar members to perform the merge



Grouping Clusters in Bottom-up Fashion

1. Compute the distance matrix of all examples
2. Represent each data point as a singleton cluster
3. Merge the two closest clusters based on the distance between the most dissimilar (distant) members
4. Update the similarity matrix
5. Repeat steps 2-4 until one single cluster remains

Grouping Clusters in Bottom-up Fashion

```
import pandas as pd
import numpy as np

np.random.seed(123)

variables = ['X', 'Y', 'Z']
labels = ['ID_0', 'ID_1', 'ID_2', 'ID_3', 'ID_4']

X = np.random.random_sample([5, 3])*10
df = pd.DataFrame(X, columns=variables, index=labels)
df
```

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

Grouping Clusters in Bottom-up Fashion

- Performing hierarchical clustering on a distance matrix

```
from scipy.spatial.distance import pdist, squareform

row_dist = pd.DataFrame(squareform(pdist(df, metric='euclidean')),
                        columns=labels,
                        index=labels)
row_dist
```

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

Grouping Clusters in Bottom-up Fashion

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

	ID_1	ID_2	ID_3	ID_0_4
ID_1	0			
ID_2	4.347073	0		
ID_3	5.104311	7.244262	0	
ID_0_4	**6.698233**	**8.316594**	**5.899885**	0

- The first cluster is ID_0 and ID_4
- Distance between other vectors and ID_0_4:
- ID_1 to ID_0_4 = max(ID_1→ID_0, ID_1→ID_4) = max(4.973534, 6.698233) = **6.698233**
- ID_2 to ID_0_4 = max(ID_2→ID_0, ID_2→ID_4) = max(5.516653, 8.316594) = **8.316594**
- ID_3 to ID_0_4 = max(ID_3→ID_0, ID_3→ID_4) = max(5.899885, 4.382864) = **5.899885**

Grouping Clusters in Bottom-up Fashion

	ID_1	ID_2	ID_3	ID_0_4
ID_1	0			
ID_2	4.347073	0		
ID_3	5.104311	7.244262	0	
ID_0_4	6.698233	8.316594	5.899885	0

	ID_1_2	ID_3	ID_0_4
ID_1_2	0		
ID_3	**7.244262**	0	
ID_0_4	**8.316594**	5.899885	0

Grouping Clusters in Bottom-up Fashion

	ID_1_2	ID_3	ID_0_4
ID_1_2	0		
ID_3	7.244262	0	
ID_0_4	8.316594	5.899885	0

	ID_1_2	ID_3_0_4
ID_1_2	0	
ID_3_0_4	**8.316594**	0

Grouping Clusters in Bottom-up Fashion

	ID_1_2	ID_3_0_4
ID_1_2	0	
ID_3_0_4	8.316594	0

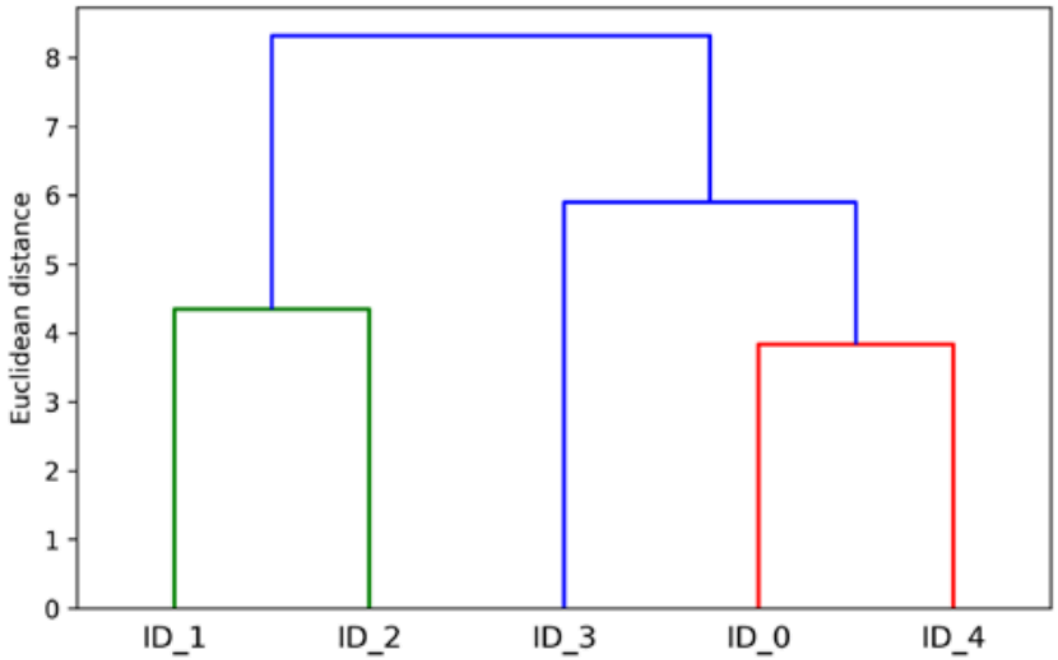
Grouping Clusters in Bottom-up Fashion

```
row_clusters = linkage(df.values, method='complete', metric='euclidean')
pd.DataFrame(row_clusters,
             columns=['row label 1', 'row label 2',
                      'distance', 'no. of items in clust.'],
             index=['cluster %d' % (i + 1)
                    for i in range(row_clusters.shape[0])])
```

	row label 1	row label 2	distance	no. of items in clust.
cluster 1	0.0	4.0	3.835396	2.0
cluster 2	1.0	2.0	4.347073	2.0
cluster 3	3.0	5.0	5.899885	3.0
cluster 4	6.0	7.0	8.316594	5.0

Grouping Clusters in Bottom-up Fashion

```
from scipy.cluster.hierarchy import dendrogram

# make dendrogram black (part 1/2)
# from scipy.cluster.hierarchy import set_link_color_palette
# set_link_color_palette(['black'])

row_dendr = dendrogram(row_clusters,
                       labels=labels,
                       # make dendrogram black (part 2/2)
                       # color_threshold=np.inf
                       )
plt.tight_layout()
plt.ylabel('Euclidean distance')
#plt.savefig('images/11_11.png', dpi=300,
#            bbox_inches='tight')
plt.show()
```



- Dendrogram summarizes different clusters that were formed during the agglomerative hierarchical clustering

Applying Agglomerative Clustering via scikit-learn

```
from sklearn.cluster import AgglomerativeClustering

ac = AgglomerativeClustering(n_clusters=3,
                             affinity='euclidean',
                             linkage='complete')
labels = ac.fit_predict(X)
print('Cluster labels: %s' % labels)
```

```
Cluster labels: [1 0 0 2 1]
```

```
ac = AgglomerativeClustering(n_clusters=2,
                             affinity='euclidean',
                             linkage='complete')
labels = ac.fit_predict(X)
print('Cluster labels: %s' % labels)
```

```
Cluster labels: [0 1 1 0 0]
```

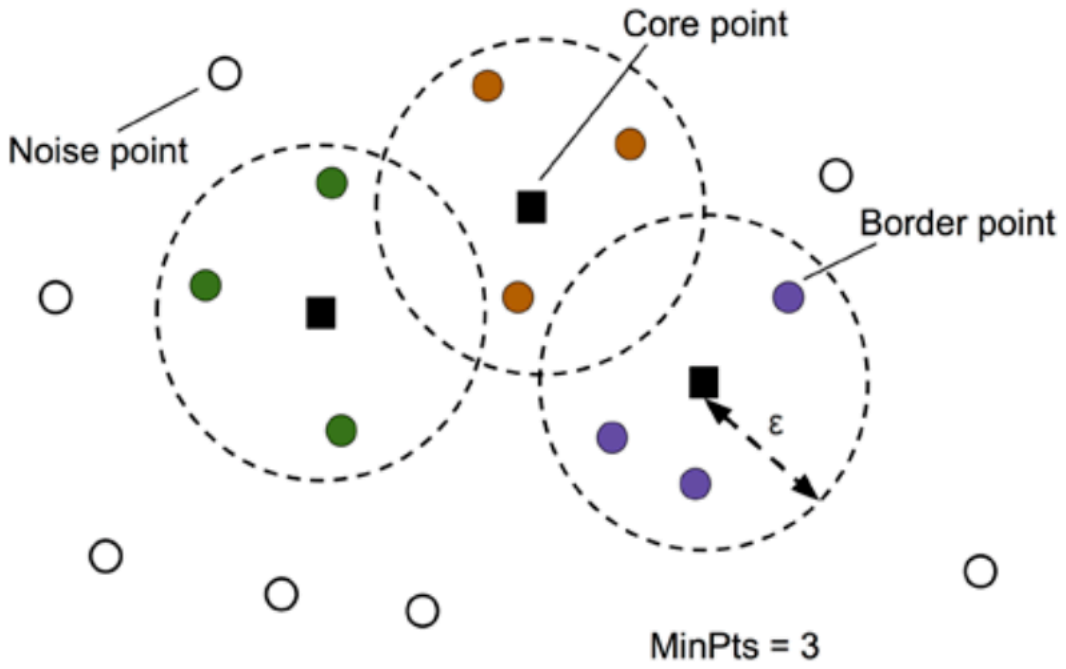
Density-based Clustering

Locating Regions of High Density via DBSCAN

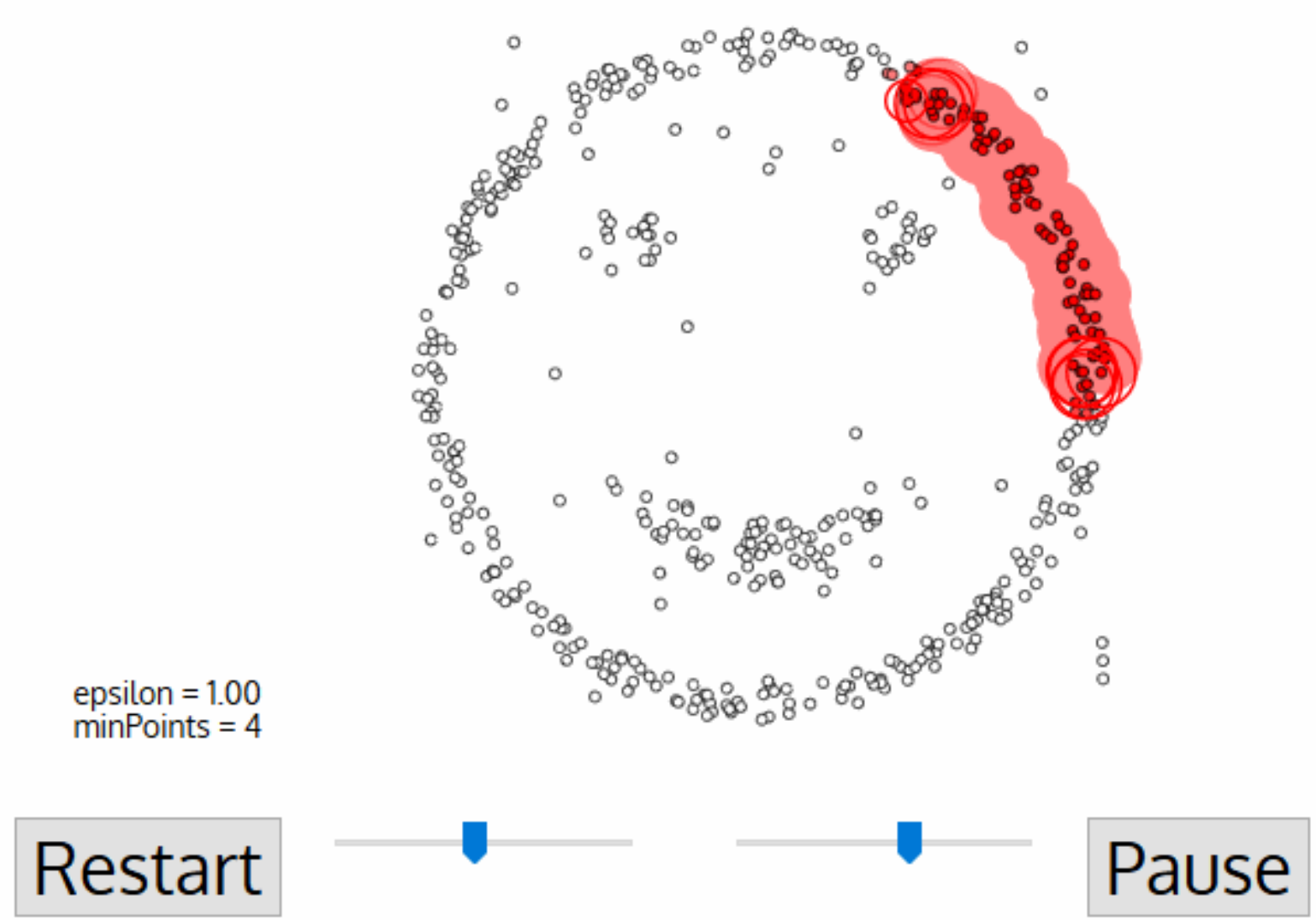
- Density-based spatial clustering of applications with noise (DBSCAN)
  - ✓No assumptions about spherical clusters like k-means
  - ✓No partition of the dataset into hierarchies that require a manual cut-off point
  - ✓Doesn't necessarily assign each point to a cluster but is capable of removing noise points
- Assigns cluster labels based on dense regions of points
  - ✓Density – the number of points within a specified radius, ε
- Assign each example (data point) using the following criteria:
  - ✓A point is considered a **core point** if at least a specified number (MinPts) of neighboring points fall within the specified radius, ε
  - ✓A **border point** is a point that has fewer neighbors than MinPts within ε, but lies within the ε radius of a core point
  - ✓All other points that are neither core nor border points are considered **noise points**

DBSCAN Algorithm

1. Form a separate cluster for each core point or connected group of core points (Core points are connected if they are no farther away than ε)
2. Assign each border point to the cluster of its corresponding core point

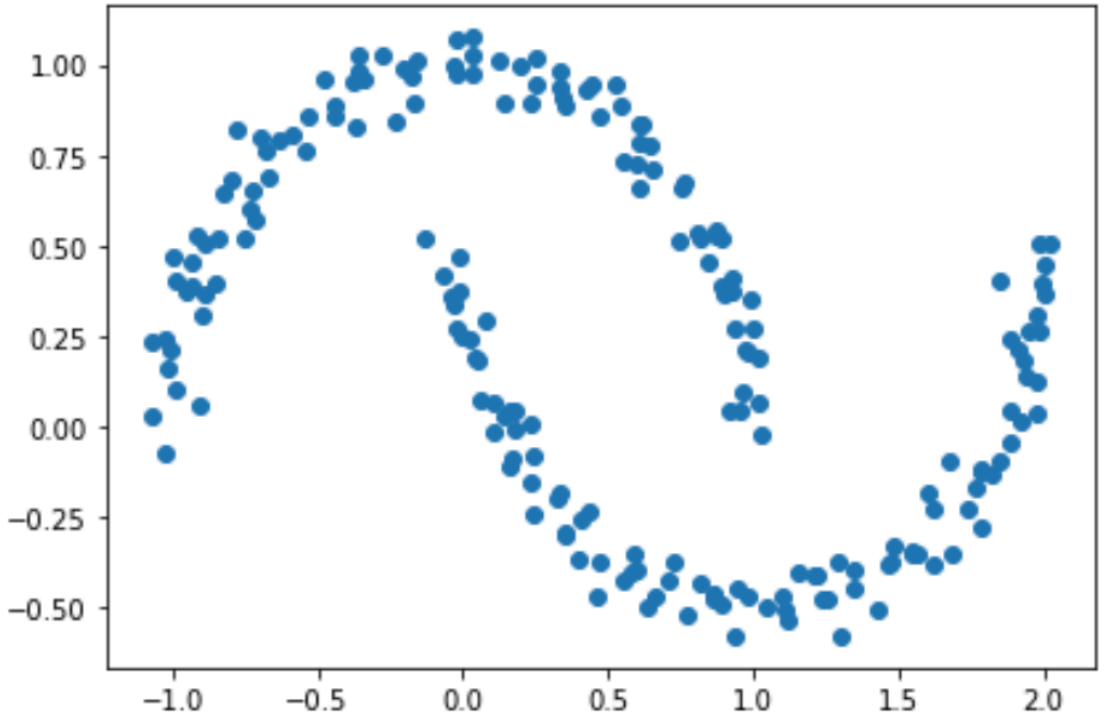


DBSCAN Algorithm



DBSCAN Implementation - A half-moon dataset

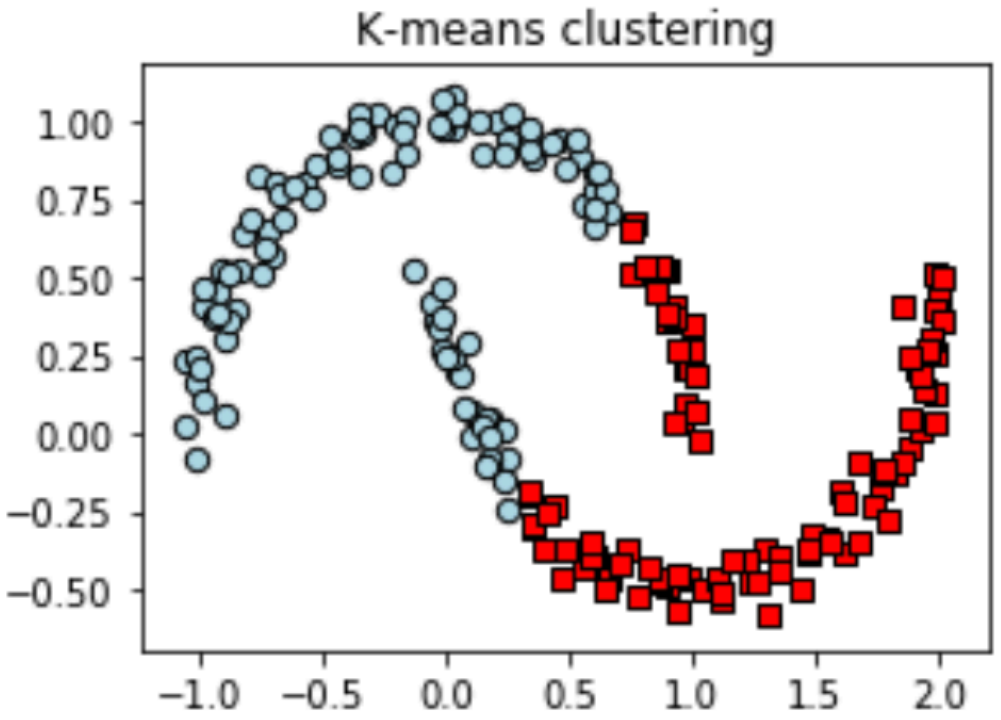
```
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
plt.scatter(X[:, 0], X[:, 1])
plt.tight_layout()
#plt.savefig('images/11_14.png', dpi=300)
plt.show()
```

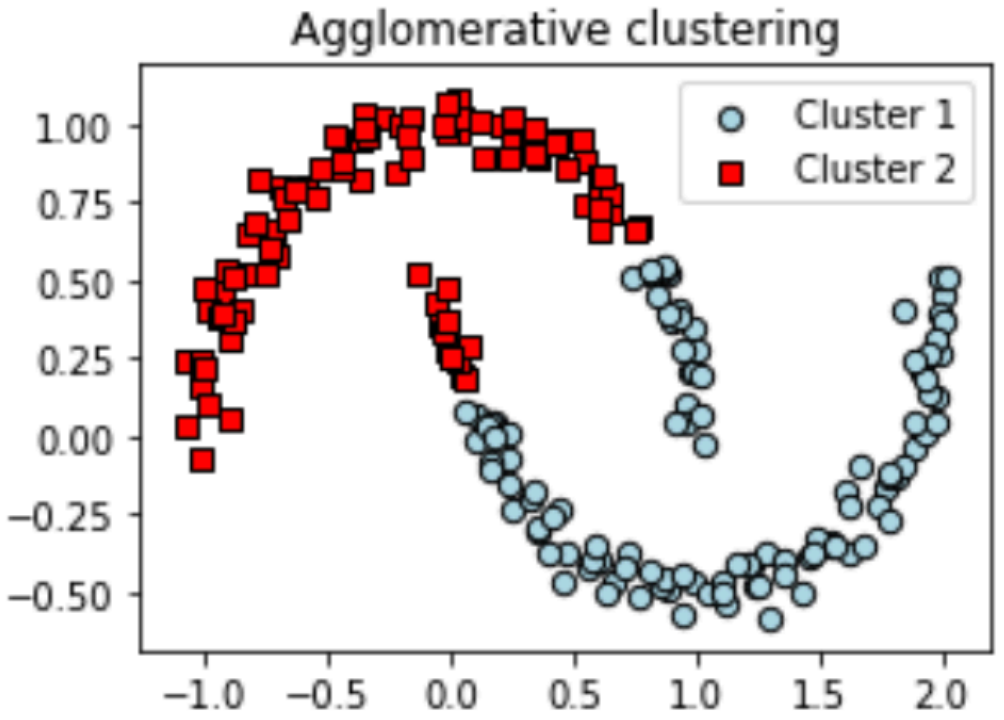


DBSCAN Implementation - A half-moon dataset

```
km = KMeans(n_clusters=2, random_state=0)
y_km = km.fit_predict(X)
```

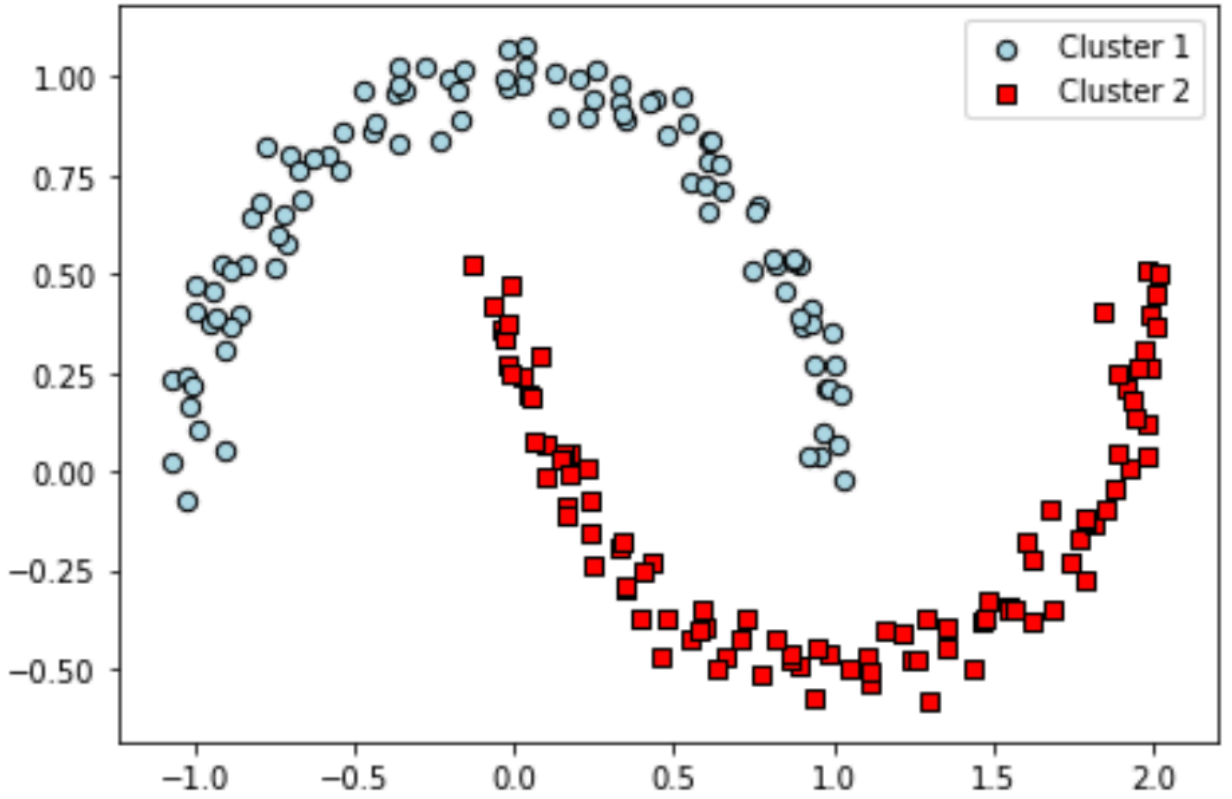
```
ac = AgglomerativeClustering(n_clusters=2,
                             affinity='euclidean',
                             linkage='complete')
y_ac = ac.fit_predict(X)
```





DBSCAN Implementation - A half-moon dataset

```
from sklearn.cluster import DBSCAN

db = DBSCAN(eps=0.2, min_samples=5, metric='euclidean')
y_db = db.fit_predict(X)
```



DBSCAN Implementation - Disadvantages

- Two hyperparameters in DBSCAN (MinPts and ε) need to be optimized to yield good clustering results ← Finding a good combination of MinPts and ε can be problematic if the density differences in the dataset are relatively large
- Not always obvious which clustering algorithm will perform best on a given dataset, especially if the data comes in multiple dimensions that make it hard or impossible to visualize
- Furthermore, it is important to emphasize that a successful clustering does not only depend on the algorithm and its hyperparameters; rather, the choice of an appropriate distance metric and the use of domain knowledge that can help to guide the experimental setup can be even more important

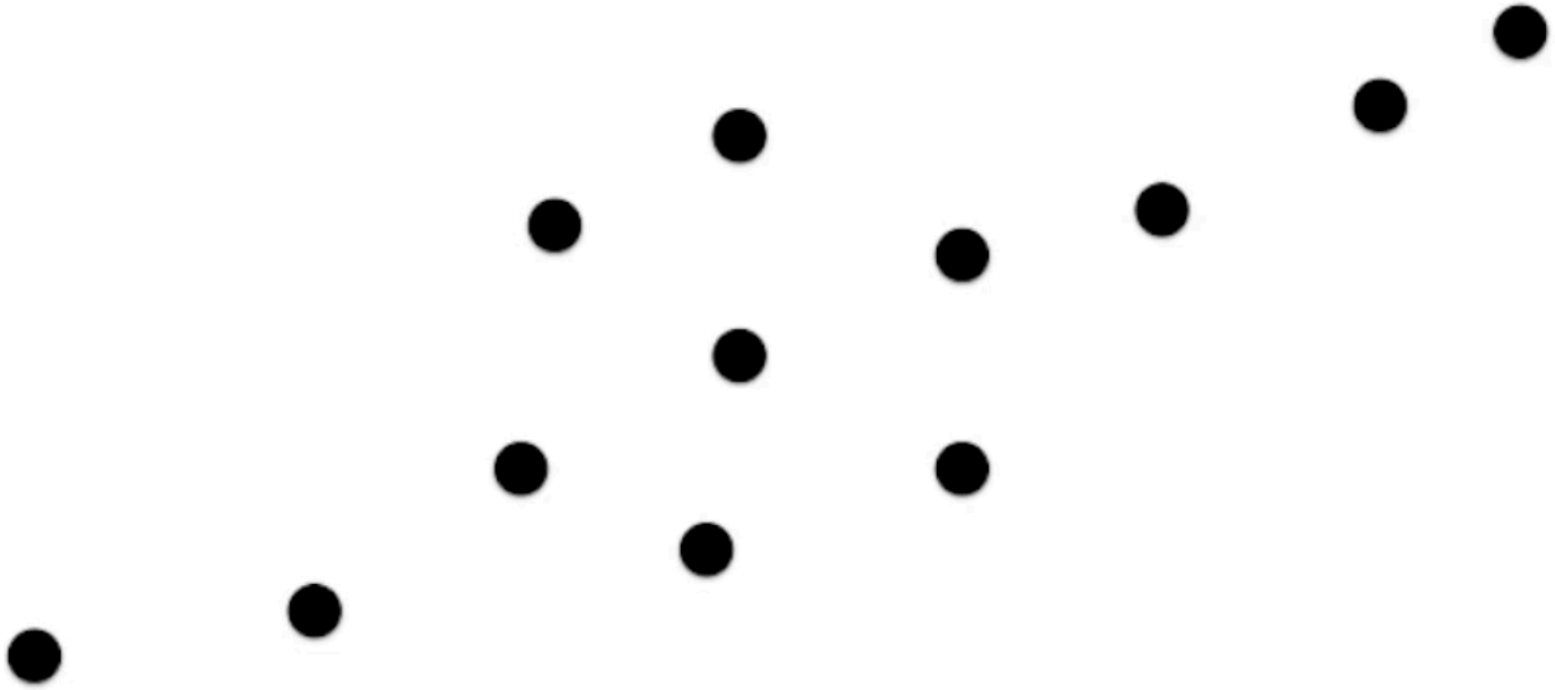
Gaussian Mixture Model

Gaussian Mixtures Model – Algorithm

```
Start with random Gaussians
while (not converge)
    1. Color points according to Gaussians
    2. Recalculate Gaussians according to colors
```

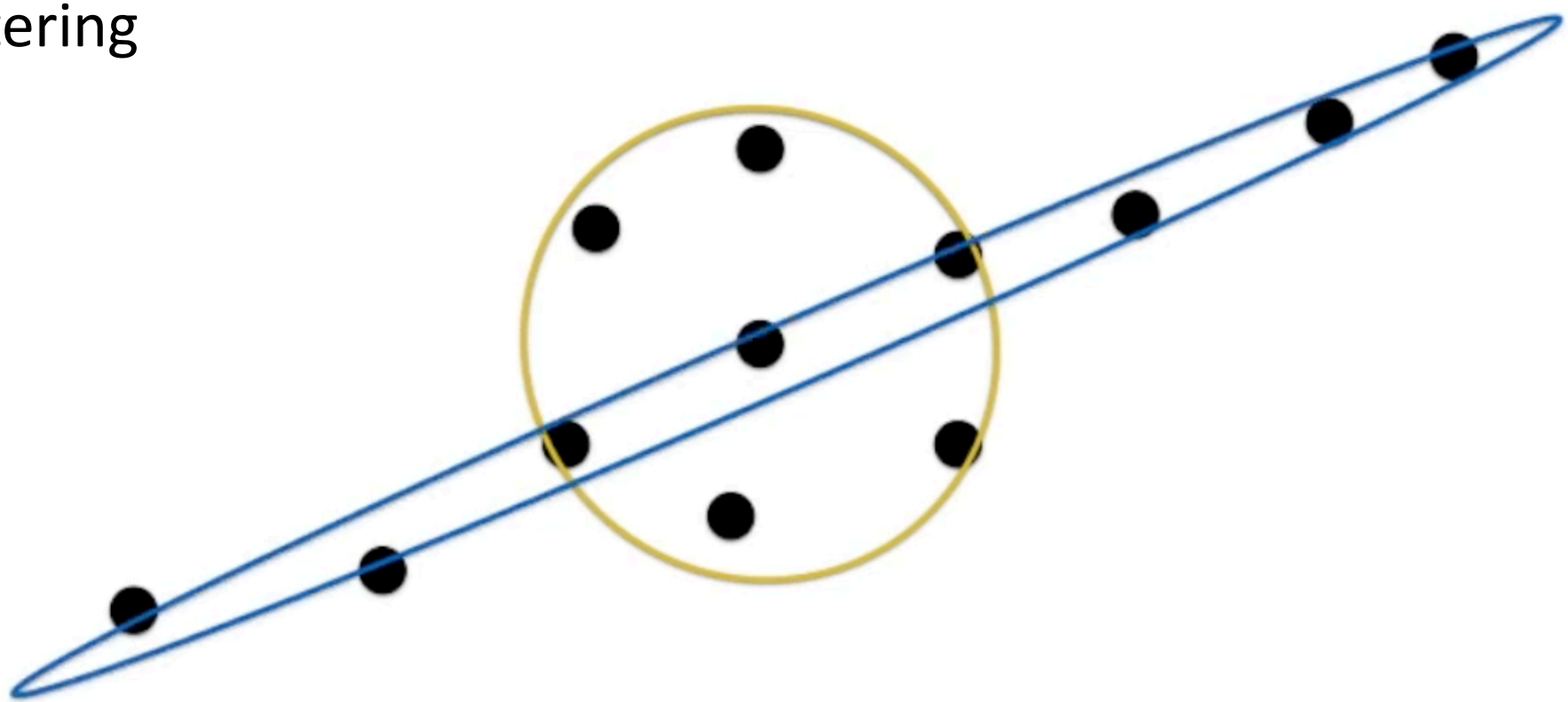
Gaussian Mixtures Model – Algorithm

- Samples



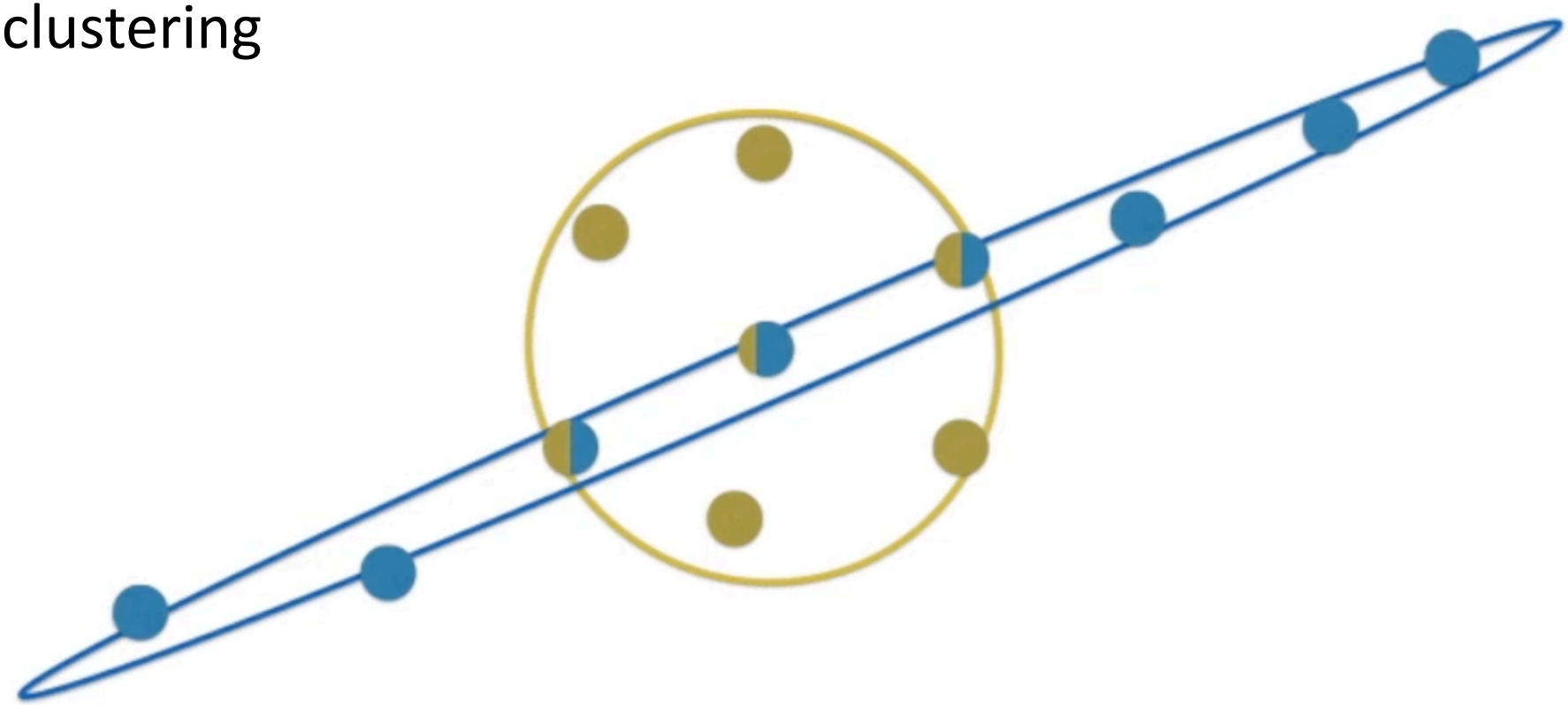
Gaussian Mixtures Model – Algorithm

- Clustering



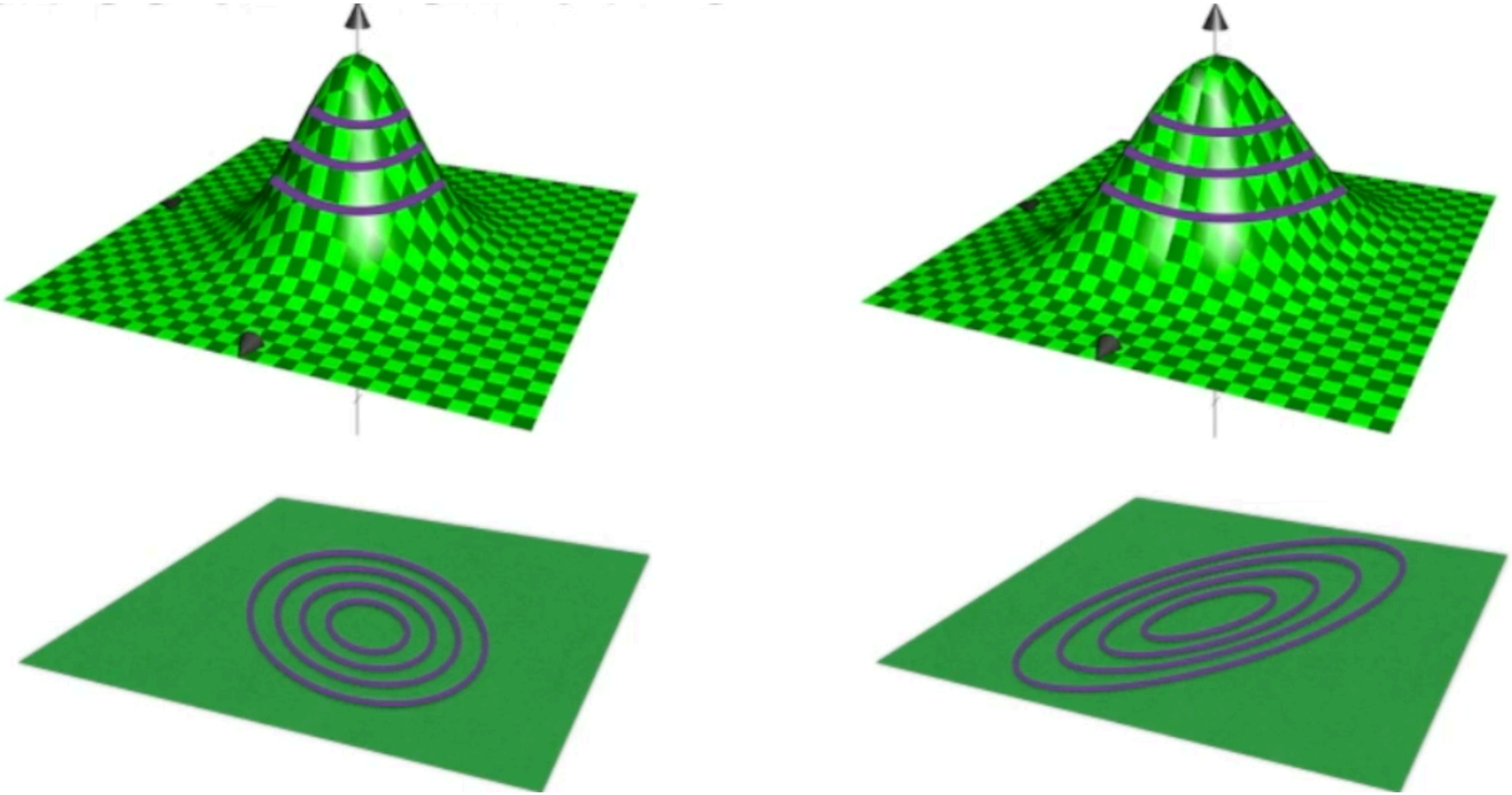
Gaussian Mixtures Model – Algorithm

- Soft clustering



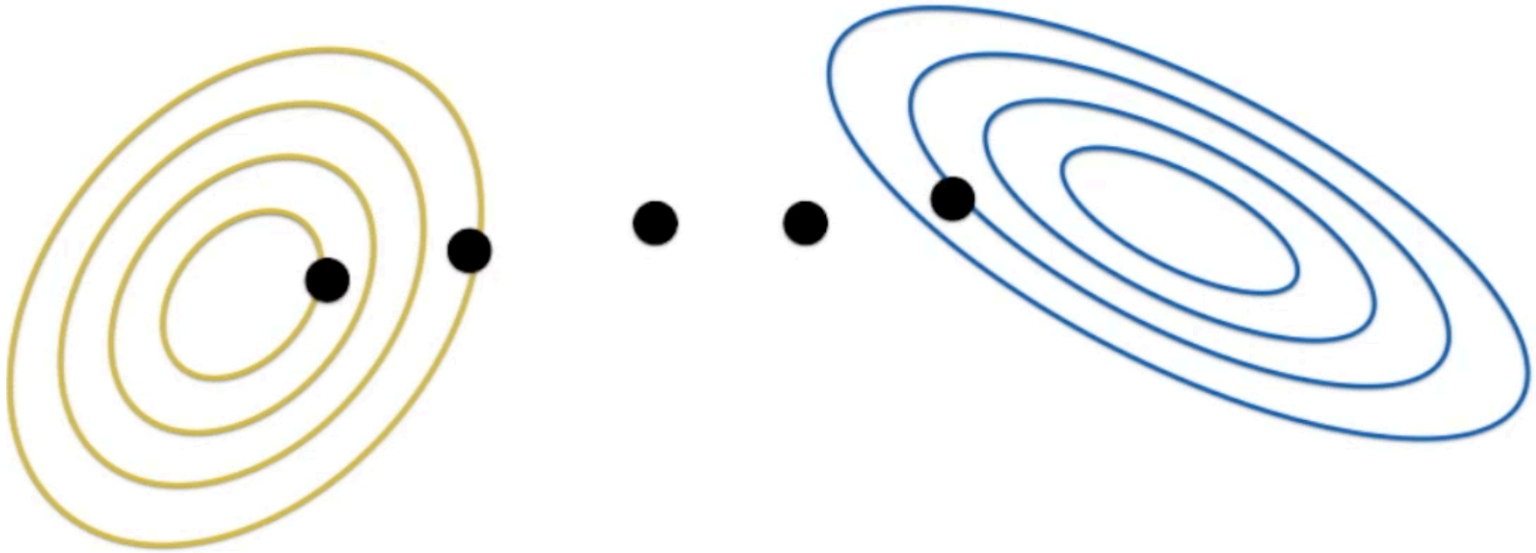
Gaussian Mixtures Model – Algorithm

- Gaussian distribution



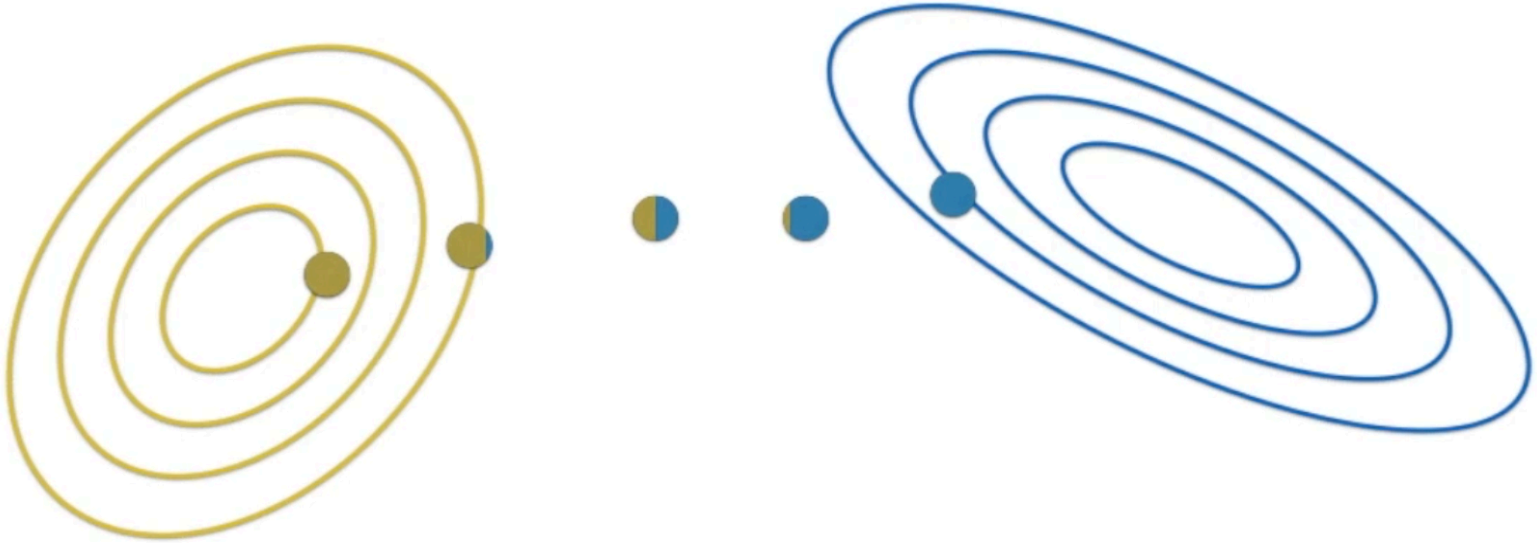
Gaussian Mixtures Model – Algorithm

- Coloring points -



Gaussian Mixtures Model – Algorithm

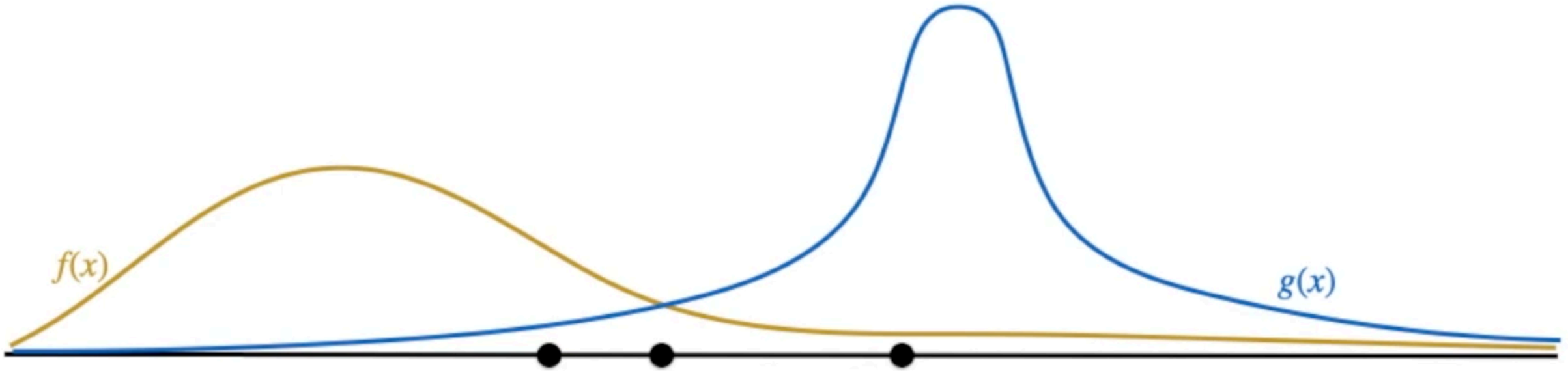
- Coloring points
  - ✓Points close to yellow (blue) mountain have more portion of yellow (blue)
  - ✓Points in between yellow and blue have half yellow and half blue



How exactly well do this?

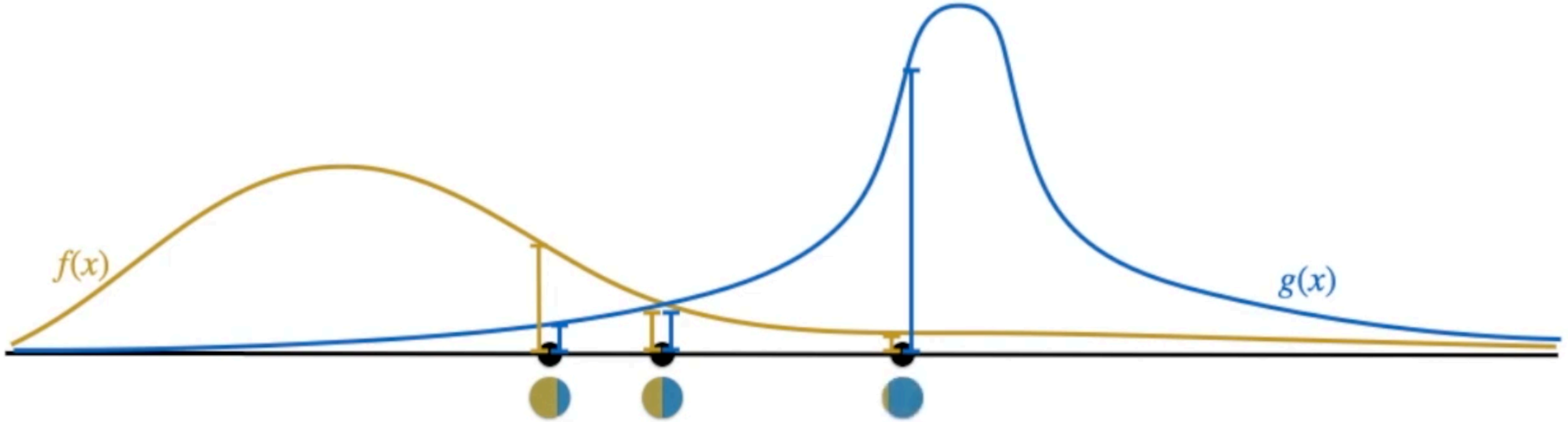
Gaussian Mixtures Model – Algorithm

- Coloring points



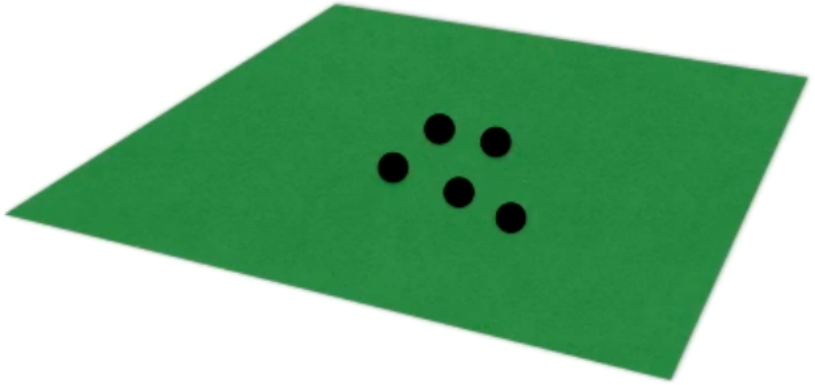
Gaussian Mixtures Model – Algorithm

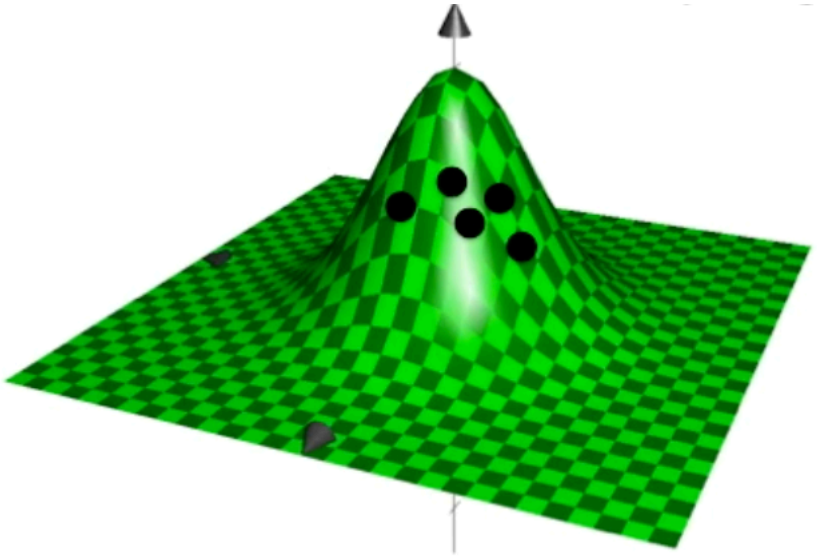
- Coloring points
  - ✓First point – yellow distance is larger than blue distance : say, 75% yellow 25% blue
  - ✓Second point – heights are same : 50% and 50% for example



Gaussian Mixtures Model – Algorithm

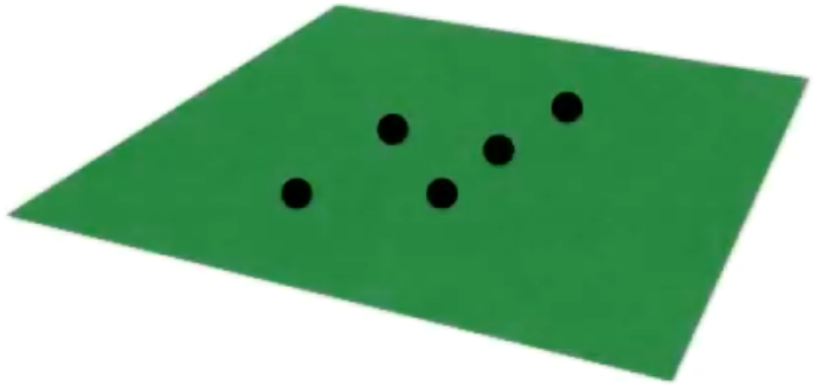
- Fitting a Gaussian
  - ✓Find the Gaussian that lifts points the most
  - ✓Looking at it from the top in flatland (left)

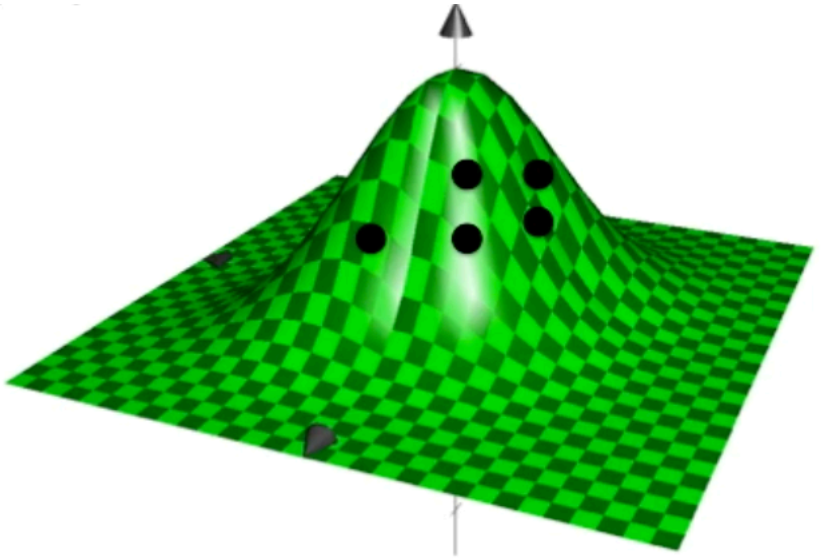




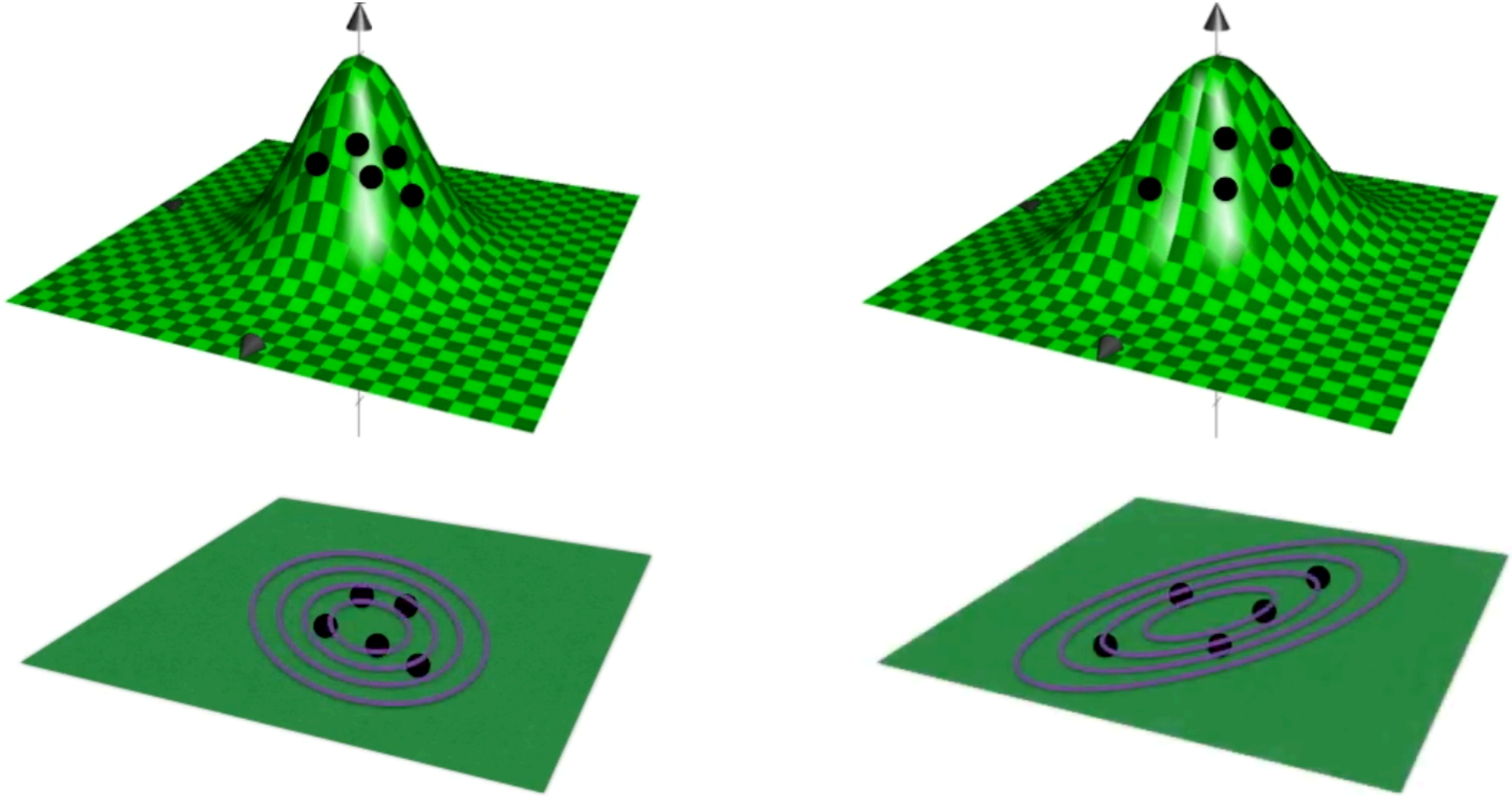
Gaussian Mixtures Model – Algorithm

- Fitting a Gaussian
  - ✓Find the Gaussian that lifts points the most
  - ✓Looking at it from the top in flatland (left)



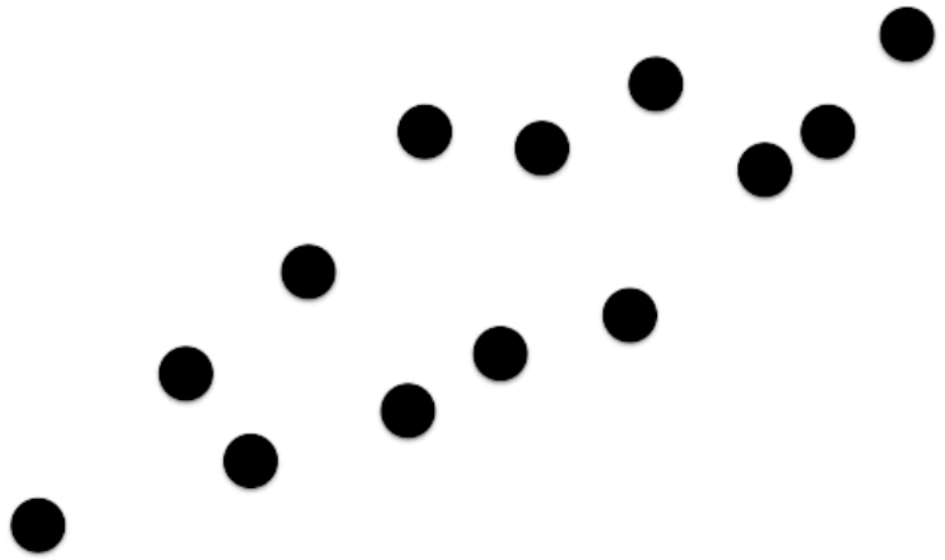


Gaussian Mixtures Model – Algorithm



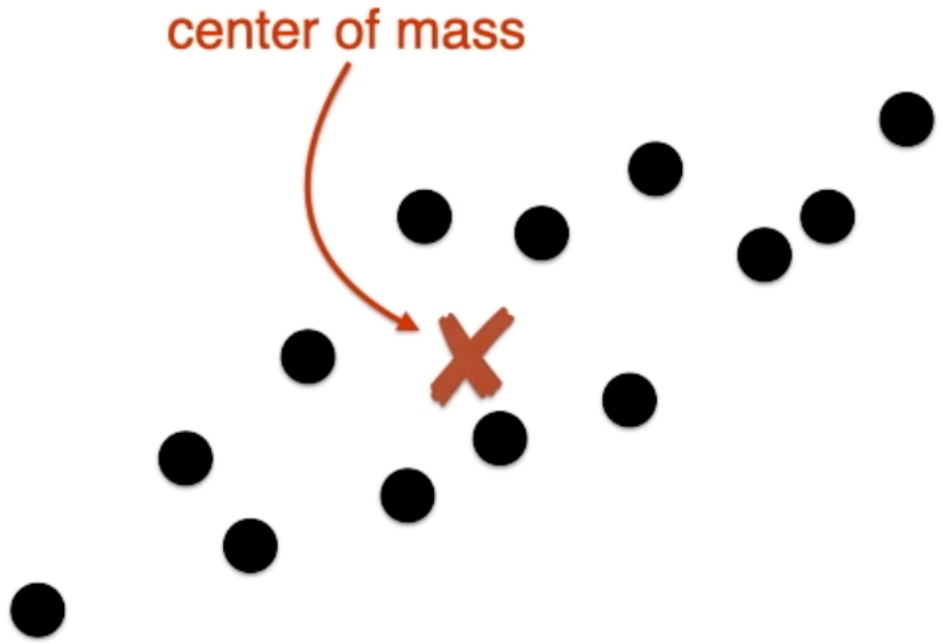
Gaussian Mixtures Model – Algorithm

- Find center of these points



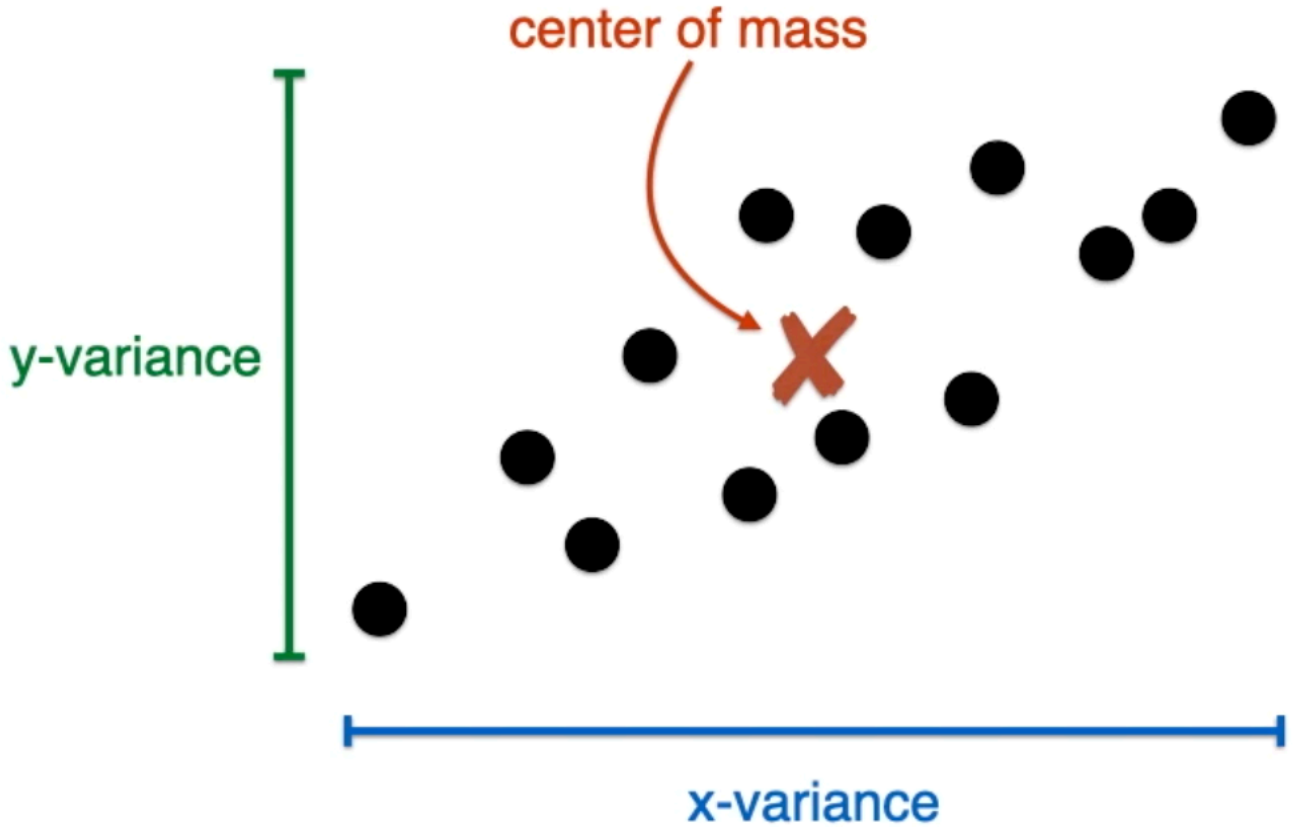
Gaussian Mixtures Model – Algorithm

- Find center of these points
  - ✓Average of X-coordinate, average of Y-coordinate



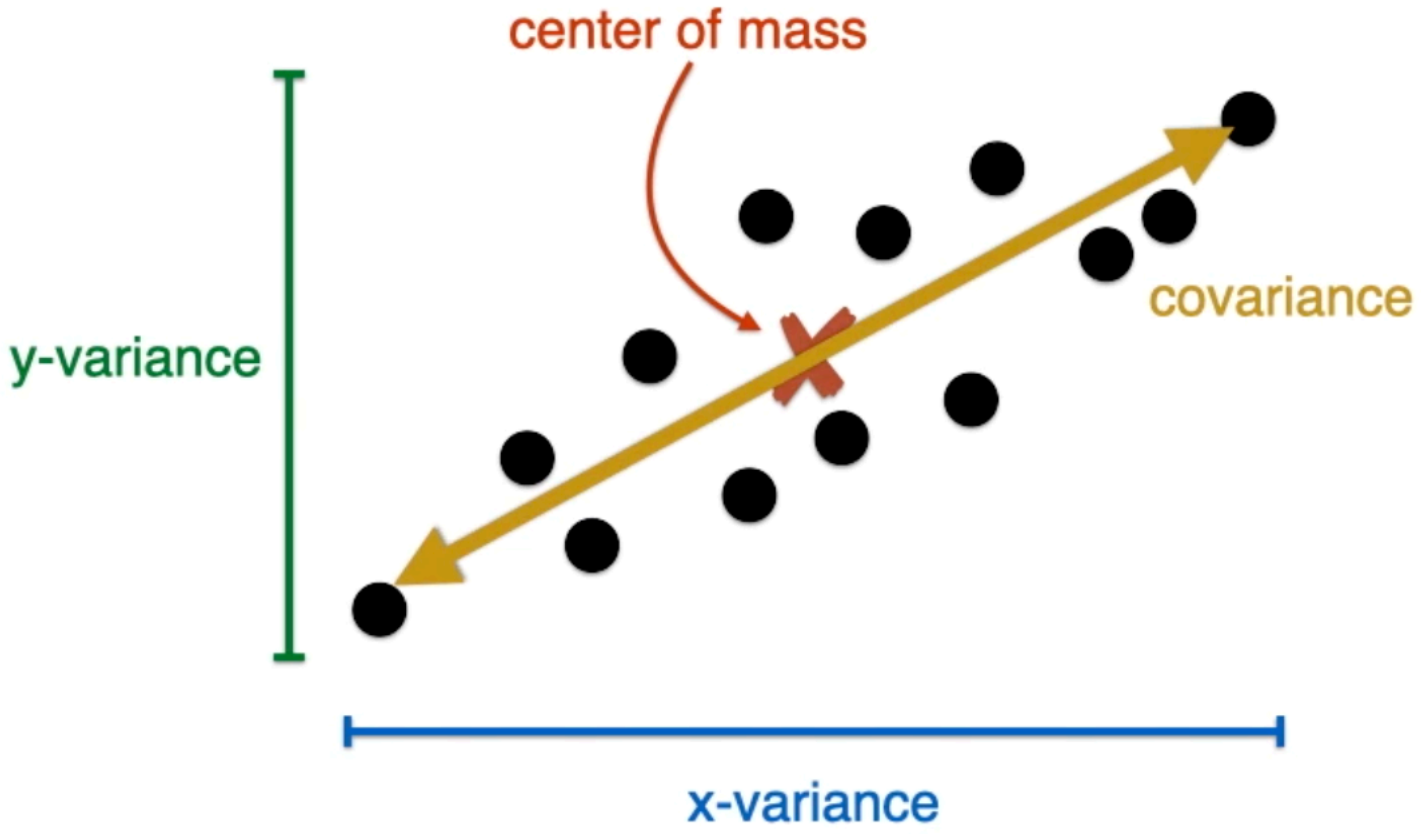
Gaussian Mixtures Model – Algorithm

- Find center of these points



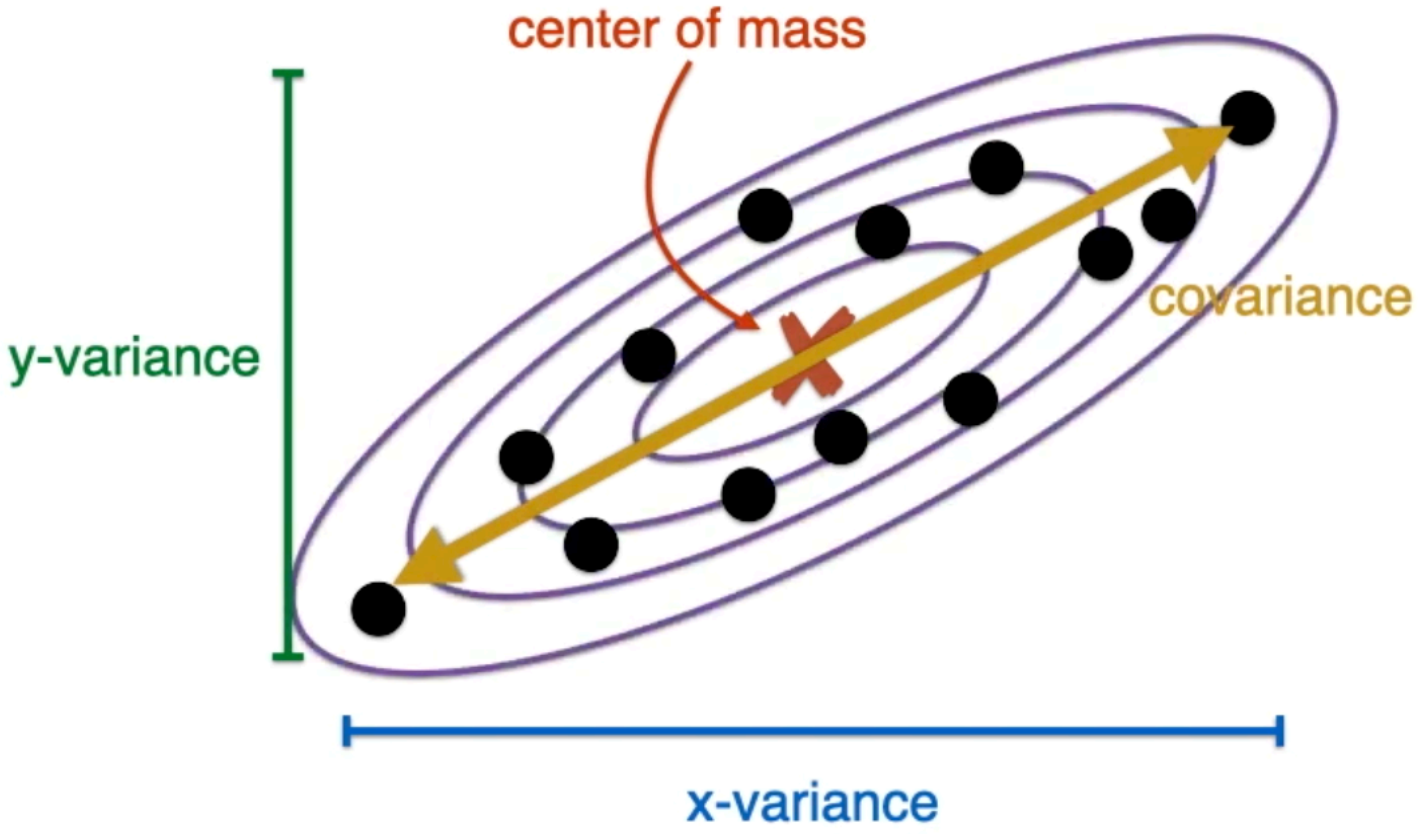
Gaussian Mixtures Model – Algorithm

- Find the x-variance and y-variance



Gaussian Mixtures Model – Algorithm

- Find the perfect Gaussian



Gaussian Mixtures Model – Algorithm

- X variance + y variance + xy covariance = all information how the dataset looks like → helps to find the perfect gaussian
- Plug into the formula:

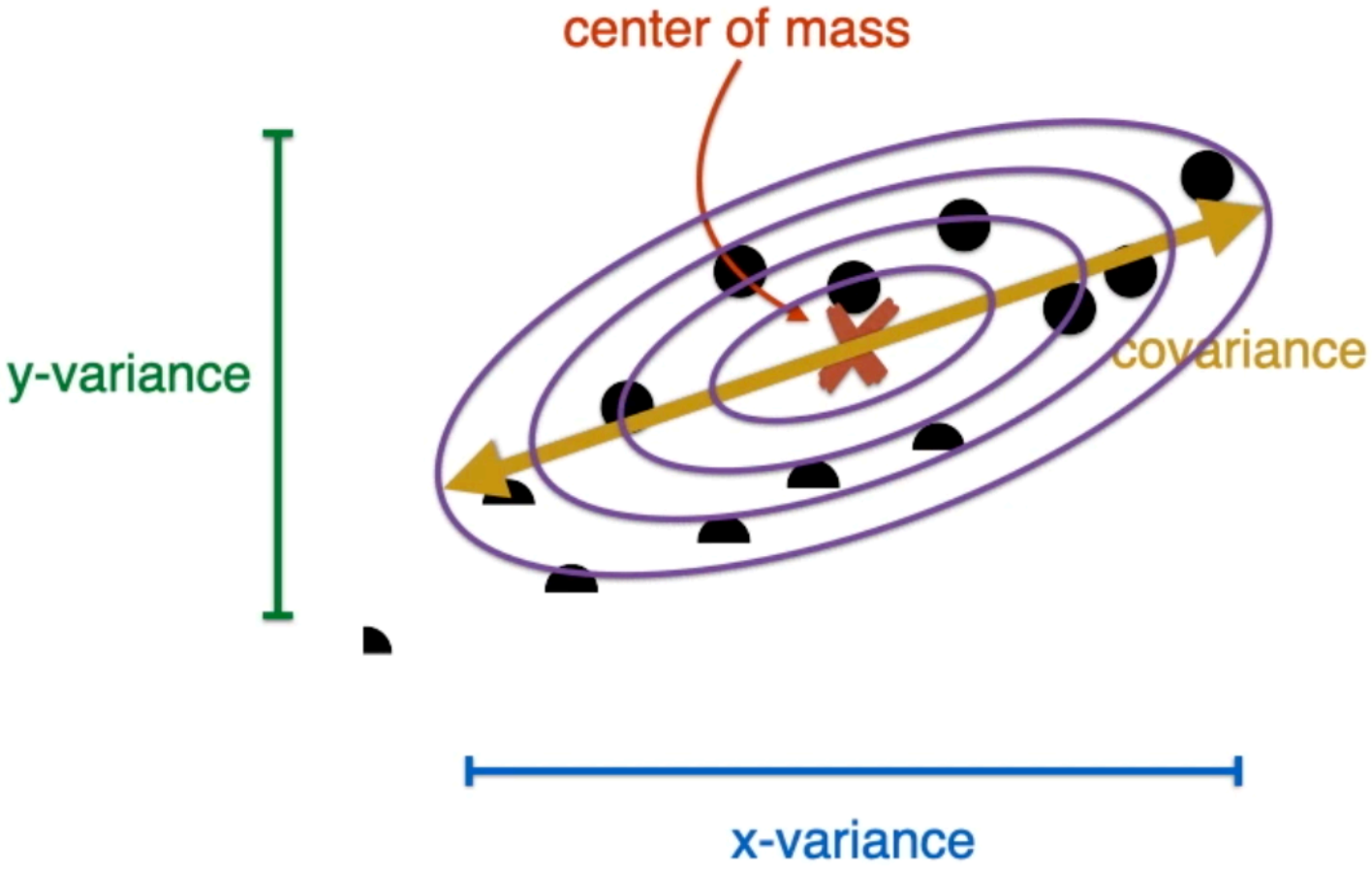
$$f(x \mid \mu, \Sigma) = \frac{1}{\sqrt{2\pi|\Sigma|}} \exp\left[-\tfrac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})'\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right] \qquad f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where, μ = average, Σ = covariance matrix $\qquad \Sigma = \begin{pmatrix} Var(x) & Cov(x,y) \\ Cov(x,y) & Var(y) \end{pmatrix}$

- Plug-in x into the formular → Height of the mountain at the point (gaussian distribution)

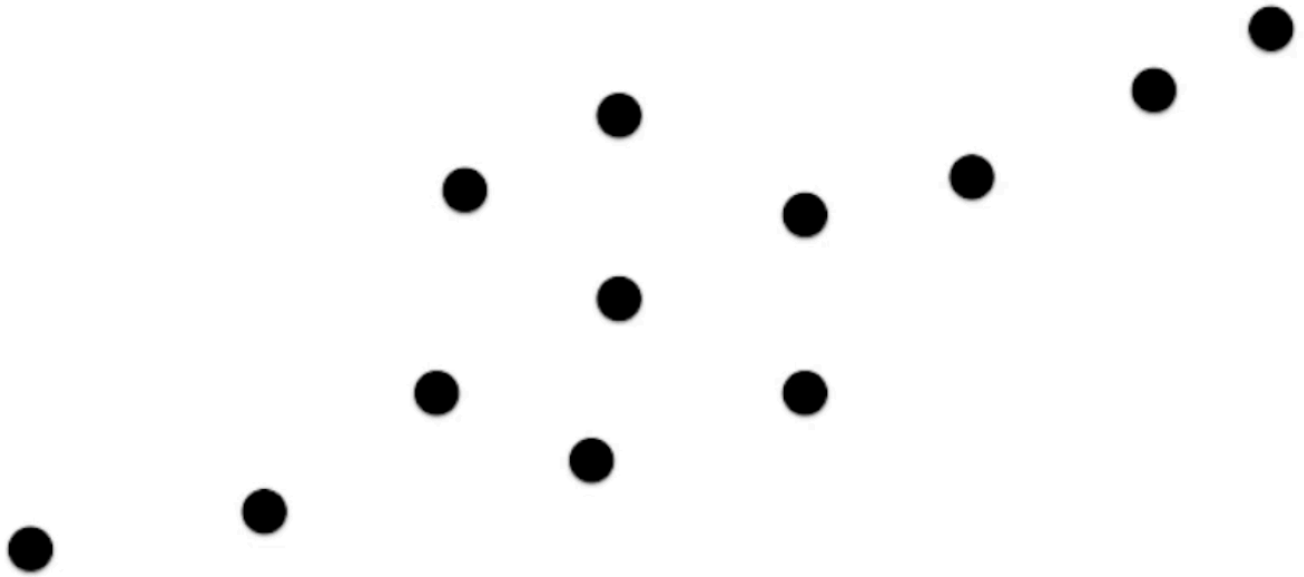
Gaussian Mixtures Model – Algorithm

- Plug the proportions of points into the formula



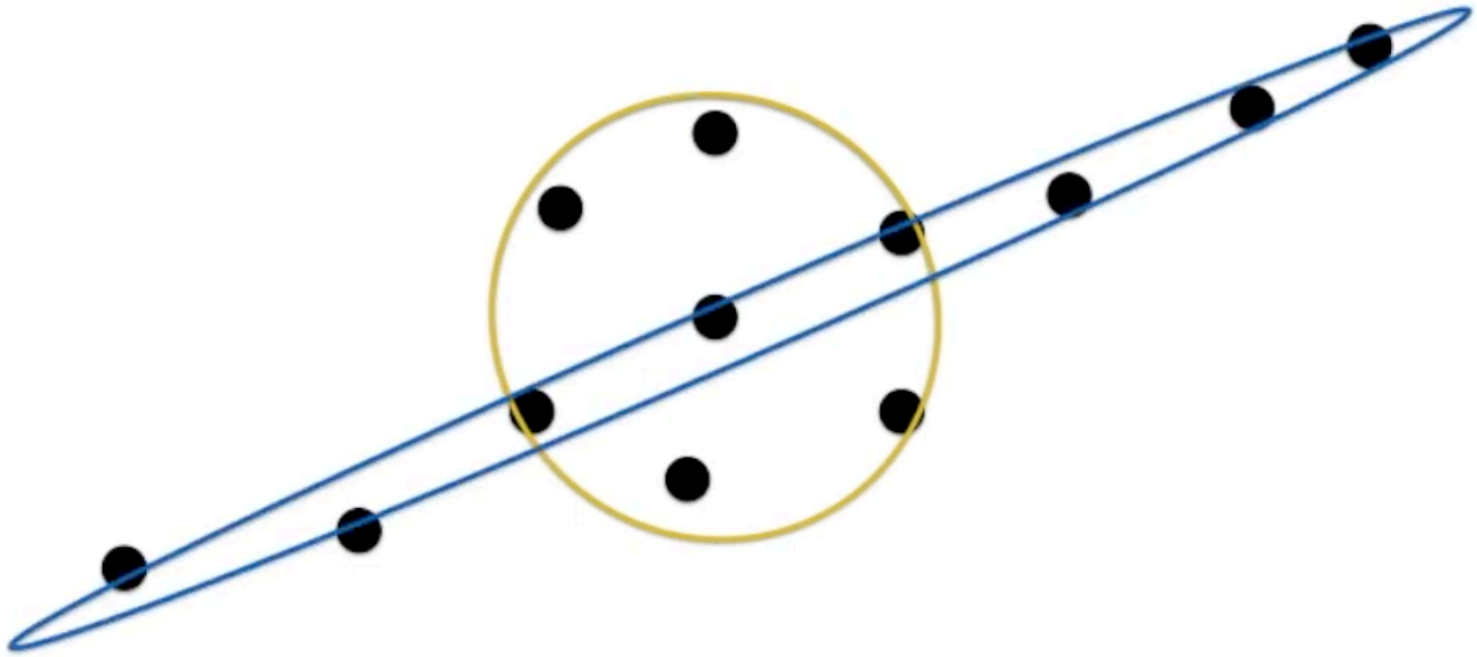
Gaussian Mixtures Model – Algorithm

- Separate this data set into two clusters



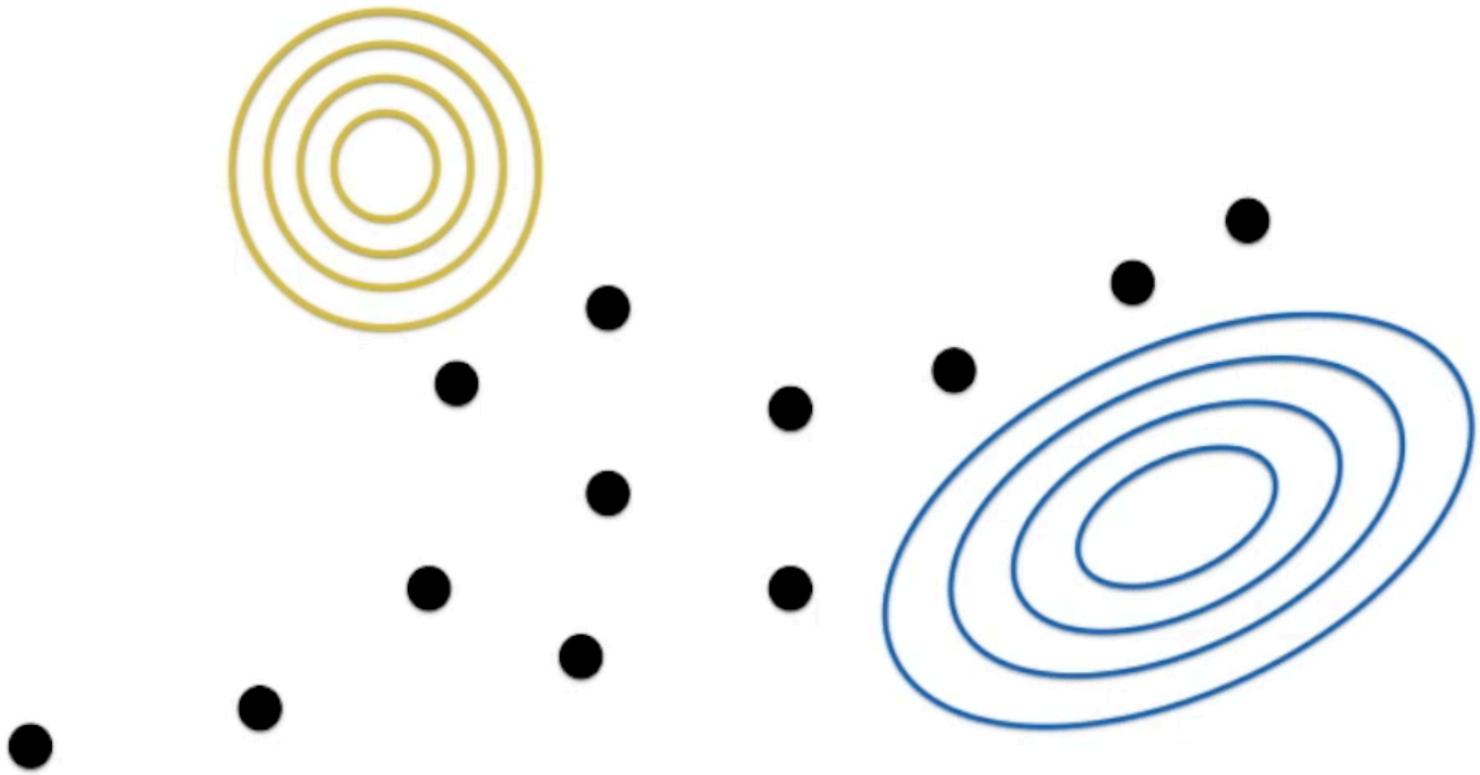
Gaussian Mixtures Model – Algorithm

- Separate this data set into two clusters
  - ✓Some points belong mostly to the circle
  - ✓some belong mostly to the line
  - ✓some to both



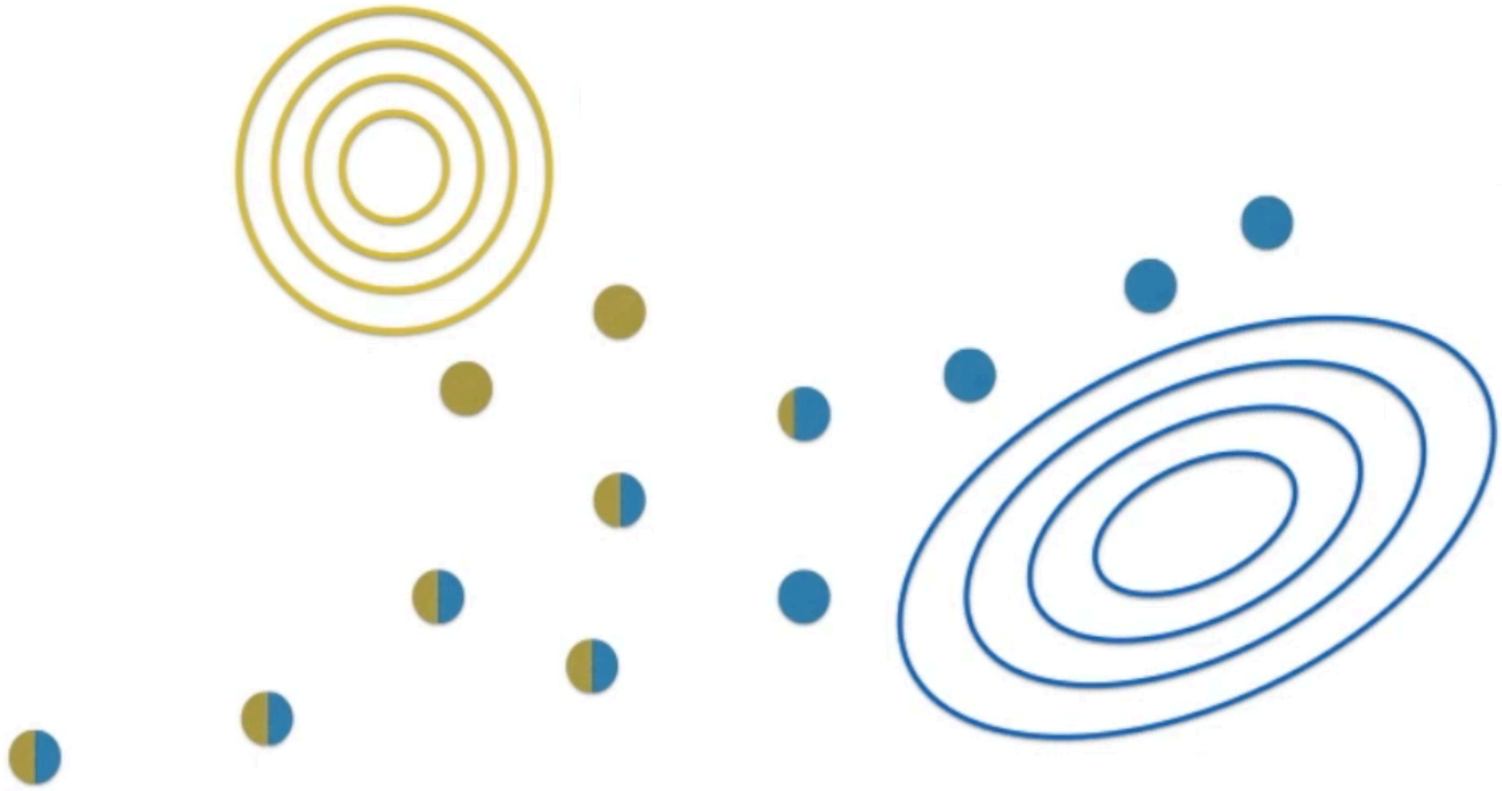
Gaussian Mixtures Model – Algorithm

- Start with random Gaussians



Gaussian Mixtures Model – Algorithm

- Color points according to Gaussians



Gaussian Mixtures Model – Algorithm

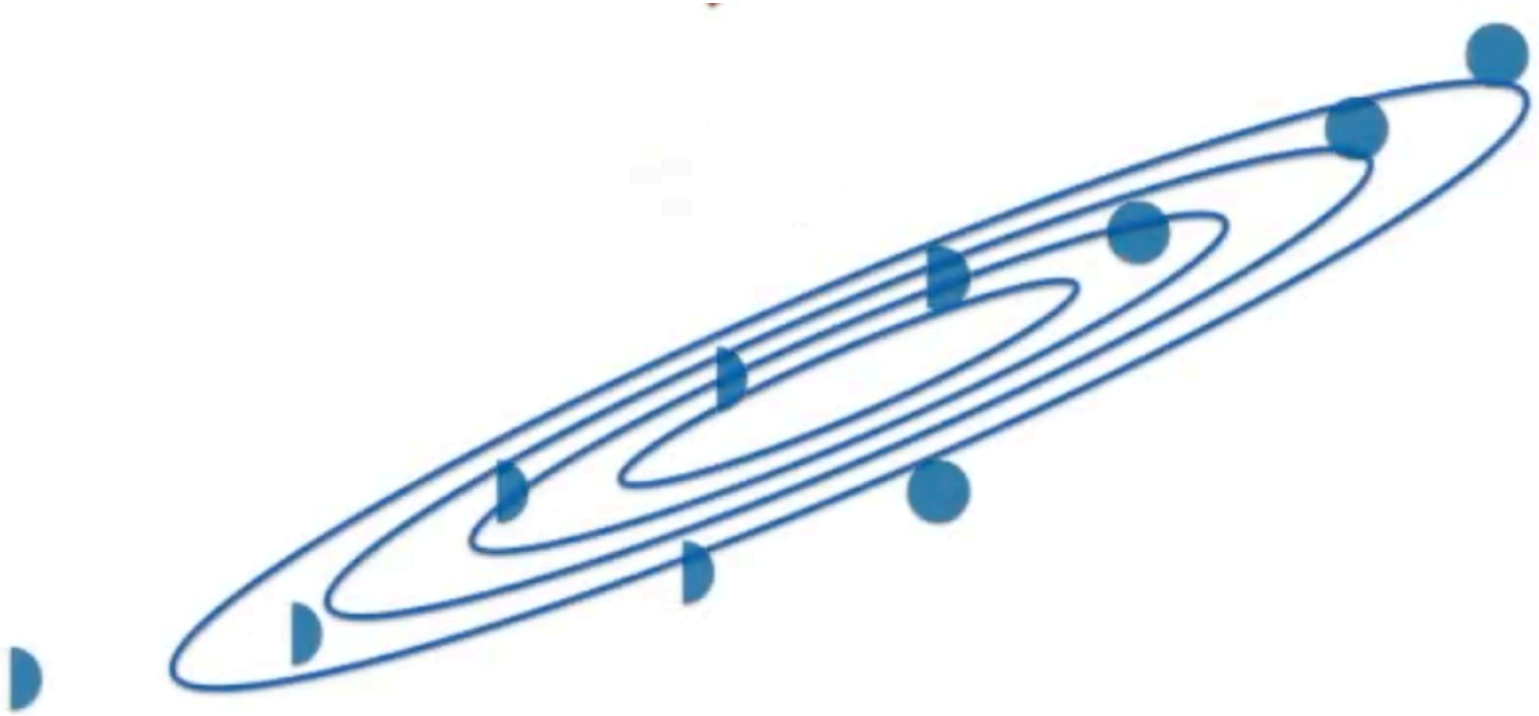
- Recalculate Gaussians according to colors

Gaussian Mixtures Model – Algorithm

- Recalculate Gaussians according to colors

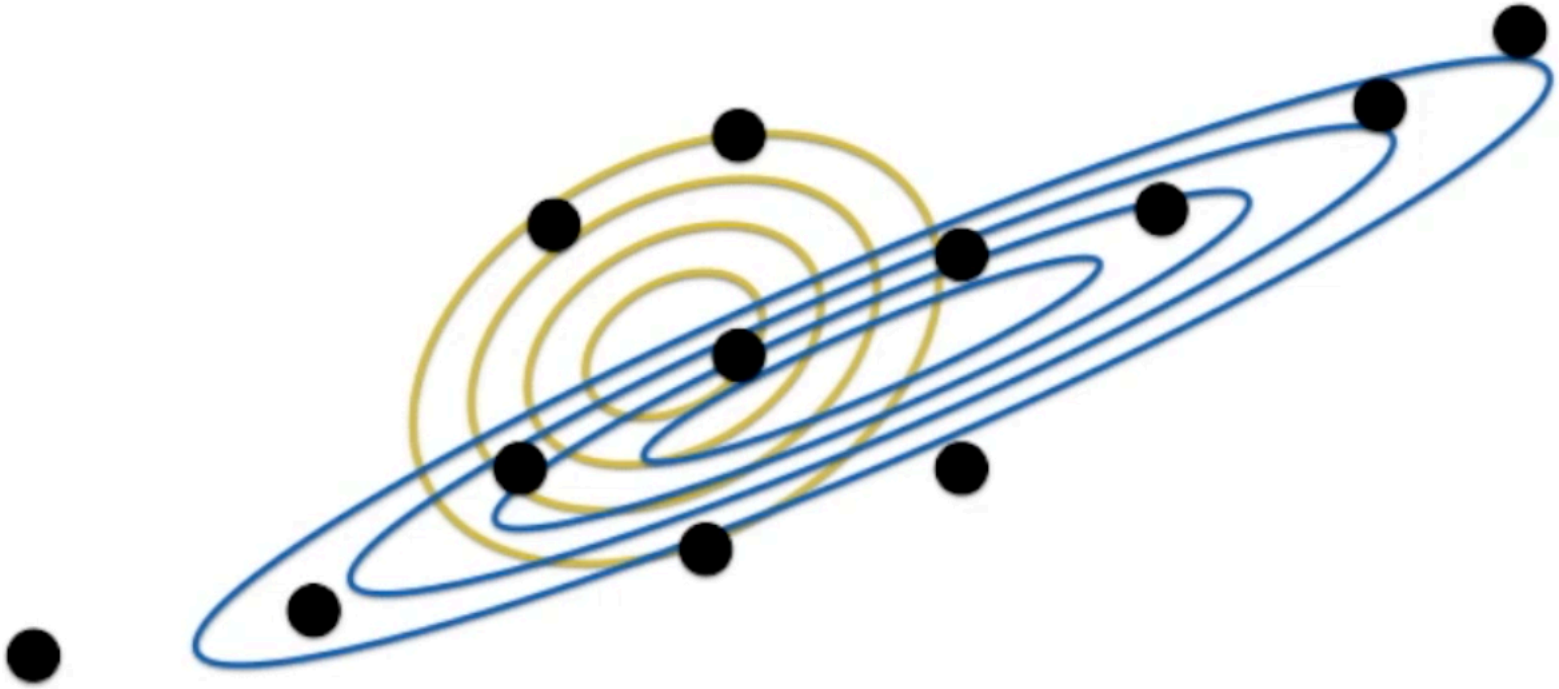
Gaussian Mixtures Model – Algorithm

- Recalculate Gaussians according to colors



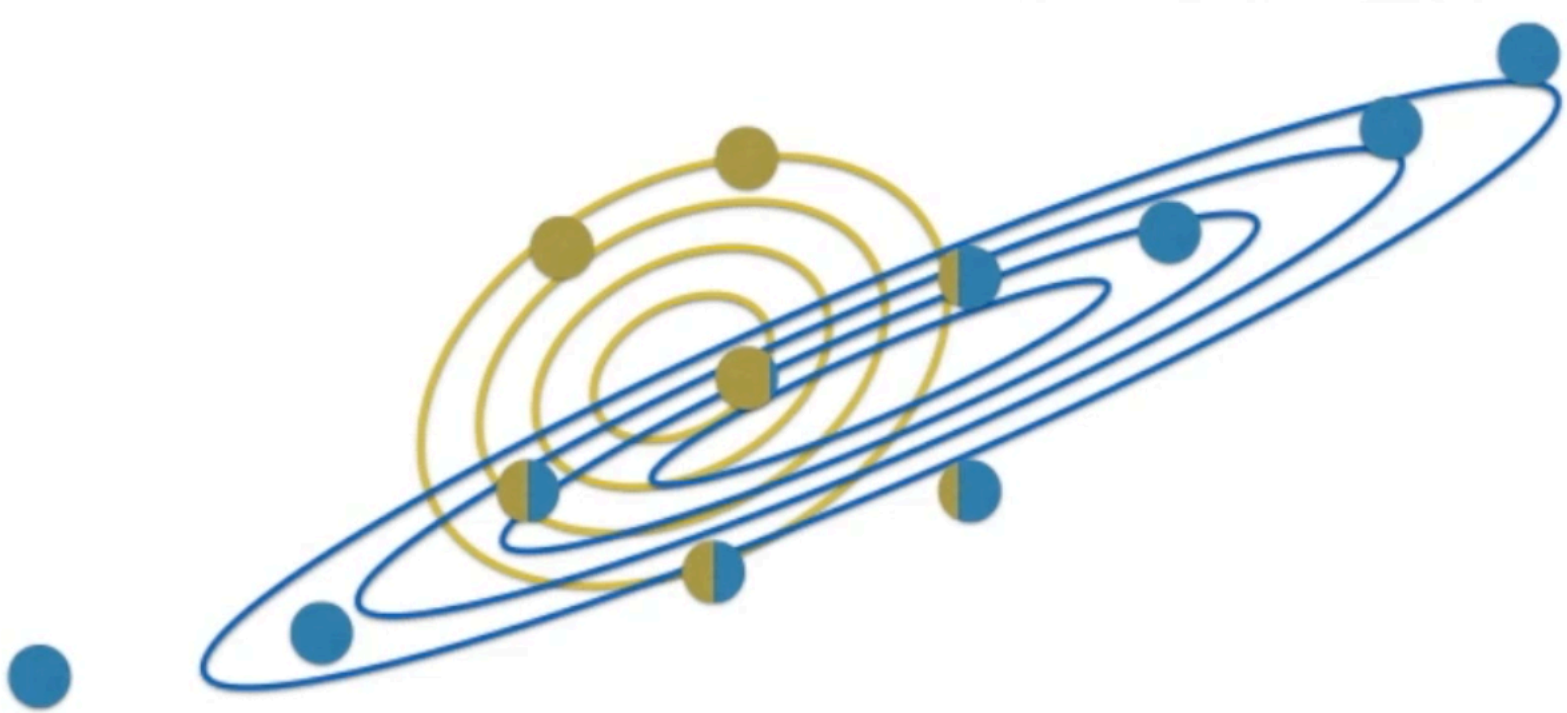
Gaussian Mixtures Model – Algorithm

- Recalculate Gaussians according to colors



Gaussian Mixtures Model – Algorithm

- Color points according to Gaussians

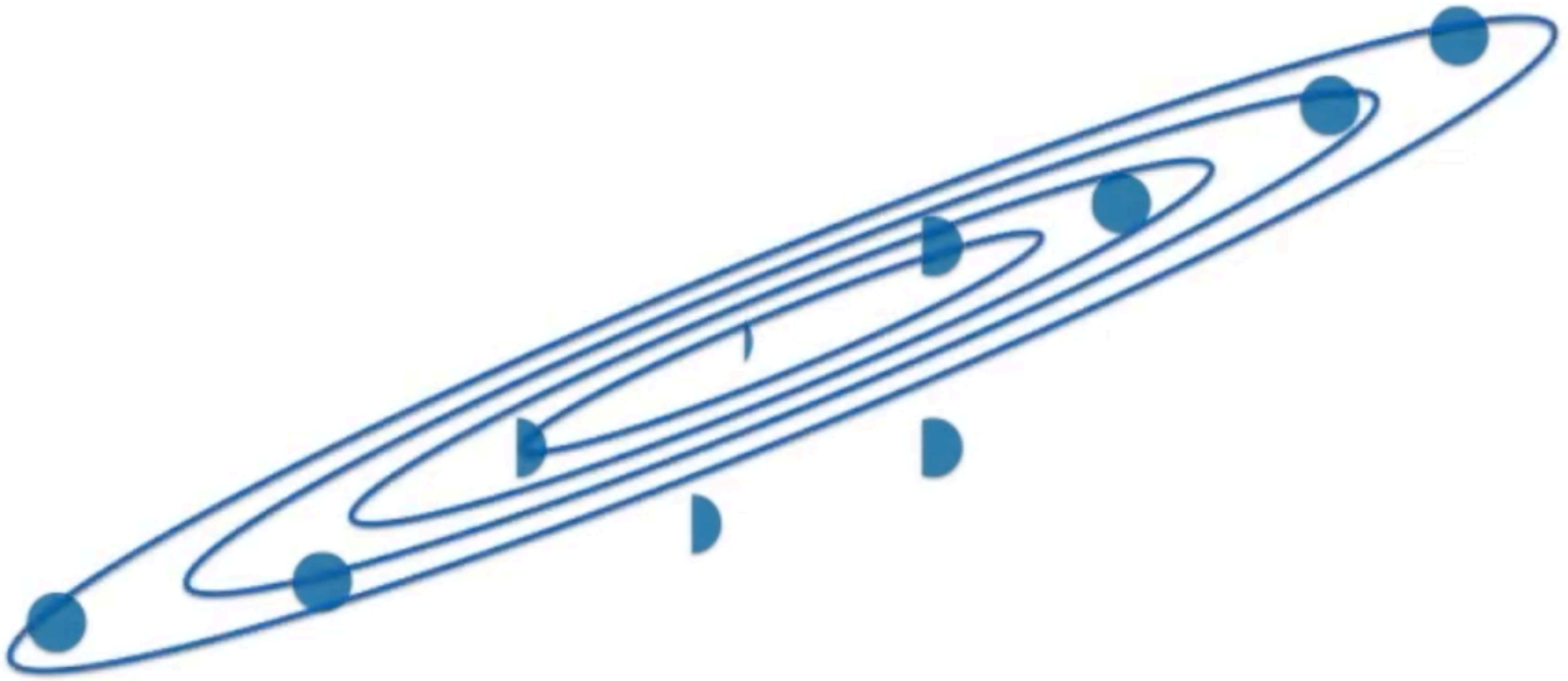


Gaussian Mixtures Model – Algorithm

- Recalculate Gaussians according to colors

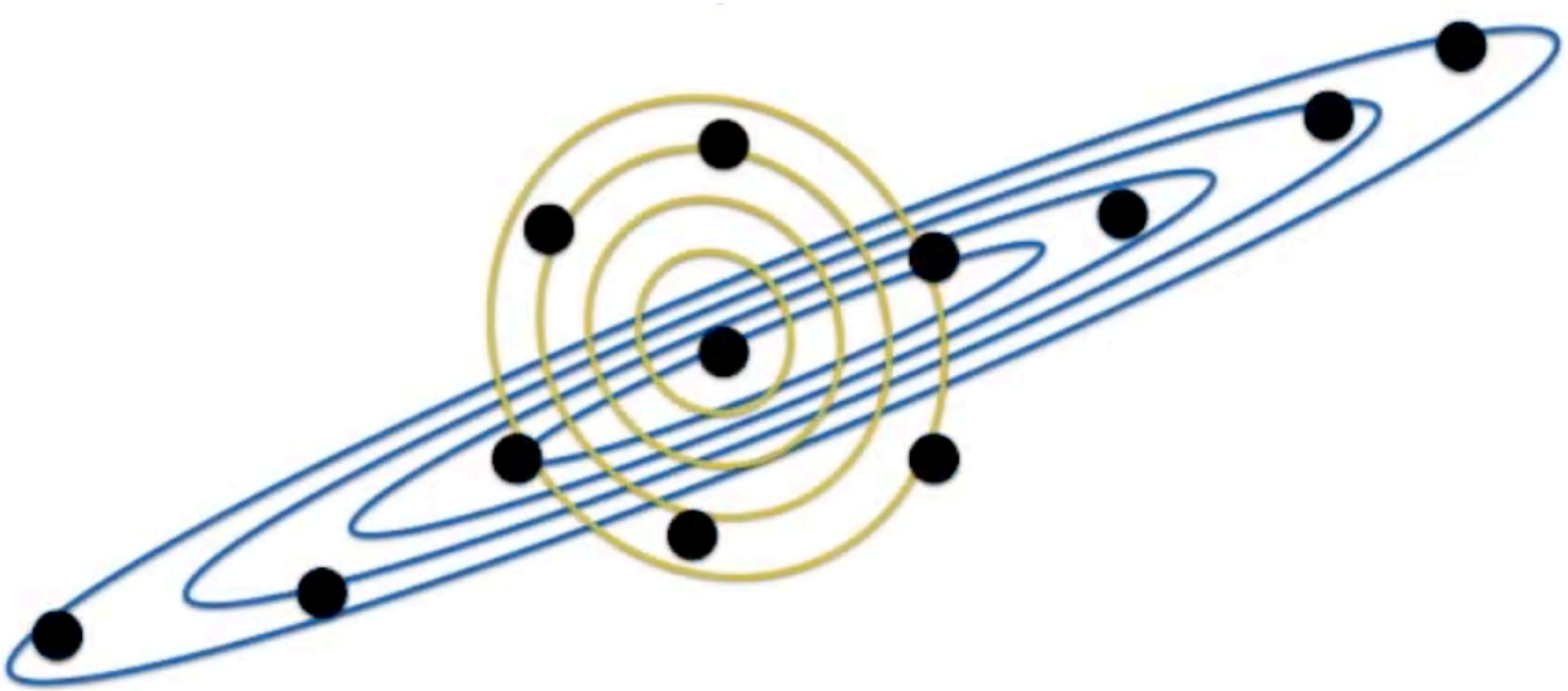
Gaussian Mixtures Model – Algorithm

- Recalculate Gaussians according to colors



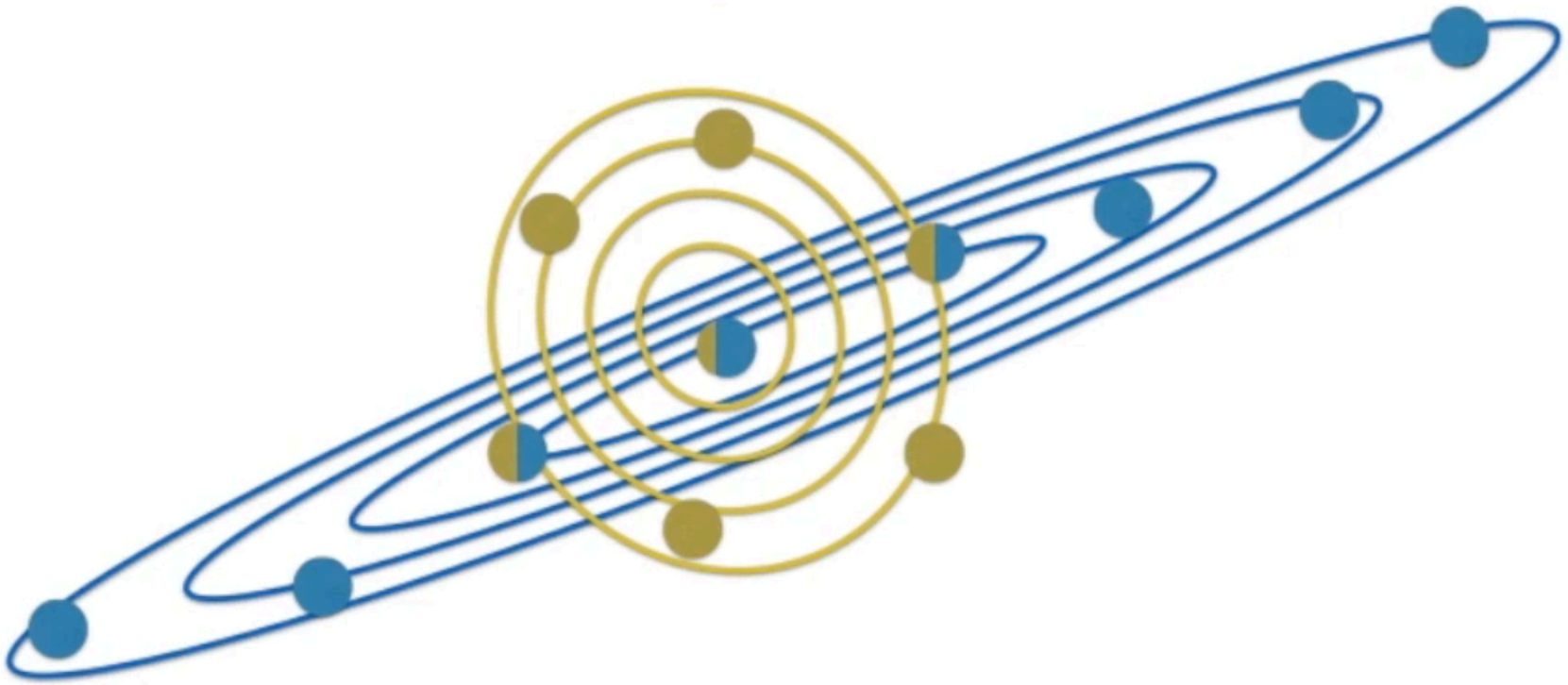
Gaussian Mixtures Model – Algorithm

- Recalculate Gaussians according to colors



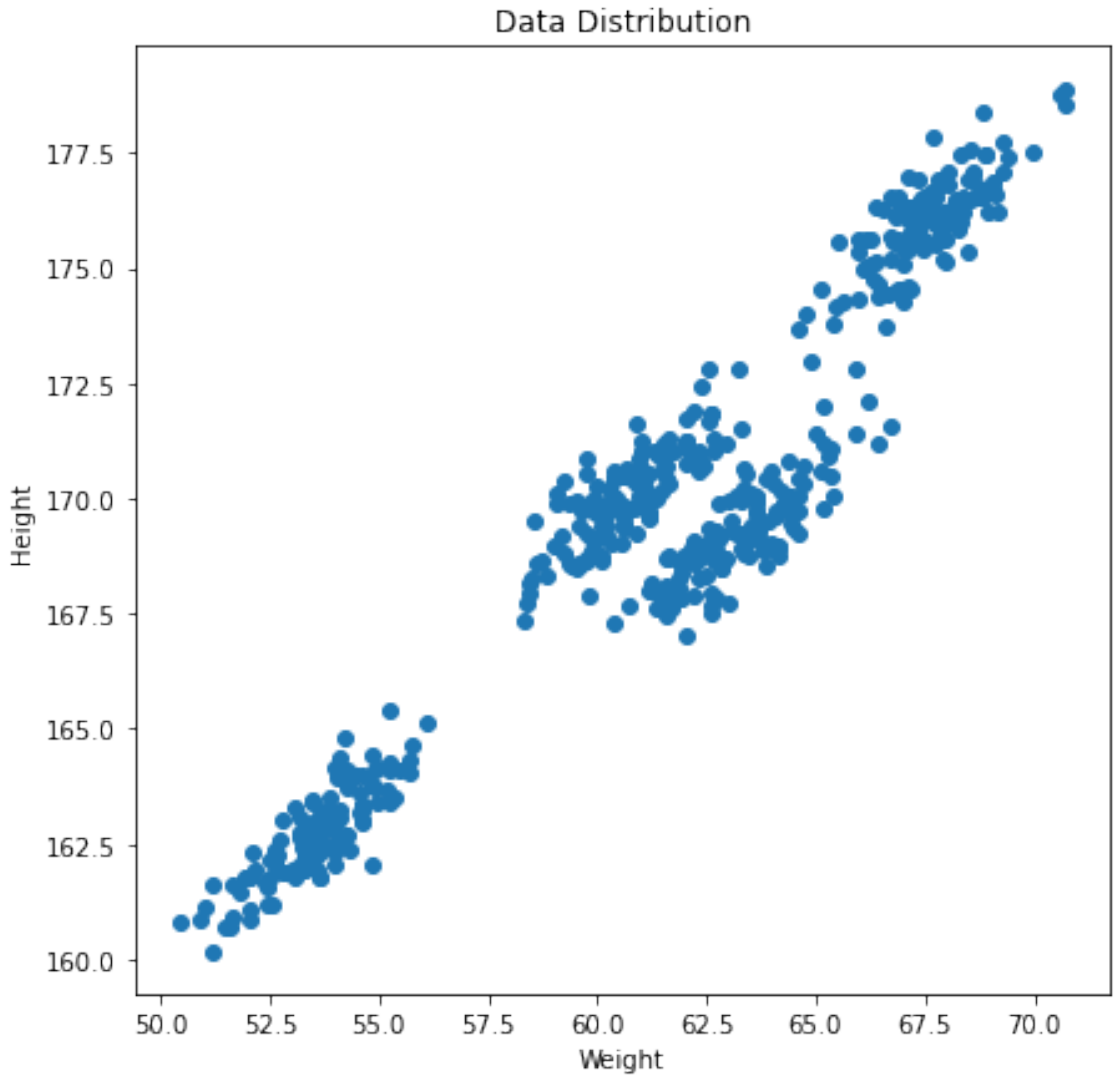
Gaussian Mixtures Model – Algorithm

- Reached ending point



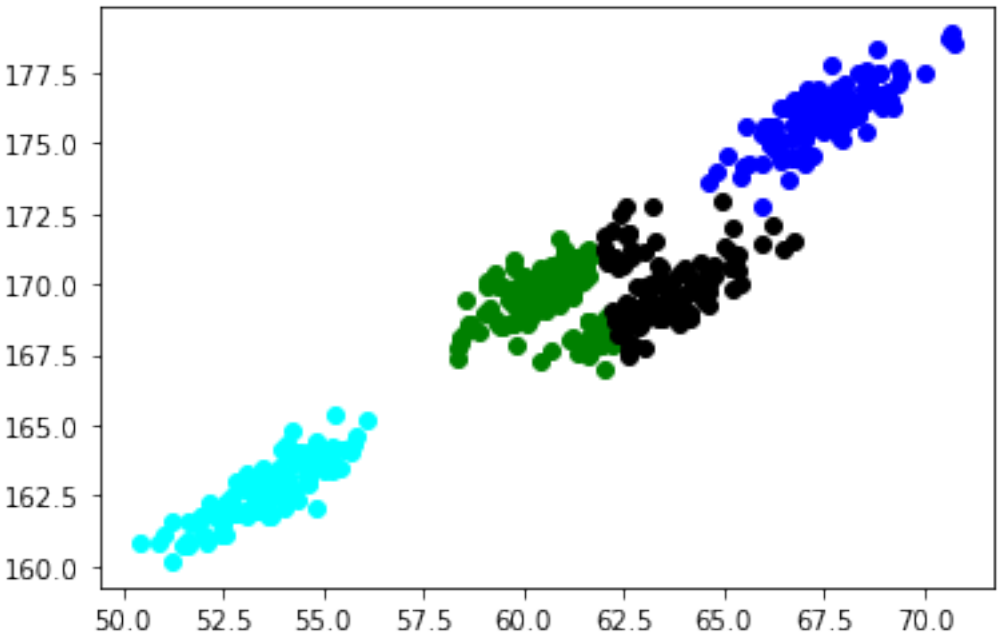
Gaussian Mixtures Model – Implementation

```
from sklearn.mixture import GaussianMixture
from matplotlib.patches import Ellipse
import pandas as pd

data = pd.read_csv('Clustering_gmm.csv')

plt.figure(figsize=(7,7))
plt.scatter(data["Weight"],data["Height"])
plt.xlabel('Weight')
plt.ylabel('Height')
plt.title('Data Distribution')
plt.show()
```



Gaussian Mixtures Model – Implementation

```
#training k-means model
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4)
kmeans.fit(data)

#predictions from kmeans
pred = kmeans.predict(data)
frame = pd.DataFrame(data)
frame['cluster'] = pred
frame.columns = ['Weight', 'Height', 'cluster']

#plotting results
color=['blue','green','cyan', 'black']
for k in range(0,4):
    data = frame[frame["cluster"]==k]
    plt.scatter(data["Weight"],data["Height"],c=color[k])
plt.show()
```



Gaussian Mixtures Model – Implementation

```
import pandas as pd
data = pd.read_csv('Clustering_gmm.csv')

# training gaussian mixture model
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=4)
gmm.fit(data)

#predictions from gmm
labels = gmm.predict(data)
frame = pd.DataFrame(data)
frame['cluster'] = labels
frame.columns = ['Weight', 'Height', 'cluster']

color=['blue','green','cyan', 'black']
for k in range(0,4):
    data = frame[frame["cluster"]==k]
    plt.scatter(data["Weight"],data["Height"],c=color[k])
plt.show()
```