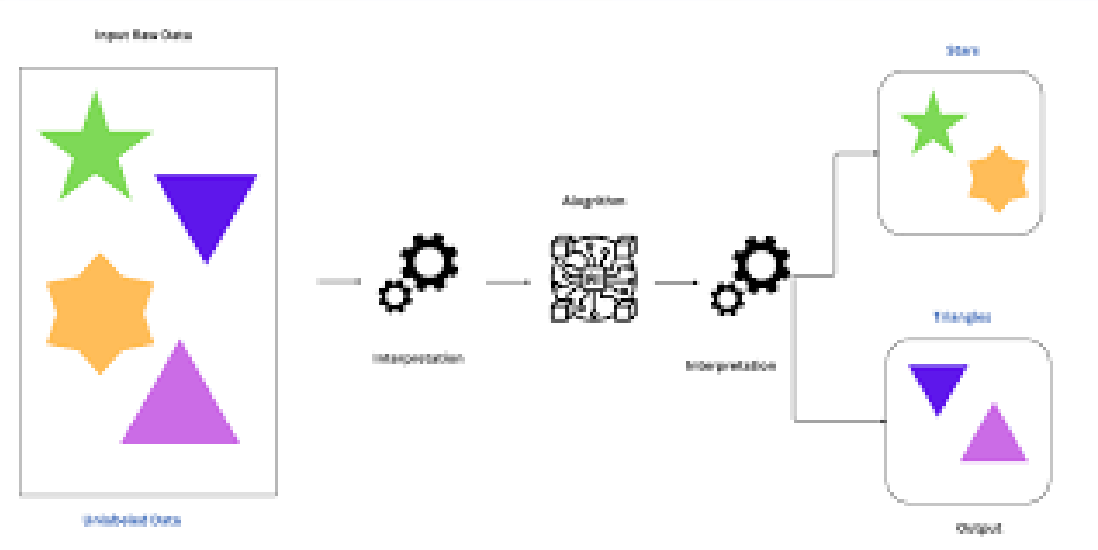
AI in Biomedical Data

Dr. M.B. Khodabakhshi
Amir Hossein Fouladi
Alireza Javadi

github.com/mbkhodabakhshi/AI_in_BiomedicalData

ستاد توسعه فناوری های
هوش مصنوعی و رباتیک



یادگیری ماشین در زیست پزشکی

Chapter 9. Unsupervised Learning Techniques

دکتر محمدباقر خدابخشی
mb.khodabakhshi@gmail.com

Presenter: Dr.Khodabakhshi

Unsupervised Learning Techniques

Wouldn't it be great if the algorithm could just exploit the unlabeled data without needing humans to label every picture? Enter unsupervised learning.

Clustering

The goal is to group similar instances together into *clusters*. Clustering is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and more.

Anomaly detection

The objective is to learn what "normal" data looks like, and then use that to detect abnormal instances, such as defective items on a production line or a new trend in a time series.

Density estimation

This is the task of estimating the *probability density function* (PDF) of the random process that generated the dataset. Density estimation is commonly used for anomaly detection: instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualization.

Clustering

It is the task of identifying similar instances and assigning them to *clusters*, or groups of similar instances.

Just like in classification, each instance gets assigned to a group. However, unlike classification, clustering is an unsupervised task. Consider Figure 9-1: on the left is the iris dataset (introduced in Chapter 4), where each instance's species (i.e., its class) is represented with a different marker.

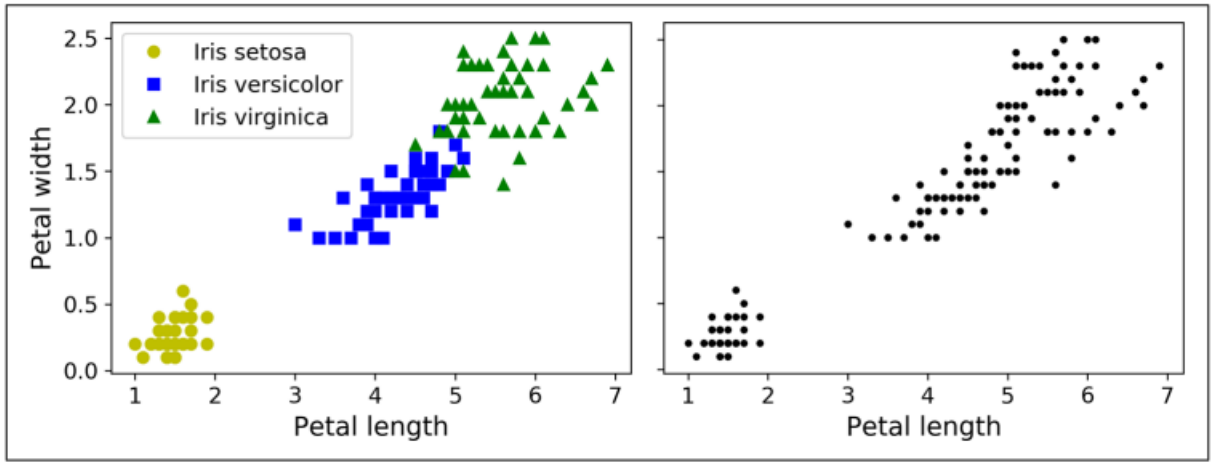
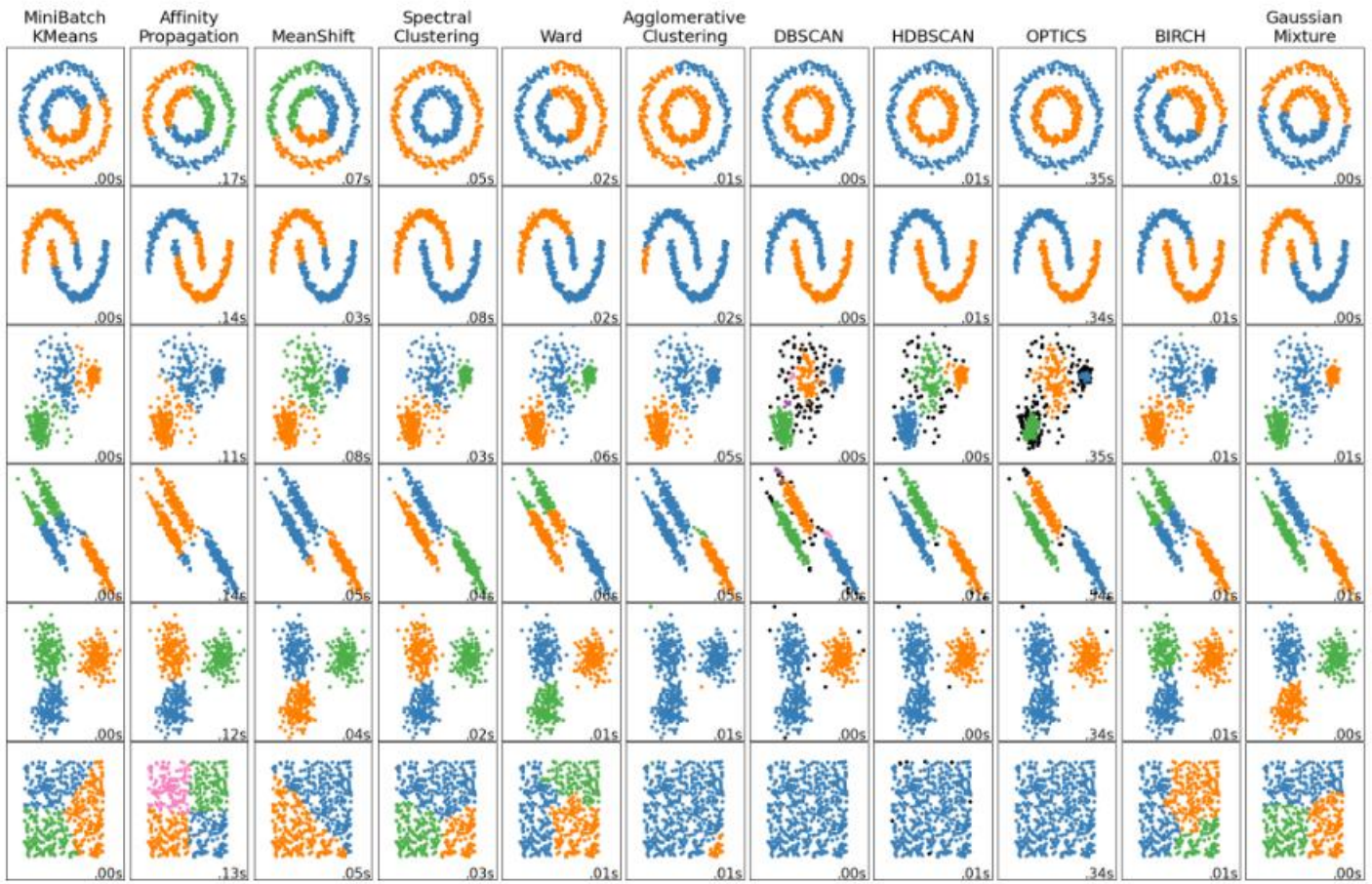


Figure 9-1. Classification (left) versus clustering (right)

Overview of clustering methods in Scikit-Learn



A comparison of the clustering algorithms in scikit-learn

In this section, we will look at some popular clustering algorithms, and explore some of their applications, such as nonlinear dimensionality reduction, semi-supervised learning, and anomaly detection.

K-Means

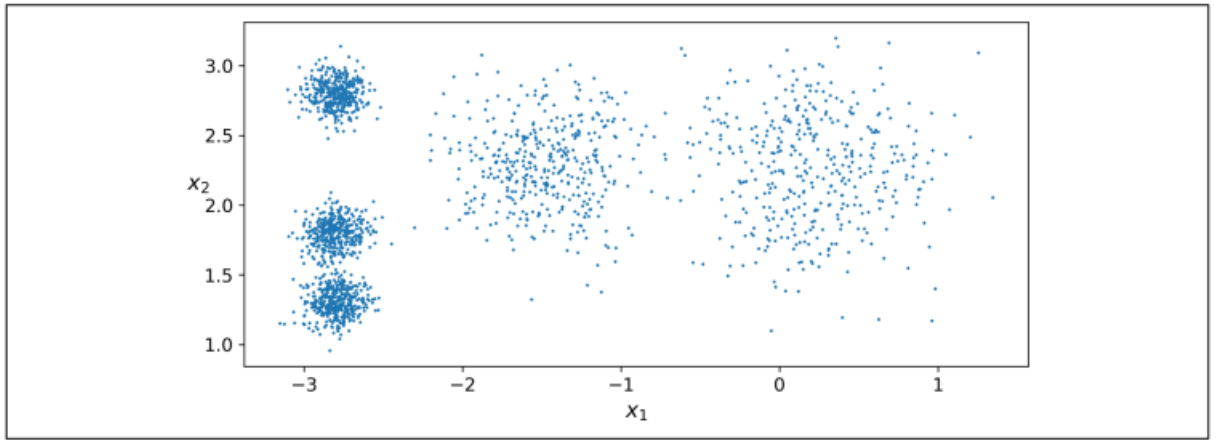


Figure 9-2. An unlabeled dataset composed of five blobs of instances

Let's train a K-Means clusterer on this dataset. It will try to find each blob's center and assign each instance to the closest blob:

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

The *KMeans* instance preserves a copy of the labels of the instances it was trained on, available via the *labels_* instance variable:

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

We can also take a look at the five centroids that the algorithm found:

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

You can easily assign new instances to the cluster whose centroid is closest:

```
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

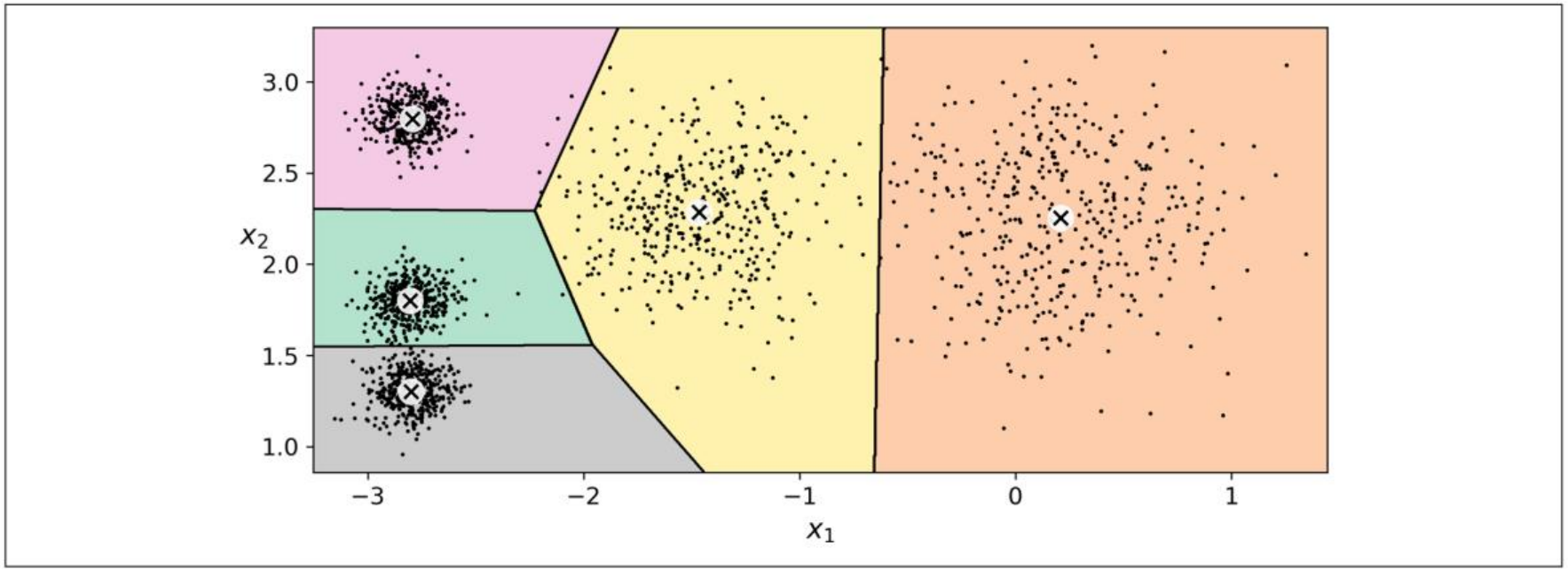


Figure 9-3. K-Means decision boundaries (Voronoi tessellation)

- Instead of assigning each instance to a single cluster, which is called *hard clustering*, it can be useful to give each instance a score per cluster, which is called *soft clustering*.
- The score can be the distance between the instance and the centroid;
- In the `KMeans` class, the `transform()` method measures the distance from each instance to every centroid:

```
>>> kmeans.transform(X_new)
array([[2.81093633, 0.32995317, 2.9042344 , 1.49439034, 2.88633901],
       [5.80730058, 2.80290755, 5.84739223, 4.4759332 , 5.84236351],
       [1.21475352, 3.29399768, 0.29040966, 1.69136631, 1.71086031],
       [0.72581411, 3.21806371, 0.36159148, 1.54808703, 1.21567622]])
```

The K-Means algorithm

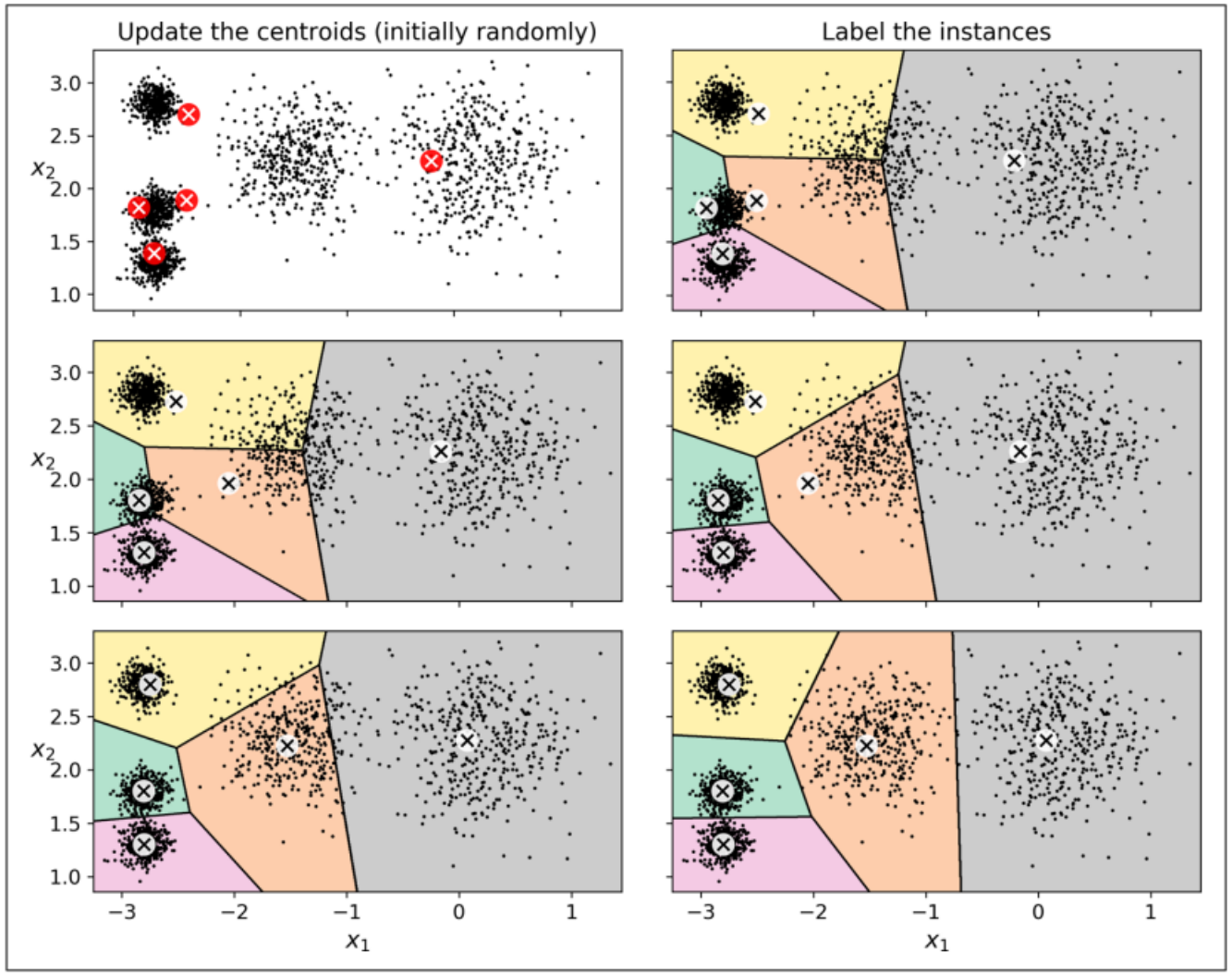


Figure 9-4. The K-Means algorithm

Given a set of observations $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k ($\leq n$) sets $\mathbf{S} = \{S_1, S_2, \ldots, S_k\}$ so as to minimize the within-cluster sum of squares (WCSS) (i.e. variance). Formally, the objective is to find:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i$$

where $\boldsymbol{\mu}_i$ is the mean (also called centroid) of points in S_i, i.e.

$$\boldsymbol{\mu}_i = \frac{1}{|S_i|} \sum_{\mathbf{x} \in S_i} \mathbf{x},$$

$|S_i|$ is the size of S_i, and $\|\cdot\|$ is the usual L^2 norm . This is equivalent to minimizing the pairwise squared deviations of points in the same cluster:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \frac{1}{|S_i|} \sum_{\mathbf{x}, \mathbf{y} \in S_i} \|\mathbf{x} - \mathbf{y}\|^2$$

Although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): whether it does or not depends on the centroid initialization. Figure 9-5 shows two suboptimal solutions that the algorithm can converge to if you are not lucky with the random initialization step.

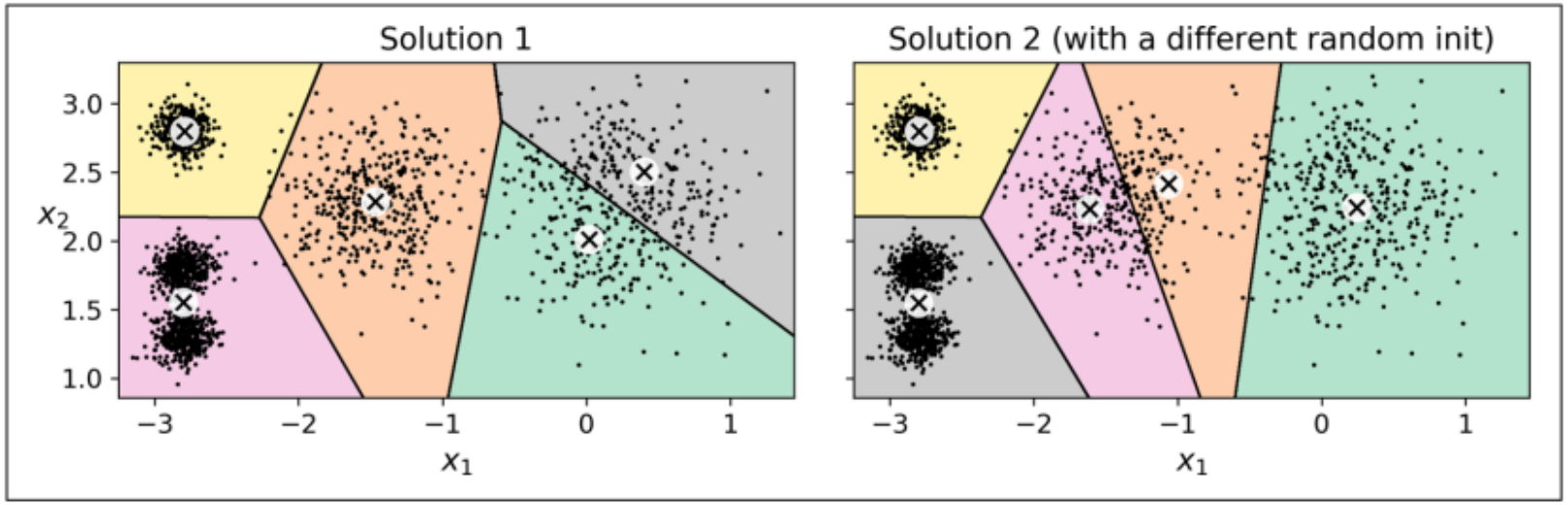


Figure 9-5. Suboptimal solutions due to unlucky centroid initializations

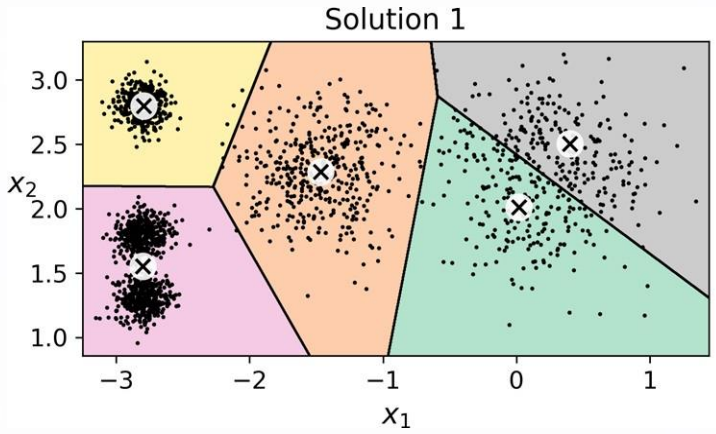
Centroid initialization methods

If you happen to know approximately where the centroids should be:

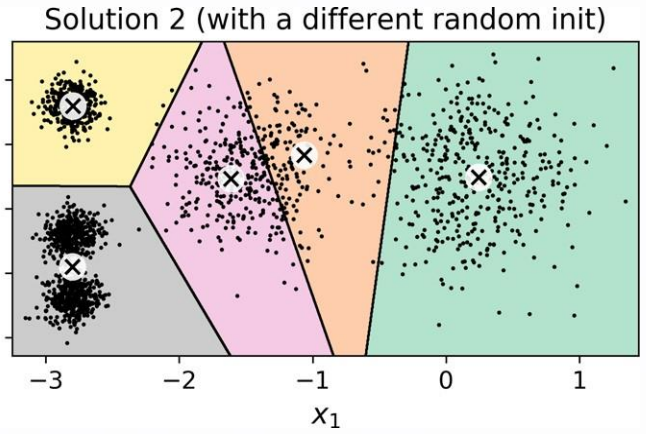
```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

- Another solution is to run the algorithm multiple times with different random initializations and keep the best solution. The number of random initializations is controlled by the `n_init` hyperparameter:
- by default, it is equal to `10`, which means that the whole algorithm described earlier runs 10 times when you call `fit()`, and Scikit-Learn keeps the best solution.
- But how exactly does it know which solution is the best? It uses a performance metric! That metric is called the model's *inertia* which is the mean squared distance between each instance and its closest centroid.

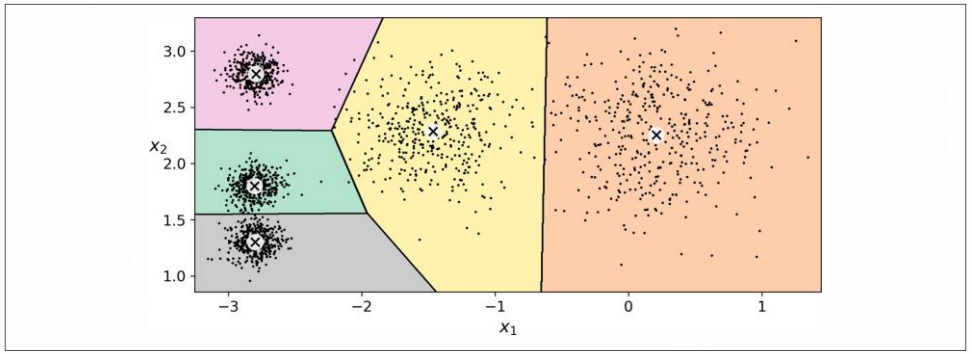
$$J = \sum_{k=1}^{K} \sum_{i=1}^{n_k} ||x_i - \mu_k||^2$$



Model's inertia = 223.3



Model's inertia = 237.5



Model's inertia = 211.6

The `KMeans` class runs the algorithm `n_init` times and keeps the model with the lowest inertia. a model's inertia is accessible via the `inertia_` instance variable

```
>>> kmeans.inertia_
211.59853725816856
```

K-Means++

- An important improvement to the K-Means algorithm, *K-Means++*, was proposed in a 2006 paper by David Arthur and Sergei Vassilvitskii.
- They introduced a smarter initialization step that tends to select centroids that are distant from one another,
- This improvement makes the K-Means algorithm much less likely to converge to a suboptimal solution.
- The `Kmeans` class uses this initialization method by default. If you want to force it to use the original method (i.e., picking *k* instances randomly to define the initial centroids), then you can set the `init` hyperparameter to `"random"`.

1. Take one centroid $\mathbf{c}^{(1)}$, chosen uniformly at random from the dataset.
2. Take a new centroid $\mathbf{c}^{(i)}$, choosing an instance $\mathbf{x}^{(i)}$ with probability $D\left(\mathbf{x}^{(i)}\right)^2 / \sum_{j=1}^{m} D\left(\mathbf{x}^{(j)}\right)^2$, where $D(\mathbf{x}^{(i)})$ is the distance between the instance $\mathbf{x}^{(i)}$ and the closest centroid that was already chosen. This probability distribution ensures that instances farther away from already chosen centroids are much more likely be selected as centroids.
3. Repeat the previous step until all *k* centroids have been chosen.

mini-batch K-Means

Instead of using the full dataset at each iteration, the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration.

This speeds up the algorithm typically by a factor of three or four and makes it possible to cluster huge datasets that do not fit in memory. Scikit-Learn implements this algorithm in the `MiniBatchKMeans` class.

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

Although the Mini-batch K-Means algorithm is much faster than the regular K- Means algorithm, its inertia is generally slightly worse, especially as the number of clusters increases

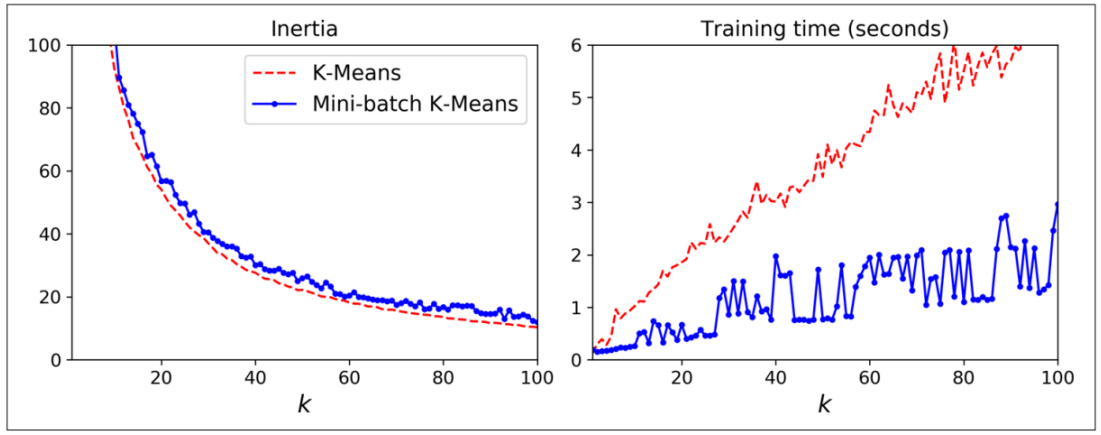
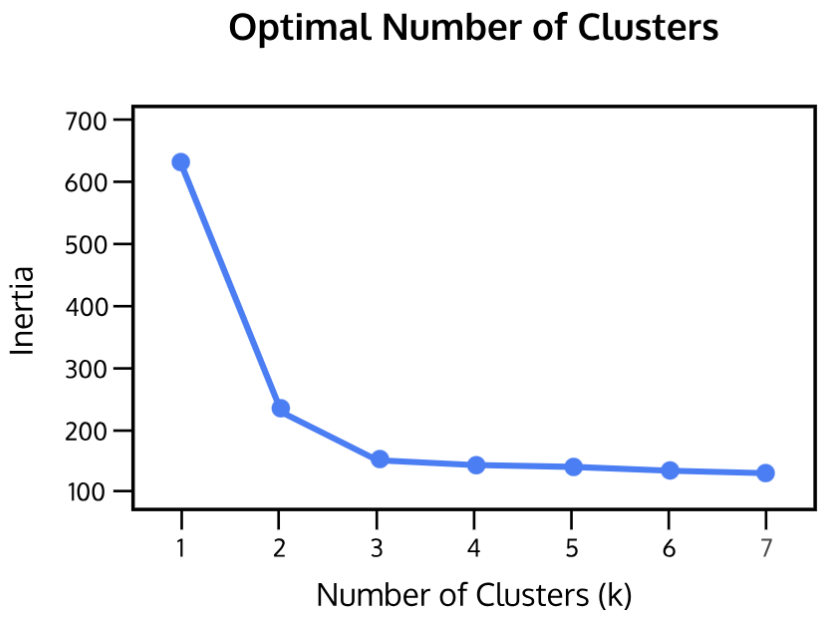


Figure 9-6. Mini-batch K-Means has a higher inertia than K-Means (left) but it is much faster (right), especially as k increases

Finding the optimal number of clusters

- Inertia can also be used to find optimal value of K.
- A good model is one with low inertia AND a low number of clusters (K).
- However, this is a tradeoff because as K increases, inertia decreases.
- To find the optimal K for a dataset, use the *Elbow method*; find the point where the decrease in inertia begins to slow. K=3 is the “elbow” of this graph.



The inertia is not a good performance metric when trying to choose *k* because it keeps getting lower as we increase *k*. Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. Let's plot the inertia as a function of *k* (see Figure 9-8).

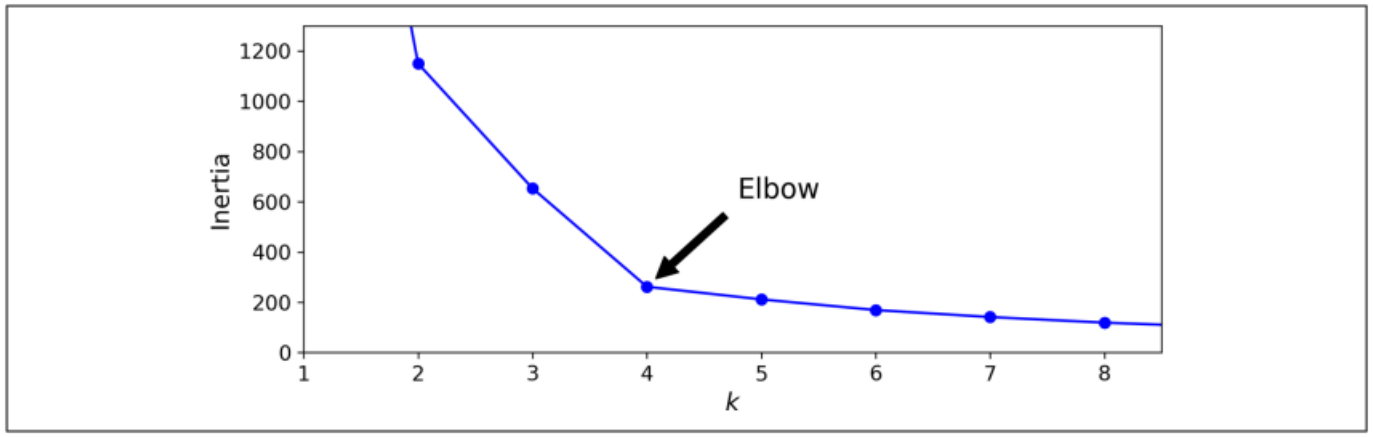


Figure 9-8. When plotting the inertia as a function of the number of clusters k, the curve often contains an inflexion point called the "elbow"

This technique for choosing the best value for the number of clusters is rather coarse.

A more precise approach (but also more computationally expensive) is to use the *silhouette score*, which is the mean *silhouette coefficient* over all the instances.

Silhouette coefficient

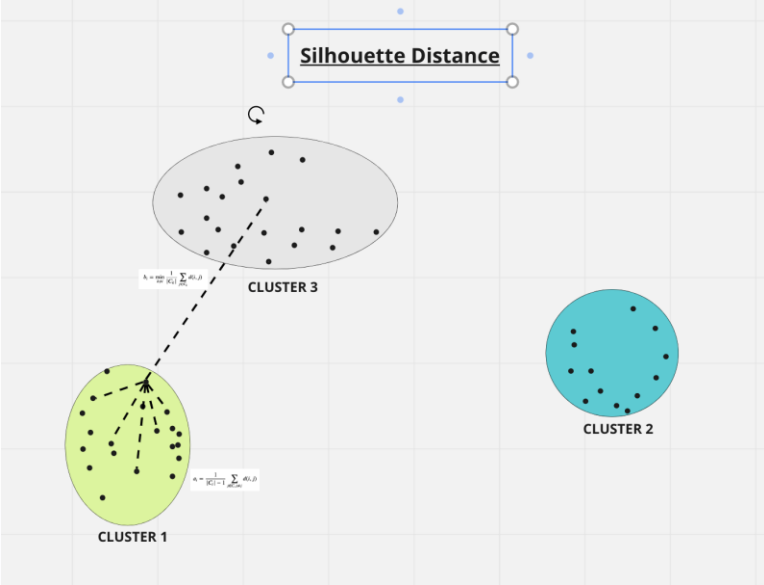
- A more precise approach (but also more computationally expensive) is to use the *silhouette score*, which is the mean *silhouette coefficient* over all the instances.
- It measures how similar a data point is to its own cluster compared to other clusters.
- For each data point, the silhouette score is calculated using the following steps:

✓ **Calculate the average distance (a) between the data point and all other points in its own cluster.**

✓ **Identify the nearest cluster to the data point's cluster and calculate the average distance (b) between the data point and all points in that nearest cluster.**

✓ **Compute the silhouette score using the formula: (b - a) / max(a, b)**

- **Score close to 1:** The data point is well-matched to its own cluster and far away from other clusters.
- **Score close to 0:** The data point lies on or near the decision boundary between two clusters.
- **Score close to -1:** The data point might have been assigned to the wrong cluster.



```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```

Let's compare the silhouette scores for different numbers of clusters (see Figure 9-9).

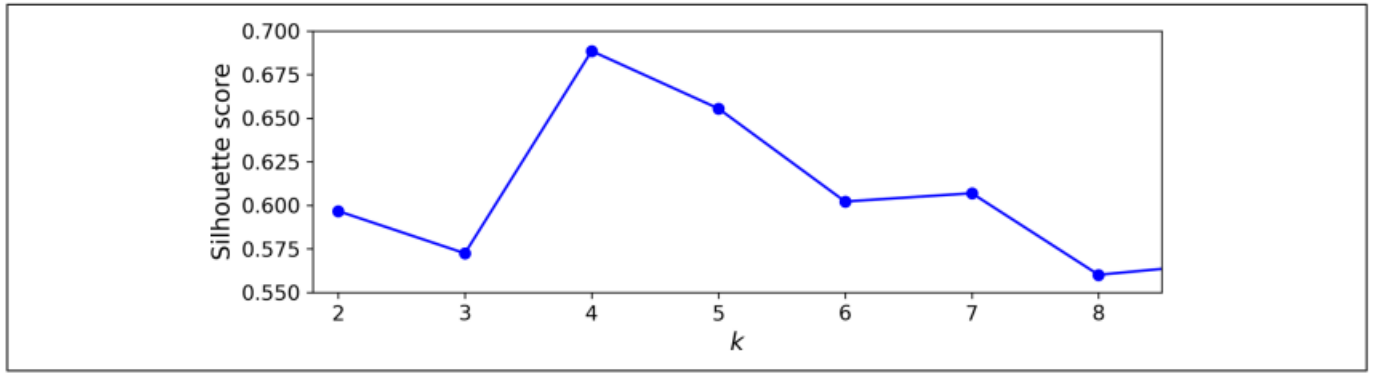


Figure 9-9. Selecting the number of clusters k using the silhouette score

- ✓ As you can see, this visualization is much richer than the previous one:
- ✓ although it confirms that $k = 4$ is a very good choice, it also underlines the fact that $k = 5$ is quite good as well, and much better than $k = 6$ or 7.
- ✓ This was not visible when comparing inertias.

Silhouette diagram

- Even more informative visualization is obtained when you plot every instance's silhouette coefficient, sorted by the cluster they are assigned to and by the value of the coefficient.
- This is called a *silhouette diagram* (see Figure 9-10).
- Each diagram contains one knife shape per cluster. The shape's height indicates the number of instances the cluster contains, and its width represents the sorted silhouette coefficients of the instances in the cluster.
- The dashed line indicates the mean silhouette coefficient.

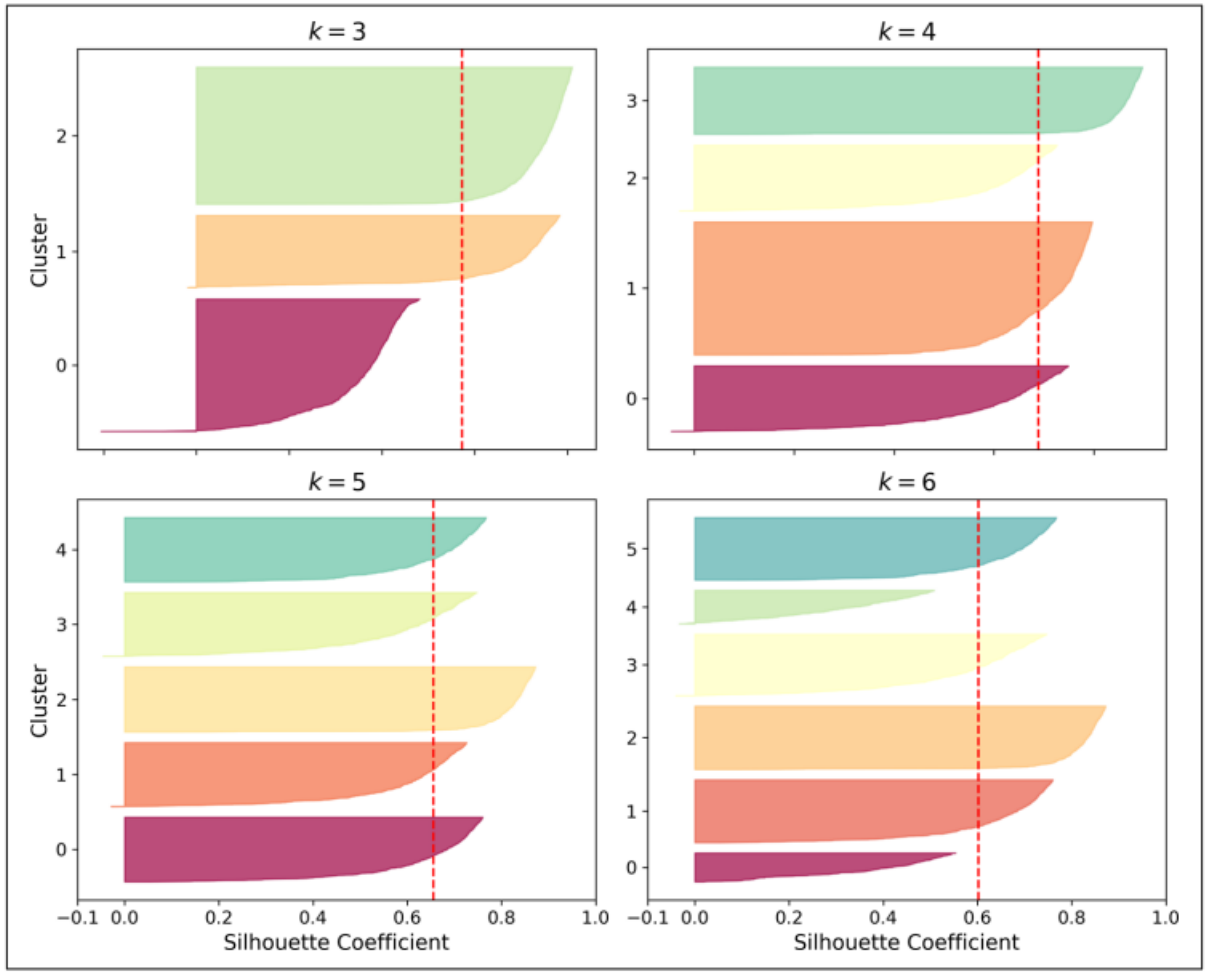


Figure 9-10. Analyzing the silhouette diagrams for various values of k

- When most of the instances in a cluster have a lower coefficient than this score (i.e., if many of the instances stop short of the dashed line, ending to the left of it), then the cluster is rather bad since this means its instances are much too close to other clusters.
- We can see that when $k = 3$ and when $k = 6$, we get bad clusters.
- But when $k = 4$ or $k = 5$, the clusters look pretty good: most instances extend beyond the dashed line, to the right and closer to 1.0.
- When $k = 4$, the cluster at index 1 (the third from the top) is rather big. When $k = 5$, all clusters have similar sizes. So, even though the overall silhouette score from $k = 4$ is slightly greater than for $k = 5$, it seems like a good idea to use $k = 5$ to get clusters of similar sizes.

Limits of K-Means

- It is necessary to run the algorithm several times to avoid suboptimal solutions,
- You need to specify the number of clusters, which can be quite a hassle.
- K-Means does not behave very well when the clusters have varying sizes, different densities, or nonspherical shapes.

For example, Figure 9-11 shows how K-Means clusters a dataset containing three ellipsoidal clusters of different dimensions, densities, and orientations.

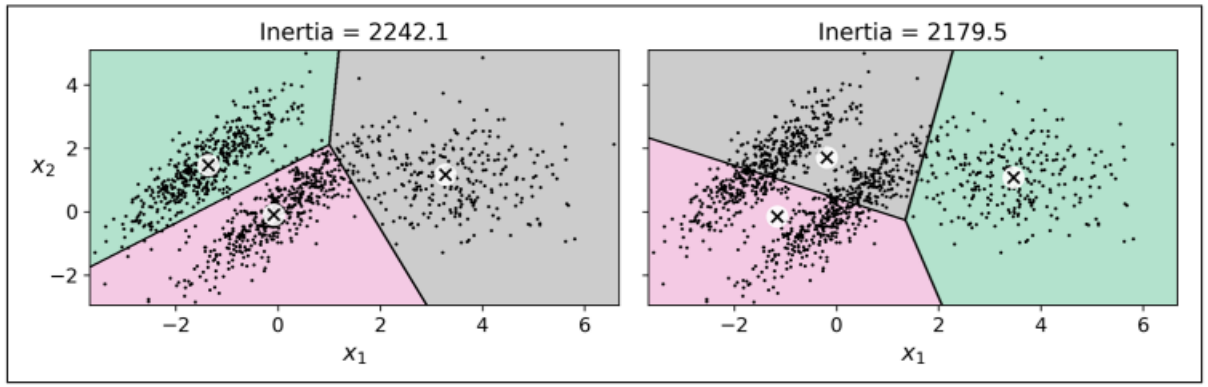


Figure 9-11. K-Means fails to cluster these ellipsoidal blobs properly

The solution on the right is just terrible, even though its inertia is lower

Using Clustering for Image Segmentation

- *Image segmentation* is the task of partitioning an image into multiple segments.
- *semantic segmentation:* all pixels that are part of the same object type get assigned to the same segment. For example, in a self-driving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the "pedestrian" segment (there would be one segment containing all the pedestrians).
- *instance segmentation:* all pixels that are part of the same individual object are assigned to the same segment. In this case there would be a different segment for each pedestrian.

- The state of the art in semantic or instance segmentation today is achieved using complex architectures based on convolutional neural networks

color segmentation. We will simply assign pixels to the same segment if they have a similar color. In some applications, this may be sufficient. For example, if you want to analyze satellite images to measure how much total forest area there is in a region, color segmentation may be just fine.

```
>>> from matplotlib.image import imread  # or `from imageio import imread`
>>> image = imread(os.path.join("images","unsupervised_learning","ladybug.png"))
>>> image.shape
(533, 800, 3)
```

```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

Figure 9-12. Image segmentation using K-Means with various numbers of color clusters

Using Clustering for Preprocessing

Clustering can be an efficient approach to dimensionality reduction, in particular as a preprocessing step before a supervised learning algorithm. As an example of using clustering for dimensionality reduction

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
```

Now, split it into a training set and a test set:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits)
```

Next, fit a Logistic Regression model:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
```

Let's evaluate its accuracy on the test set:

```
>>> log_reg.score(X_test, y_test)
0.9688888888888889
```

that's our baseline: 96.9% accuracy. Let's see if we can do better by using K-Means as a preprocessing step.

We will create a pipeline that will first cluster the training set into 50 clusters and replace the images with their distances to these 50 clusters, then apply a Logistic Regression model:

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression()),
])
pipeline.fit(X_train, y_train)
```

Now let's evaluate this classification pipeline:

```
>>> pipeline.score(X_test, y_test)
0.9777777777777777
```

But we chose the number of clusters k arbitrarily; we can surely do better. Since K-Means is just a preprocessing step in a classification pipeline, finding a good value for k is much simpler than earlier. There's no need to perform silhouette analysis or minimize the inertia; the best value of k is simply the one that results in the best classification performance during cross-validation. We can use `GridSearchCV` to find the optimal number of clusters:

```
from sklearn.model_selection import GridSearchCV

param_grid = dict(kmeans__n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)
```

Let's look at the best value for k and the performance of the resulting pipeline:

```
>>> grid_clf.best_params_
{'kmeans__n_clusters': 99}
>>> grid_clf.score(X_test, y_test)
0.9822222222222222
```

With $k = 99$ clusters, we get a significant accuracy boost, reaching 98.22% accuracy on the test set. Cool! You may want to keep exploring higher values for k, since 99 was the largest value in the range we explored.

Using Clustering for Semi-Supervised Learning

Another use case for clustering is in semi-supervised learning, when we have plenty of unlabeled instances and very few labeled instances. Let's train a Logistic Regression model on a sample of 50 labeled instances from the digits dataset:

```
n_labeled = 50
log_reg = LogisticRegression()
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

```
>>> log_reg.score(X_test, y_test)
0.8333333333333334
```

First, let's cluster the training set into 50 clusters. Then for each cluster, let's find the image closest to the centroid. We will call these images the *representative images*:

```
k = 50
kmeans = KMeans(n_clusters=k)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

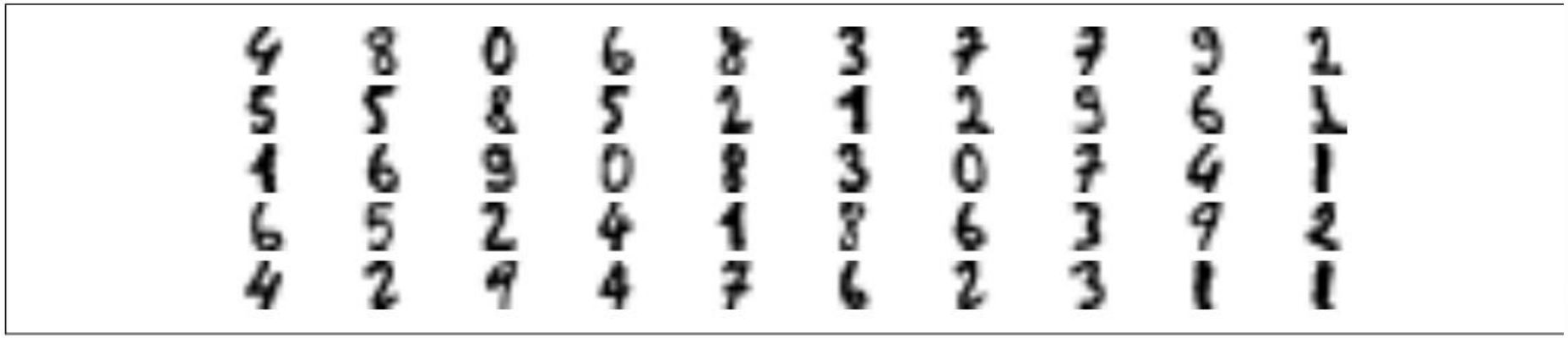


Figure 9-13. Fifty representative digit images (one per cluster)

Let's look at each image and manually label it:

```
y_representative_digits = np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])
```

Now we have a dataset with just 50 labeled instances, but instead of being random instances, each of them is a representative image of its cluster. Let's see if the performance is any better:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.9222222222222223
```

But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster? This is called *label propagation*:

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]
```

Now let's train the model again and look at its performance:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
>>> log_reg.score(X_test, y_test)
0.9333333333333333
```

We got a reasonable accuracy boost, but nothing absolutely astounding. The problem is that we propagated each representative instance's label to all the instances in the same cluster, including the instances located close to the cluster boundaries, which are more likely to be mislabeled. Let's see what happens if we only propagate the labels to the 20% of the instances that are closest to the centroids:

- Nice! With just 50 labeled instances (only 5 examples per class on average!), we got 94.0% accuracy, which is pretty close to the performance of Logistic Regression on the fully labeled digits dataset (which was 96.9%).
- This good performance is due to the fact that the propagated labels are actually pretty good

DBSCAN:

Density-based spatial clustering of applications with noise

another popular clustering algorithm that illustrates a very different approach based on local density estimation. This approach allows the algorithm to identify clusters of arbitrary shapes.

Here is how it works:

- For each instance, the algorithm counts how many instances are located within a small distance ε (epsilon) from it. This region is called the instance's *ε-neighborhood*.
- If an instance has at least `min_samples` instances in its ε-neighborhood (including itself), then it is considered a *core instance*. In other words, core instances are those that are located in dense regions.
- All instances in the neighborhood of a core instance belong to the same cluster. This neighborhood may include other core instances; therefore, a long sequence of neighboring core instances forms a single cluster.
- Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

The labels of all the instances are now available in the `labels_` instance variable:

```
>>> dbscan.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0, ...,  3,  2,  3,  3,  4,  2,  6,  3])
```

Notice that some instances have a cluster index equal to –1, which means that they are considered as anomalies by the algorithm. The indices of the core instances are available in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

```
>>> len(dbscan.core_sample_indices_)
808
>>> dbscan.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])
>>> dbscan.components_
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
                  ...
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```

If we widen each instance's neighborhood by increasing `eps` to 0.2, we get the clustering on the right, which looks perfect

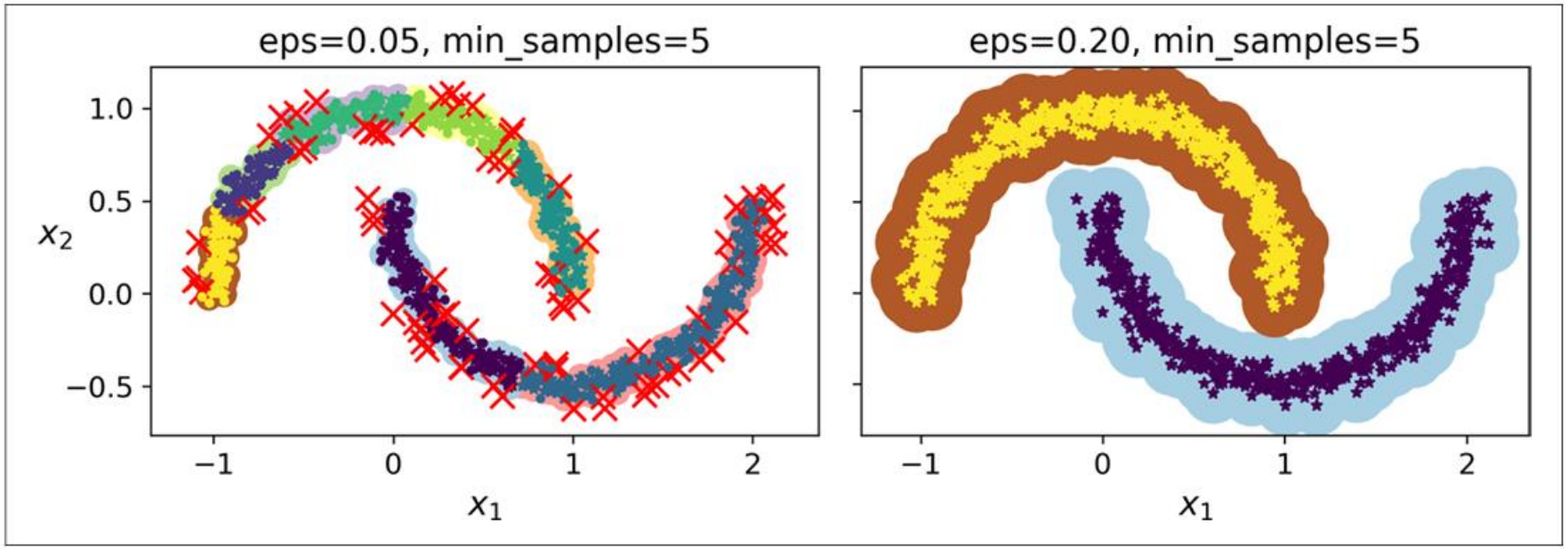


Figure 9-14. DBSCAN clustering using two different neighborhood radiuses

the DBSCAN class does not have a predict() method, although it has a fit_predict() method. In other words, it cannot predict which cluster a new instance belongs to. This implementation decision was made because different classification algorithms can be better for different tasks, so the authors decided to let the user choose which one to use. Moreover, it's not hard to implement. For example, let's train a KNeighborsClassifier:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```

Now, given a few new instances, we can predict which cluster they most likely belong to and even estimate a probability for each cluster:

```
>>> X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
>>> knn.predict(X_new)
array([1, 0, 1, 0])
>>> knn.predict_proba(X_new)
array([[0.18, 0.82],
       [1.  , 0.  ],
       [0.12, 0.88],
       [1.  , 0.  ]])
```

Note that we only trained the classifier on the core instances, but we could also have chosen to train it on all the instances, or all but the anomalies: this choice depends on the final task.

- The decision boundary is represented in Figure 9-15 (the crosses represent the four instances in `X_new`).
- Notice that since there is no anomaly in the training set, the classifier always chooses a cluster, even when that cluster is far away.
- It is fairly straight-forward to introduce a maximum distance, in which case the two instances that are far away from both clusters are classified as anomalies. To do this, use the `kneighbors()` method of the `KNeighborsClassifier`. Given a set of instances, it returns the distances and the indices of the *k* nearest neighbors in the training set (two matrices, each with *k* columns):

```
>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
>>> y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
>>> y_pred[y_dist > 0.2] = -1
>>> y_pred.ravel()
array([-1,  0,  1, -1])
```

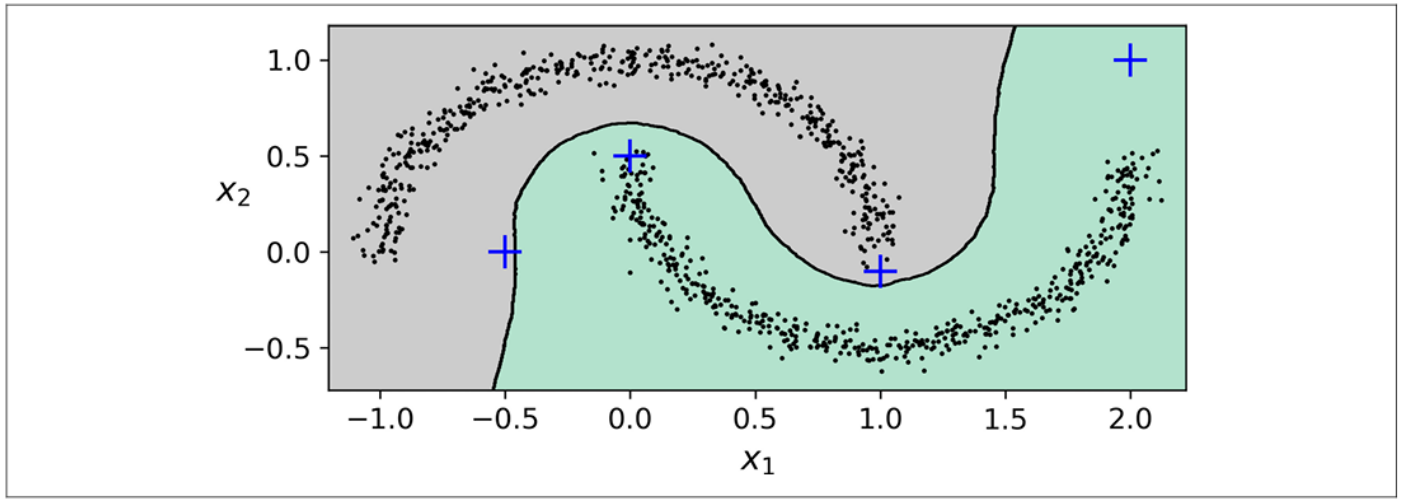


Figure 9-15. Decision boundary between two clusters

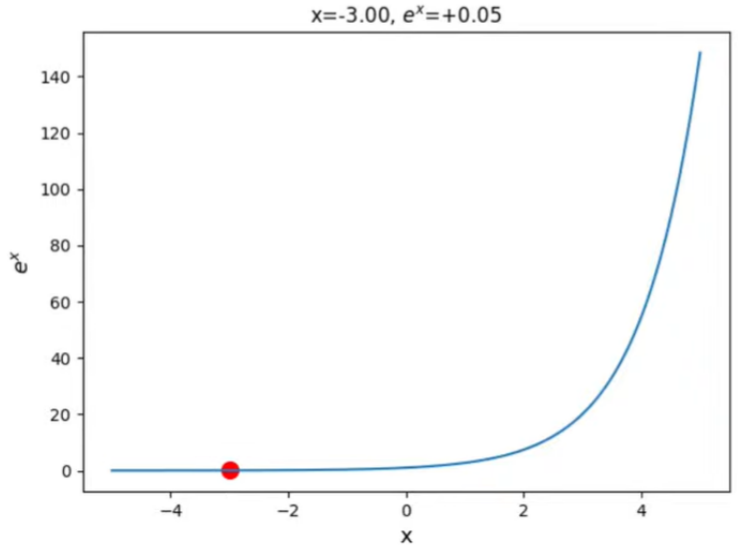
Gaussian Mixtures

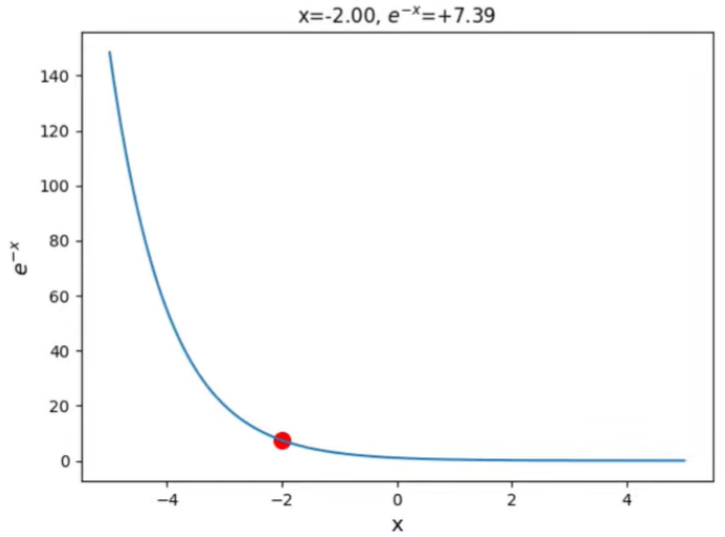
- A *Gaussian mixture model* (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown.
- Each cluster can have a different ellipsoidal shape, size, density, and orientation. When you observe an instance, you know it was generated from one of the Gaussian distributions, but you are not told which one, and you do not know what the parameters of these distributions are.

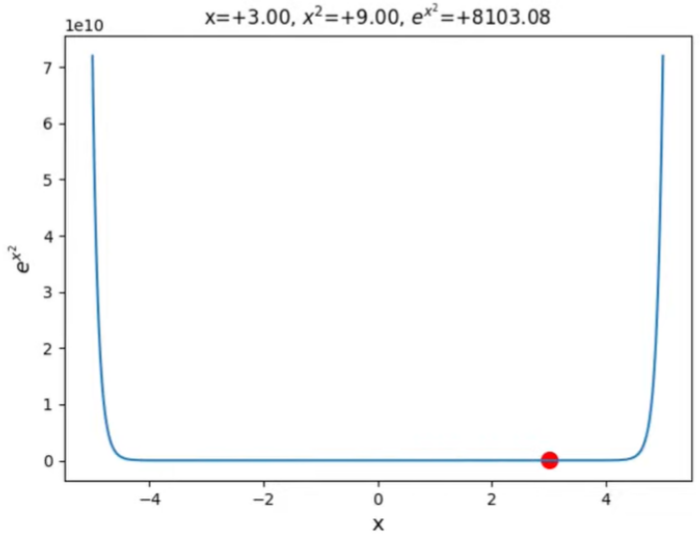
There are several GMM variants. In the simplest variant, implemented in the `GaussianMixture` class, you must know in advance the number k of Gaussian distributions. The dataset $\mathbf{X}$ is assumed to have been generated through the following probabilistic process:

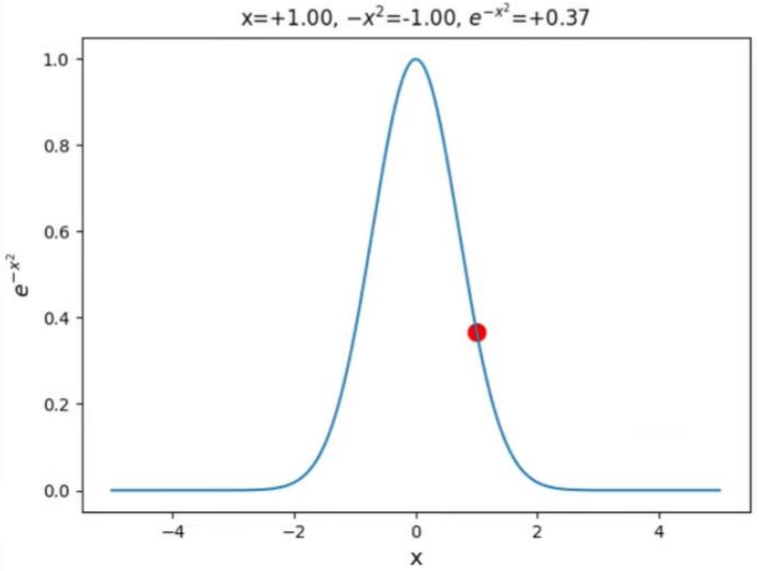
- For each instance, a cluster is picked randomly from among k clusters. The probability of choosing the j^{th} cluster is defined by the cluster's weight, $\phi^{(j)}$.[7] The index of the cluster chosen for the i^{th} instance is noted $z^{(i)}$.
- If $z^{(i)}=j$, meaning the i^{th} instance has been assigned to the j^{th} cluster, the location $\mathbf{x}^{(i)}$ of this instance is sampled randomly from the Gaussian distribution with mean $\boldsymbol{\mu}^{(j)}$ and covariance matrix $\boldsymbol{\Sigma}^{(j)}$. This is noted $\mathbf{x}^{(i)} \sim \mathcal{N}\left(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)}\right)$.

Univariate Normal (Gaussian) Distribution

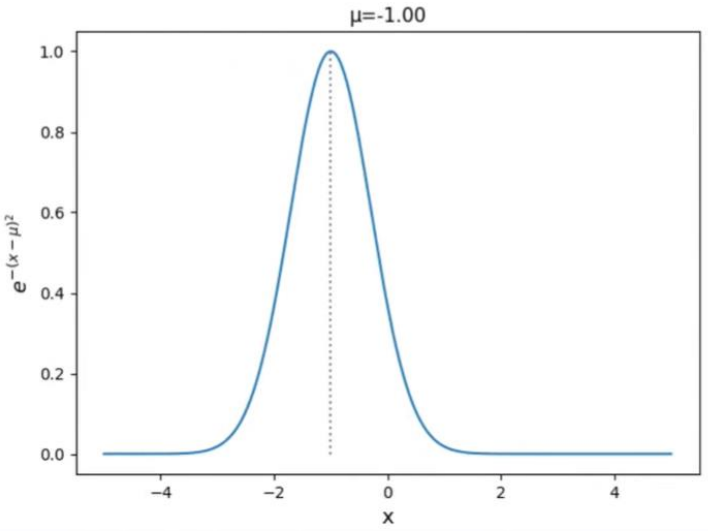




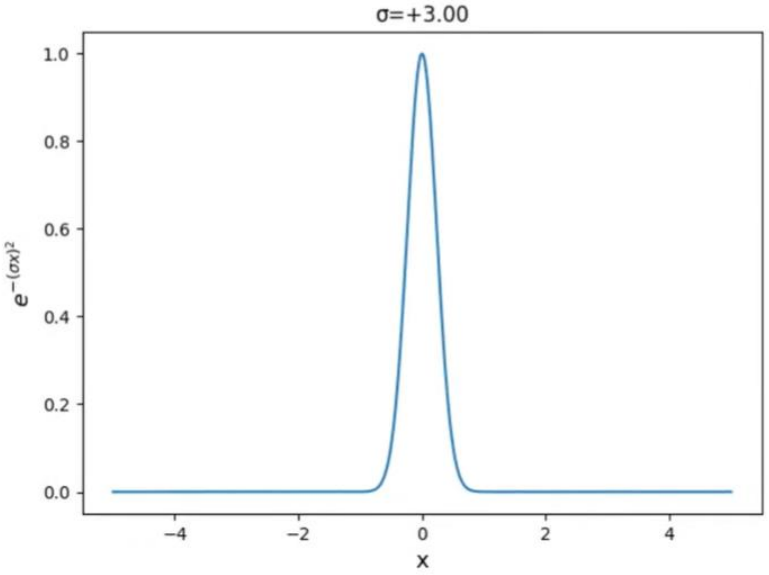




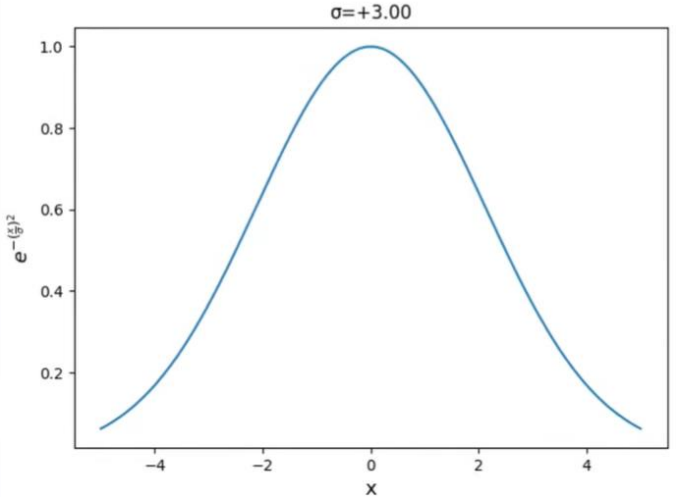
$$e^{-(x-\mu)^2}$$



$$e^{-(\sigma x)^2}$$

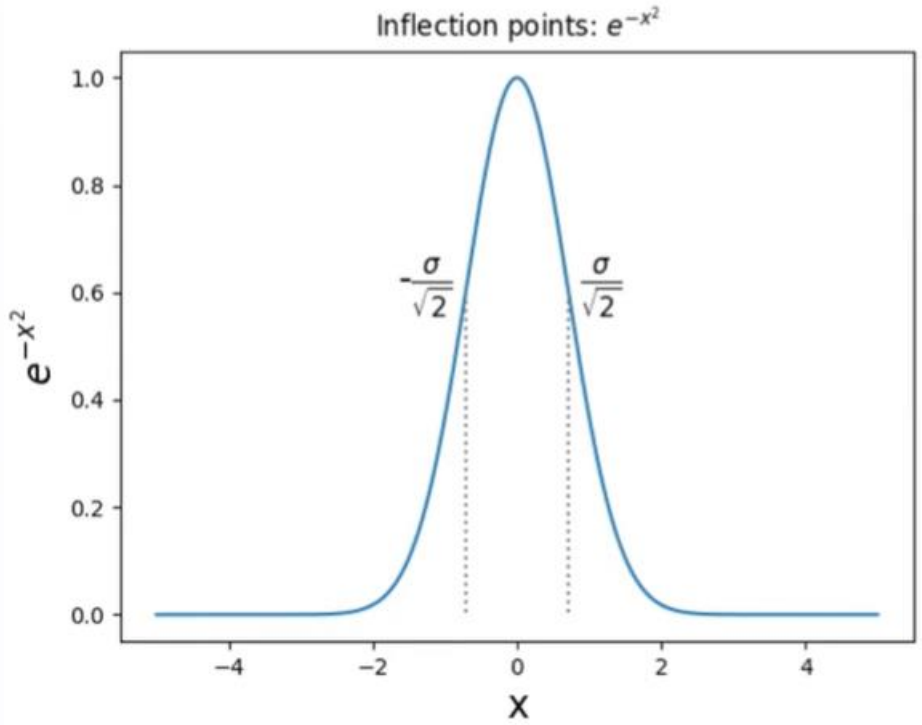


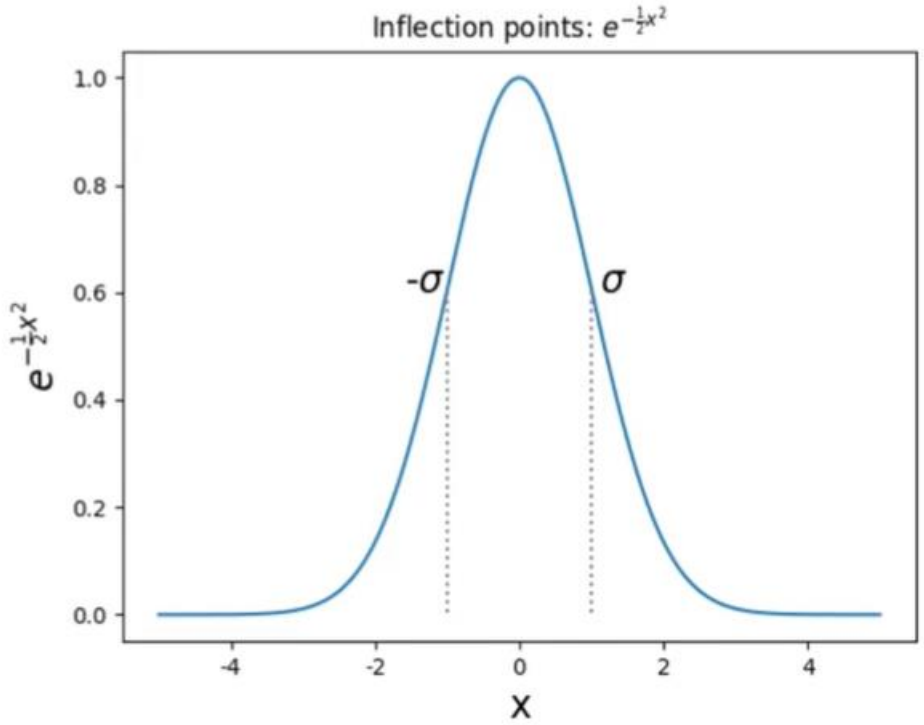
$$e^{-(\frac{x}{\sigma})^2}$$



$$e^{-\left(\frac{x-\mu}{\sigma}\right)^2}$$

$$e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

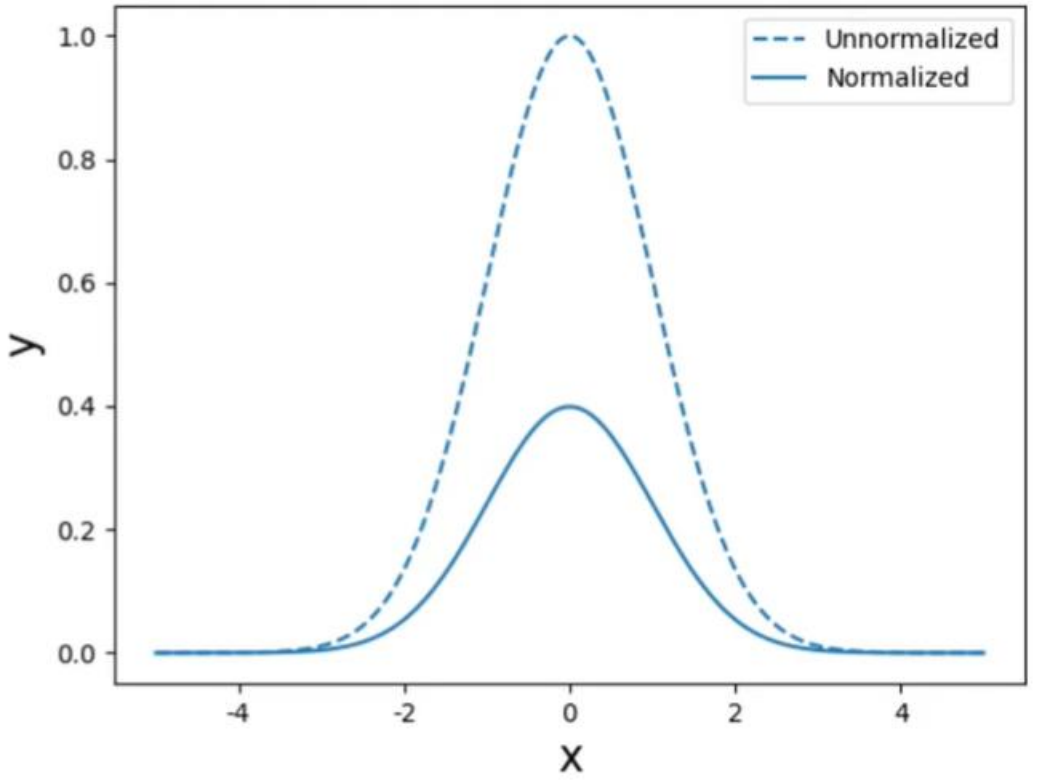




Normalized the gaussian to make it a probability distribution

$$\int_{-\inf}^{+\inf} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} = \sigma\sqrt{2\pi}$$

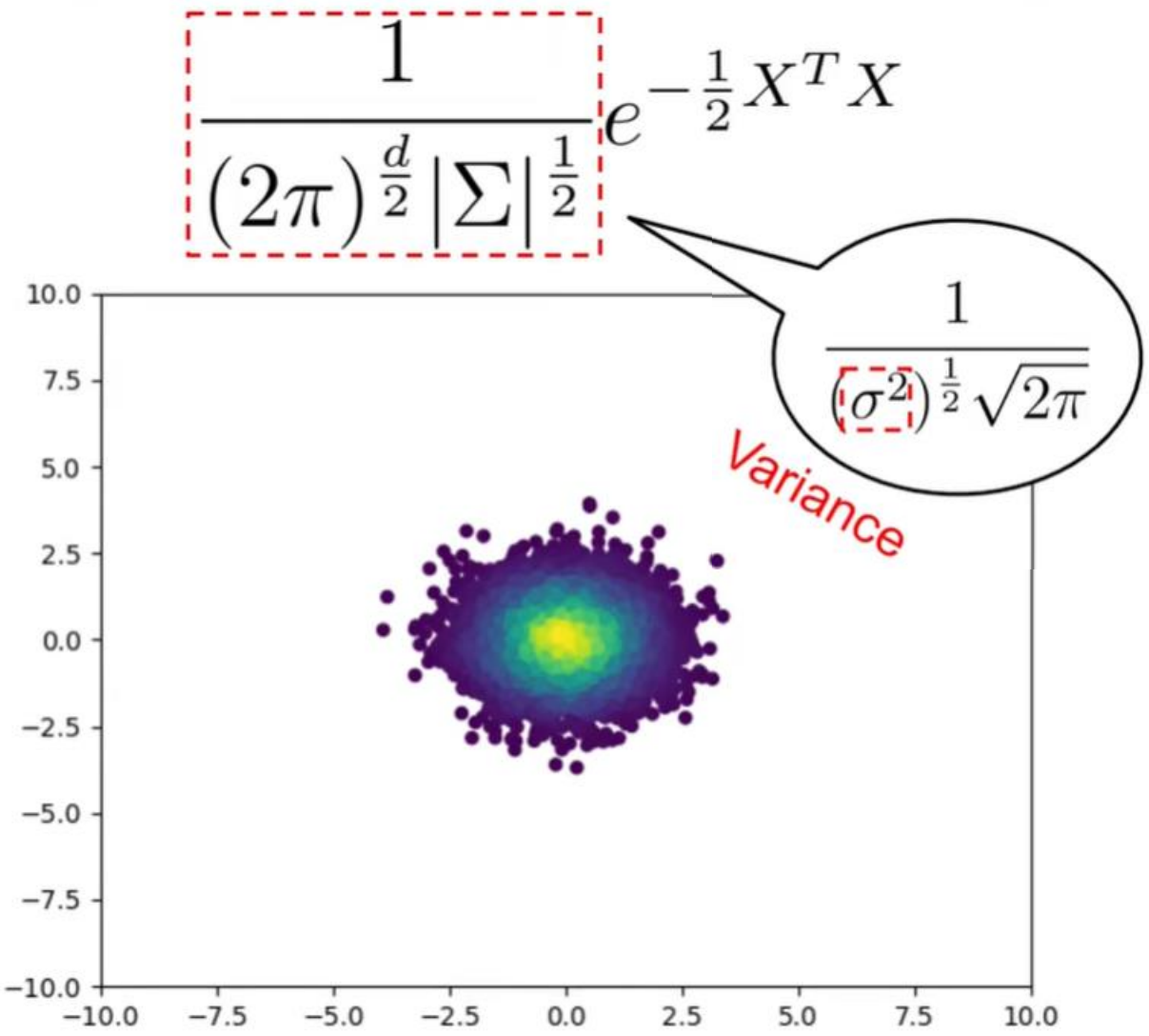
$$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$



Normal (gaussian) distribution

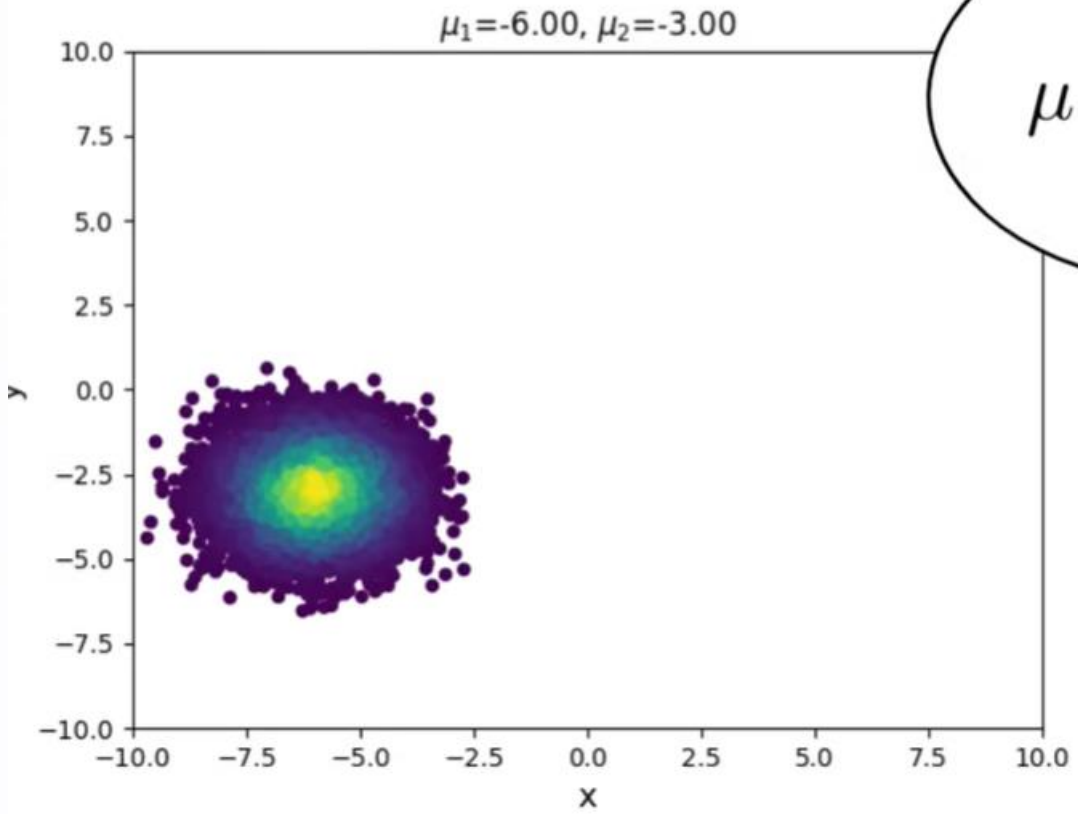
Multivariate Normal (Gaussian) Distribution

It is just a generalization of the 1D normal distribution



$$\frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma|^{\frac{1}{2}}}e^{-\frac{1}{2}(X-\mu)^T(X-\mu)}$$

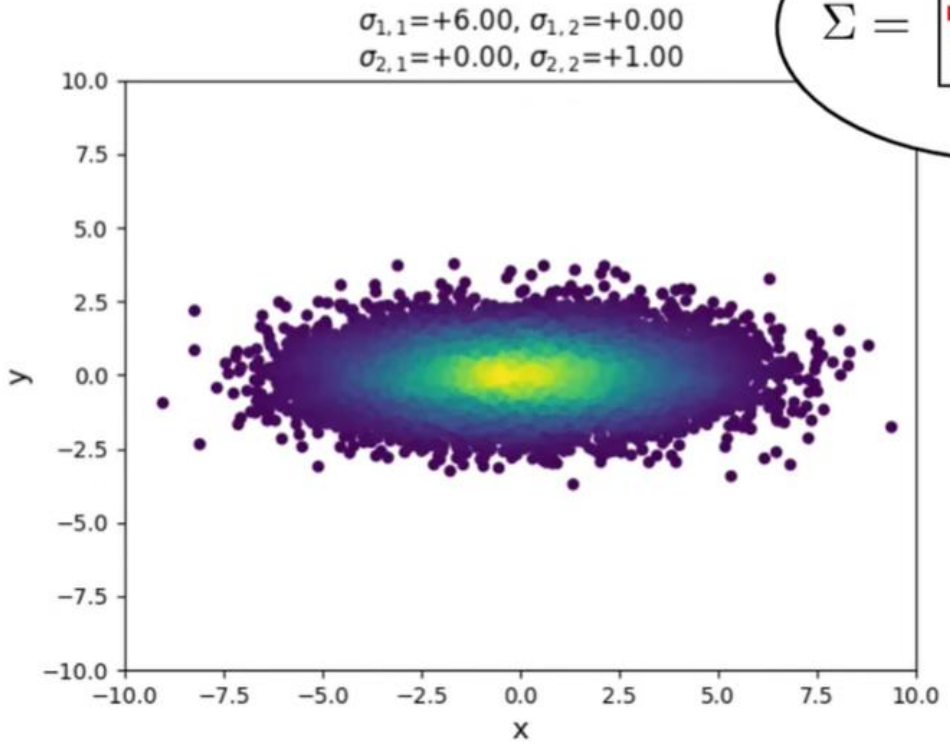
$$\mu = \begin{bmatrix}\mu_1 \\ \mu_2\end{bmatrix}$$



Impact of Covariance Matrix

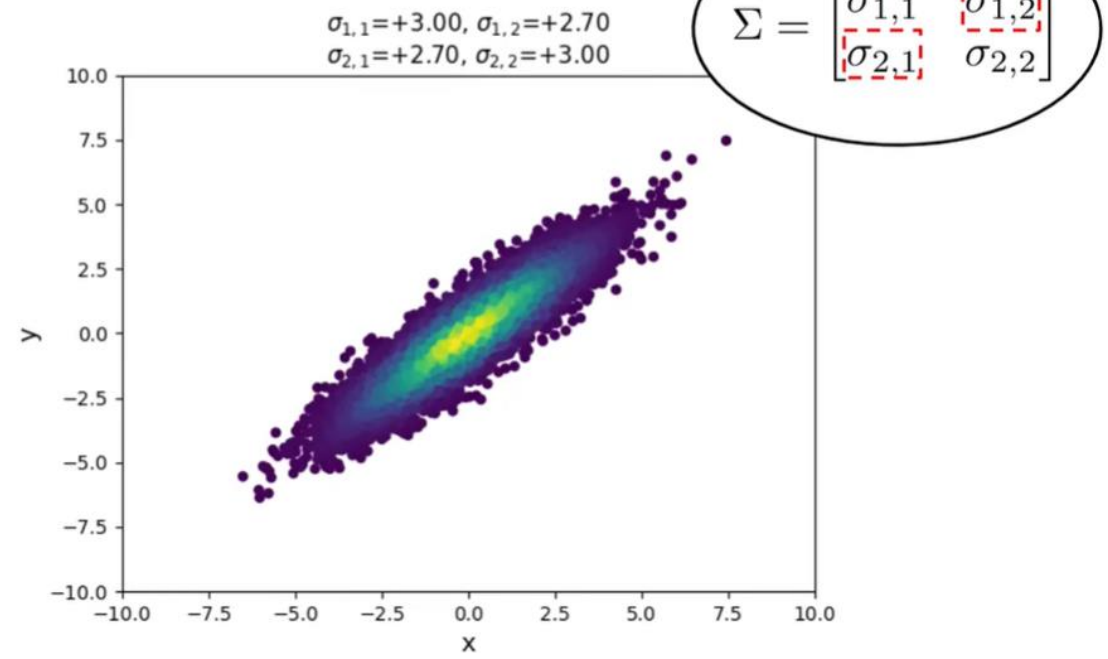
$$\frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma|^{\frac{1}{2}}}e^{-\frac{1}{2}(X-\mu)^T\Sigma^{-1}(X-\mu)}$$

$$\Sigma = \begin{bmatrix} \sigma_{1,1} & \sigma_{1,2} \\ \sigma_{2,1} & \sigma_{2,2} \end{bmatrix}$$



$$\frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma|^{\frac{1}{2}}}e^{-\frac{1}{2}(X-\mu)^T\Sigma^{-1}(X-\mu)}$$

$$\Sigma = \begin{bmatrix} \sigma_{1,1} & \sigma_{1,2} \\ \sigma_{2,1} & \sigma_{2,2} \end{bmatrix}$$



Covariance vs Correlation

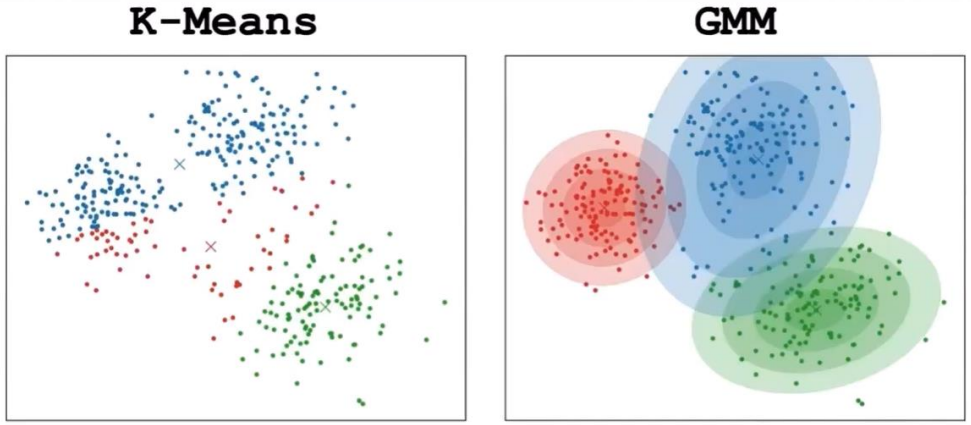
$$\mathrm{cov}(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - E(X))(y_i - E(Y))$$

$$\mathrm{corr}(X,Y) = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y}$$

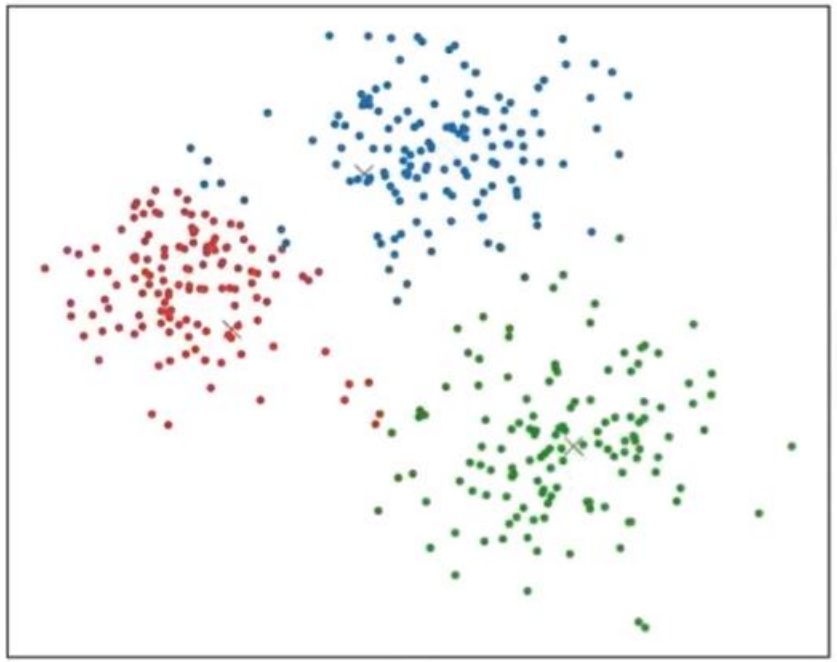
Correlation = normalised covariance

- GMM introduced a probabilistic approach by modelling the data points as a mixture of multiple Gaussian distributions

- It provides softer clustering. It tells how probably its for a data point to belong to a certain cluster.

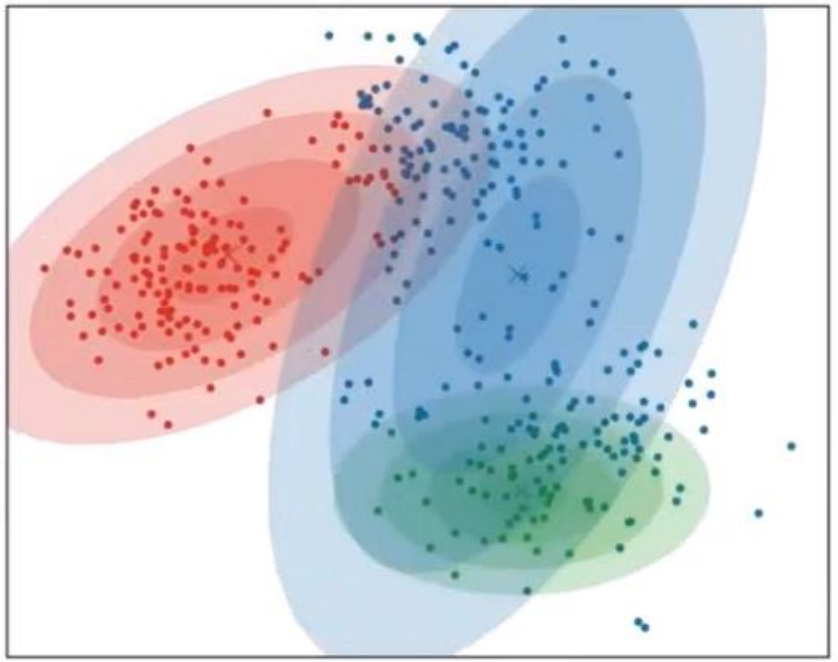


K-Means



red, blue or green

GMM



60% red
30 % blue
10% green

Expectation-Maximization for training the GMM unknown parameters

The Expectation-Maximization (EM) algorithm is a powerful iterative optimization technique used for estimating parameters in statistical models when there is missing or incomplete data such as the parameters of our gaussians in the GMM model

- **EXPECTATION STEP (E-step)**: calculate the expected value of the log-likelihood function given the current parameter estimates.

- **MAXIMIZATION STEP (M-step)**: update the parameter estimates to maximize the expected log-likelihood calculated in the E-step

- *Expectation- Maximization* (EM) algorithm, which has many similarities with the K-Means algorithm:

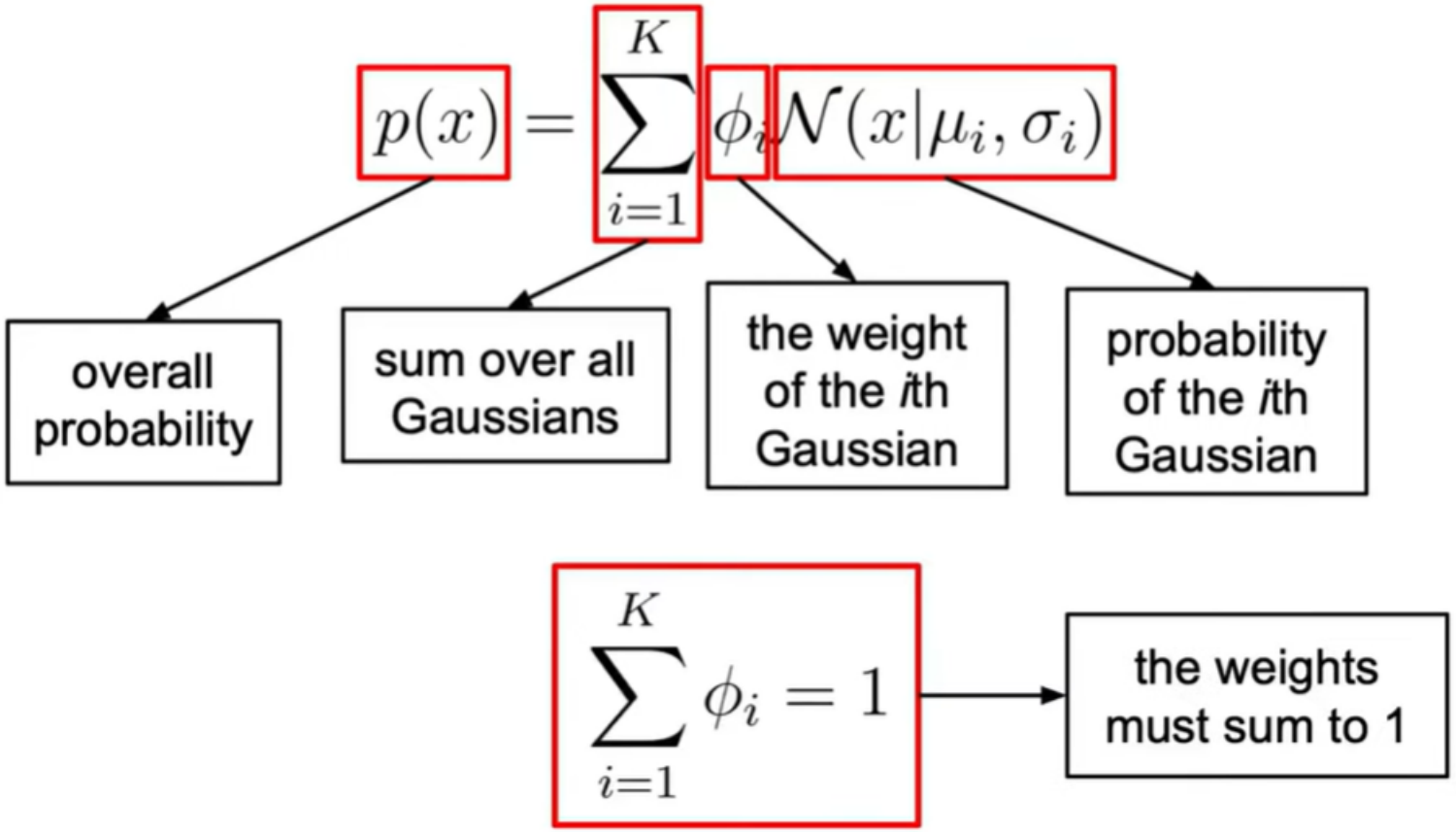
- it also initializes the cluster parameters randomly, then it repeats two steps until convergence, first assigning instances to clusters (this is called the *expectation step*) and then updating the clusters (this is called the *maximization step*).

- In the context of clustering, you can think of EM as a generalization of K-Means that not only finds the cluster centers ($\boldsymbol{\mu}^{(1)}$ to $\boldsymbol{\mu}^{(k)}$), but also their size, shape, and orientation ($\boldsymbol{\Sigma}^{(1)}$ to $\boldsymbol{\Sigma}^{(k)}$), as well as their relative weights ($\phi^{(1)}$ to $\phi^{(k)}$).

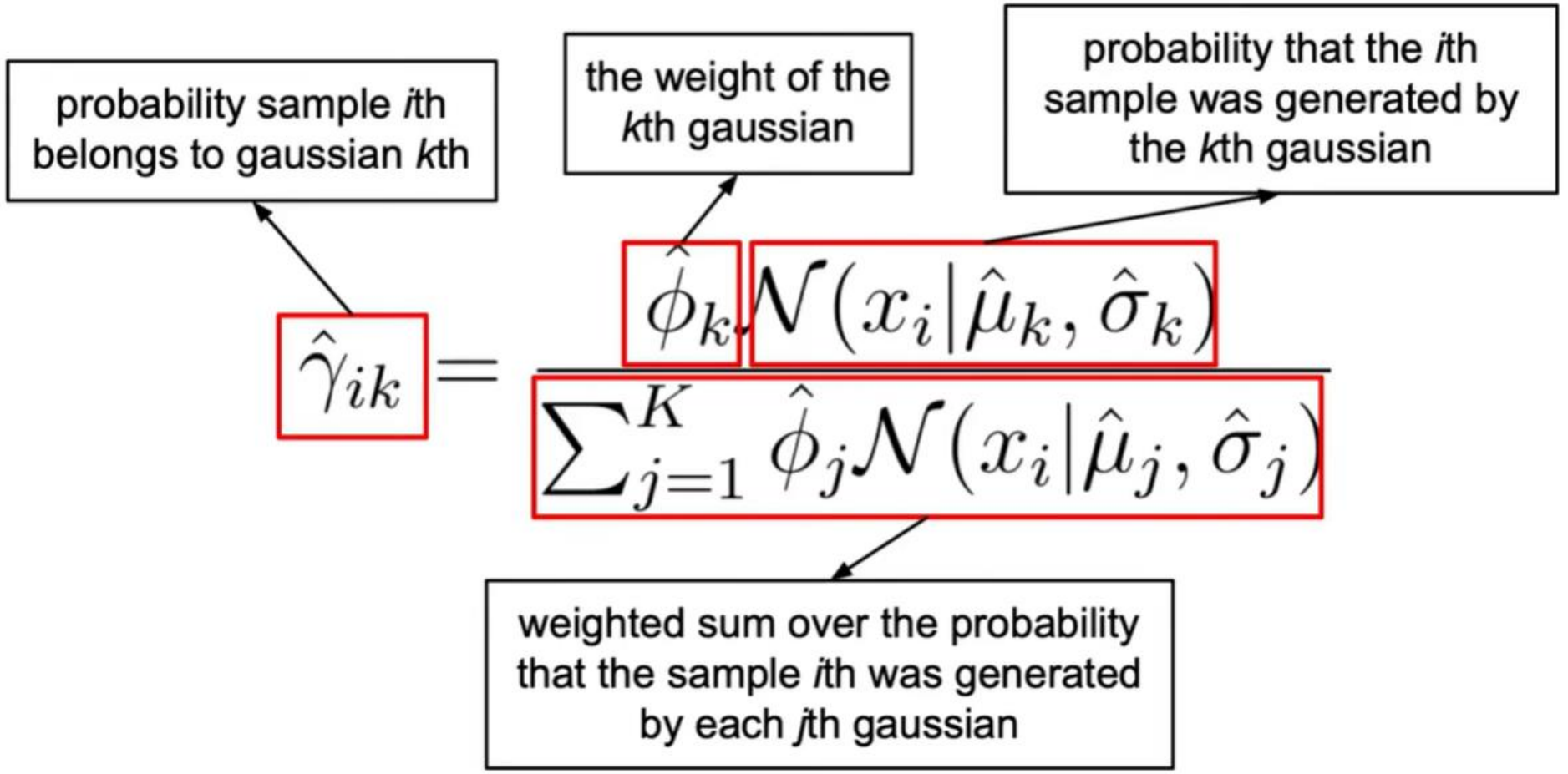
- Unlike K- Means, though, EM uses soft cluster assignments, not hard assignments. For each instance, during the expectation step, the algorithm estimates the probability that it belongs to each cluster (based on the current cluster parameters).

- Then, during the maximization step, each cluster is updated using *all* the instances in the dataset, with each instance weighted by the estimated probability that it belongs to that cluster.

GMM Probability



Expectation Step



Maximization Step

$$\hat{\phi}_k = \sum_{i=1}^{N} \frac{\hat{\gamma}_{ik}}{N}$$

The new *k*th weight becomes the average of probabilities that a point belongs to that gaussian.

$$\hat{\mu}_k = \frac{\sum_{i=1}^{N} \hat{\gamma}_{ik} x_i}{\sum_{i=1}^{N} \hat{\gamma}_{ik}}$$

The new *k*th mean becomes the weighted average of all points.

$$\hat{\sigma}_k^2 = \frac{\sum_{i=1}^{N} \hat{\gamma}_{ik} (x_i - \hat{\mu}_k)^2}{\sum_{i=1}^{N} \hat{\gamma}_{ik}}$$

The new *k*th variance becomes the weighted variance of all points.

given the dataset **X**, you typically want to start by estimating the weights ϕ and all the distribution parameters $\mu^{(1)}$ to $\mu^{(k)}$ and $\Sigma^{(1)}$ to $\Sigma^{(k)}$. Scikit-Learn's `GaussianMixture` class makes this super easy

```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
```

Let's look at the parameters that the algorithm estimated:

```
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
>>> gm.means_
array([[ 3.39909717,  1.05933727],
       [-1.40763984,  1.42710194],
       [ 0.05135313,  0.07524095]])
>>> gm.covariances_
array([[[ 1.14807234, -0.03270354],
        [-0.03270354,  0.95496237]],

       [[ 0.63478101,  0.72969804],
        [ 0.72969804,  1.1609872 ]],

       [[ 0.68809572,  0.79608475],
        [ 0.79608475,  1.21234145]]])
```

Unfortunately, just like K-Means, EM can end up converging to poor solutions, so it needs to be run several times, keeping only the best solution. This is why we set `n_init` to `10`. Be careful: by default `n_init` is set to `1`.

You can check whether or not the algorithm converged and how many iterations it took:

```
>>> gm.converged_
True
>>> gm.n_iter_
3
```

Now that you have an estimate of the location, size, shape, orientation, and relative weight of each cluster, the model can easily assign each instance to the most likely cluster (hard clustering) or estimate the probability that it belongs to a particular cluster (soft clustering). Just use the `predict()` method for hard clustering, or the `predict_proba()` method for soft clustering:

```
>>> gm.predict(X)
array([2, 2, 1, ..., 0, 0, 0])
>>> gm.predict_proba(X)
array([[2.32389467e-02, 6.77397850e-07, 9.76760376e-01],
       [1.64685609e-02, 6.75361303e-04, 9.82856078e-01],
       [2.01535333e-06, 9.99923053e-01, 7.49319577e-05],
       ...,
       [9.99999571e-01, 2.13946075e-26, 4.28788333e-07],
       [1.00000000e+00, 1.46454409e-41, 5.12459171e-16],
       [1.00000000e+00, 8.02006365e-41, 2.27626238e-15]])
```

A Gaussian mixture model is a *generative model*, meaning you can sample new instances from it (note that they are ordered by cluster index):

```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([[ 2.95400315,  2.63680992],
       [-1.16654575,  1.62792705],
       [-1.39477712, -1.48511338],
       [ 0.27221525,  0.690366  ],
       [ 0.54095936,  0.48591934],
       [ 0.38064009, -0.56240465]])

>>> y_new
array([0, 1, 2, 2, 2, 2])
```

It is also possible to estimate the density of the model at any given location. This is achieved using the `score_samples()` method: for each instance it is given, this method estimates the log of the *probability density function* (PDF) at that location. The greater the score, the higher the density

```
>>> gm.score_samples(X)
array([-2.60782346, -3.57106041, -3.33003479, ..., -3.51352783,
       -4.39802535, -3.80743859])
```

It is also possible to estimate the density of the model at any given location. This is achieved using the `score_samples()` method: for each instance it is given, this method estimates the log of the *probability density function* (PDF) at that location. The greater the score, the higher the density

```
>>> gm.score_samples(X)
array([-2.60782346, -3.57106041, -3.33003479, ..., -3.51352783,
       -4.39802535, -3.80743859])
```

- If you compute the exponential of these scores, you get the value of the PDF at the location of the given instances. These are not probabilities, but probability *densities*: they can take on any positive value, not just a value between 0 and 1.
- To estimate the probability that an instance will fall within a particular region, you would have to integrate the PDF over that region (if you do so over the entire space of possible instance locations, the result will be 1).

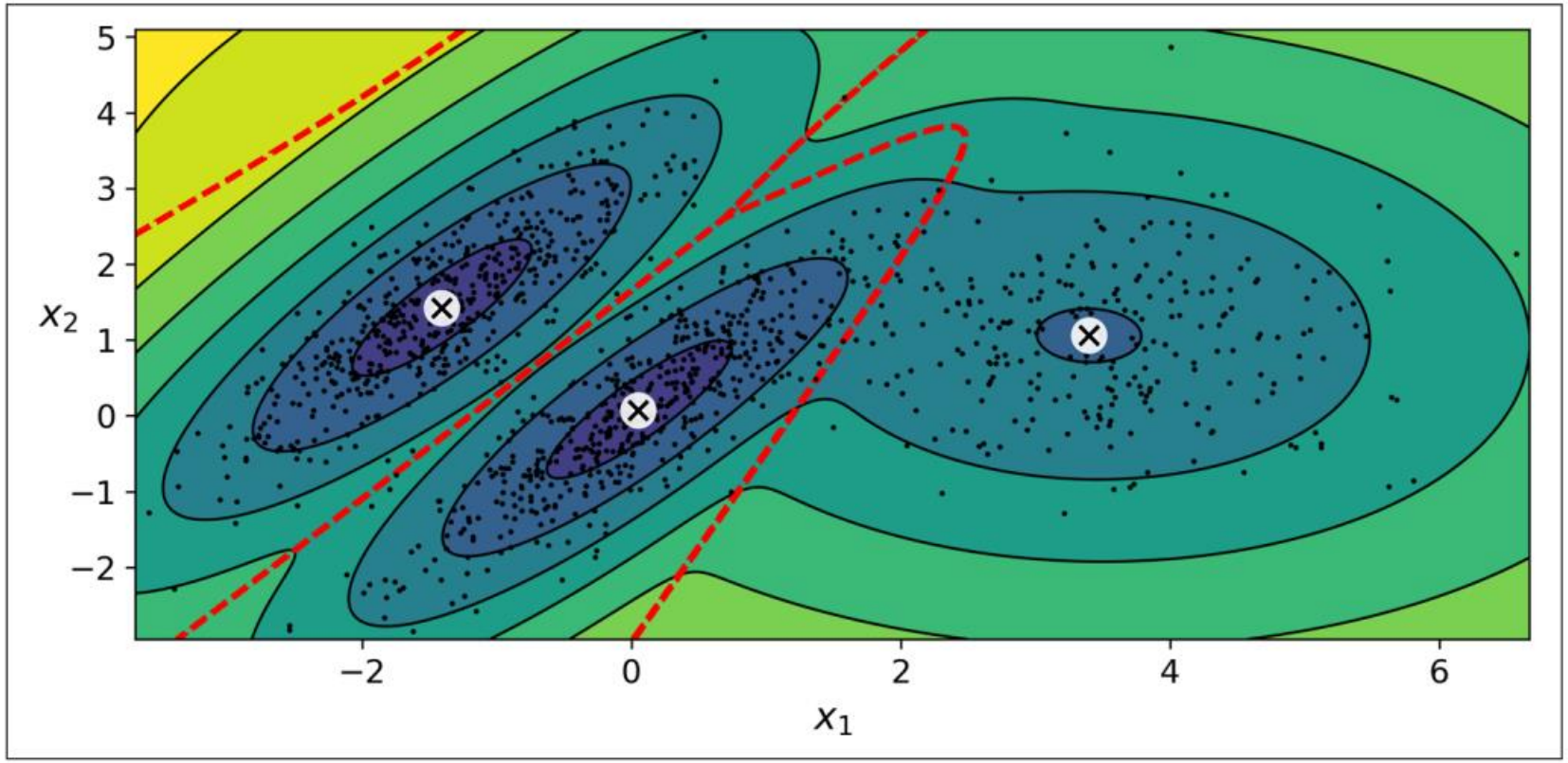


Figure 9-17. Cluster means, decision boundaries, and density contours of a trained Gaussian mixture model

When there are many dimensions, or many clusters, or few instances, EM can struggle to converge to the optimal solution.

You might need to reduce the difficulty of the task by limiting the number of parameters that the algorithm has to learn. One way to do this is to limit the range of shapes and orientations that the clusters can have. This can be achieved by imposing constraints on the covariance matrices. To do this, set the `covariance_type` hyperparameter to one of the following values:

`"spherical"`
All clusters must be spherical, but they can have different diameters (i.e., different variances).

`"diag"`
Clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the coordinate axes (i.e., the covariance matrices must be diagonal).

`"tied"`
All clusters must have the same ellipsoidal shape, size, and orientation (i.e., all clusters share the same covariance matrix).

By default, `covariance_type` is equal to `"full"`, which means that each cluster can take on any shape, size, and orientation (it has its own unconstrained covariance matrix). Figure 9-18 plots the solutions found by the EM algorithm when `covariance_type` is set to `"tied"` or "`spherical`."

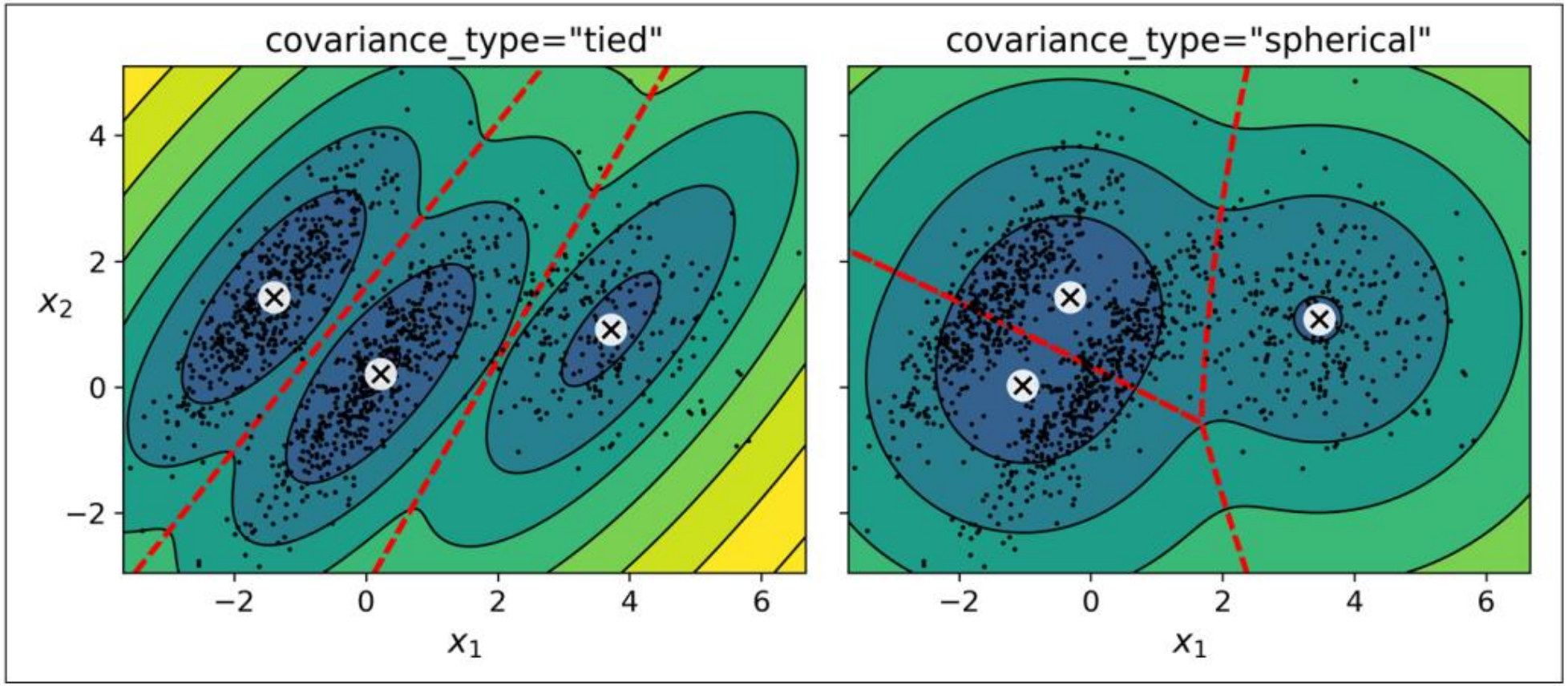


Figure 9-18. Gaussian mixtures for tied clusters (left) and spherical clusters (right)

Anomaly Detection Using Gaussian Mixtures

- Using a Gaussian mixture model for anomaly detection is quite simple: any instance located in a low-density region can be considered an anomaly.
- You must define what density threshold you want to use. For example, in a manufacturing company that tries to detect defective products, the ratio of defective products is usually well known. Say it is equal to 4%.
- You then set the density threshold to be the value that results in having 4% of the instances located in areas below that threshold density. If you notice that you get too many false positives (i.e., perfectly good products that are flagged as defective), you can lower the threshold. Conversely, if you have too many false negatives (i.e., defective products that the system does not flag as defective), you can increase the threshold.

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

Figure 9-19 represents these anomalies as stars.

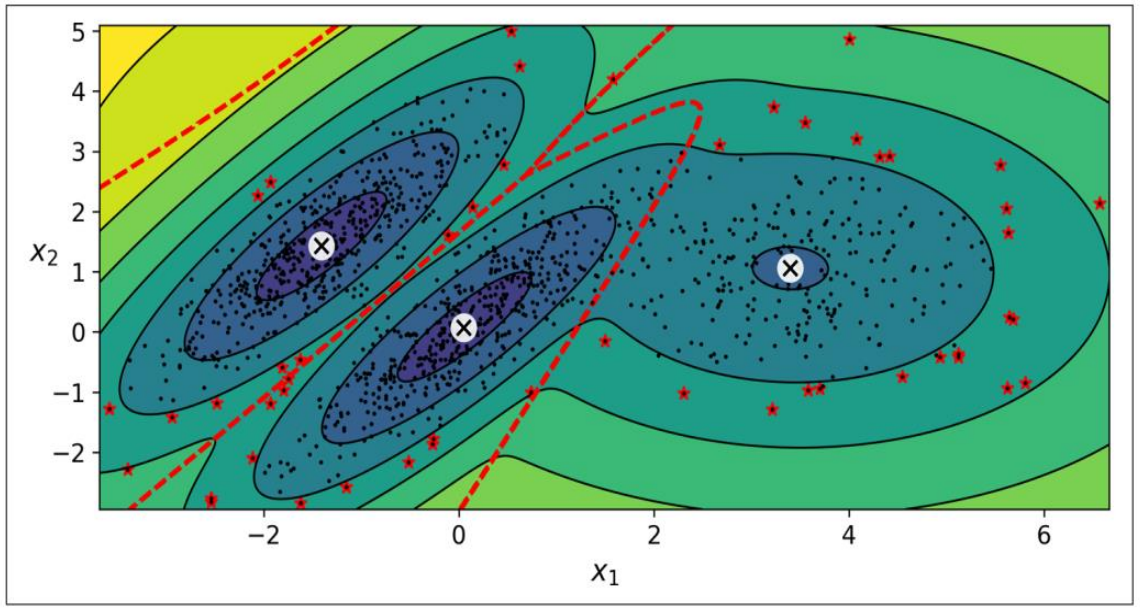


Figure 9-19. Anomaly detection using a Gaussian mixture model

Selecting the Number of Clusters

- With K-Means, you could use the inertia or the silhouette score to select the appropriate number of clusters. But with Gaussian mixtures, it is not possible to use these metrics because **they are not reliable when the clusters are not spherical or have different sizes.**
- Instead, you can try to find the model that minimizes a *theoretical information criterion*, such as the *Bayesian information criterion* (BIC) or the *Akaike information criterion* (AIC), defined in Equation 9-1.

Equation 9-1. Bayesian information criterion (BIC) and Akaike information criterion (AIC)

$$BIC = \log(m)p - 2\log\left(\hat{L}\right)$$

$$AIC = 2p - 2\log\left(\hat{L}\right)$$

In these equations:

- m is the number of instances, as always.
- p is the number of parameters learned by the model.
- $\hat{L}$ is the maximized value of the *likelihood function* of the model.

- Both the BIC and the AIC penalize models that have more parameters to learn (e.g., more clusters) and reward models that fit the data well.
- They often end up selecting the same model. When they differ, the model selected by the BIC tends to be simpler (fewer parameters) than the one selected by the AIC, but tends to not fit the data quite as well (this is especially true for larger datasets).

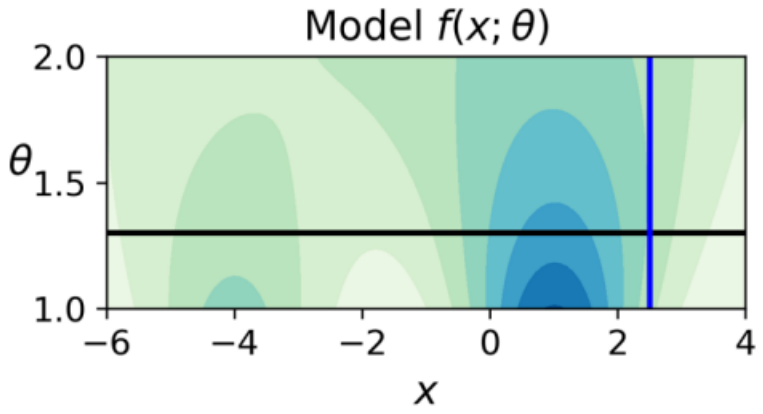
How AIC and BIC compute the maximum Likelihood??

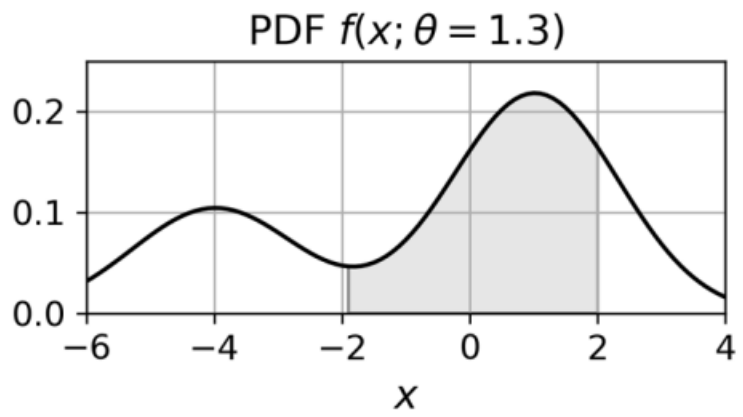
- Likelihood tells us how well a particular parameter value fits the observed data.

$$P(x,\theta) = P(x|\theta)\,P(\theta) = P(\theta|x)P(x)$$
$$= \mathcal{L}(\theta|x)P(\theta)$$

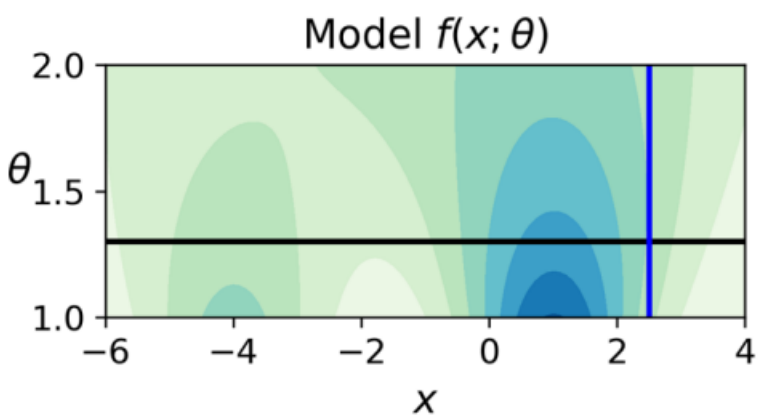
$P(\theta)$: Prior probability
$P(\theta|x)$: Posterior probability
$P(x|\theta)$: Likelihood
Likelihood function has an alternative notation $\mathcal{L}(\theta|x)$

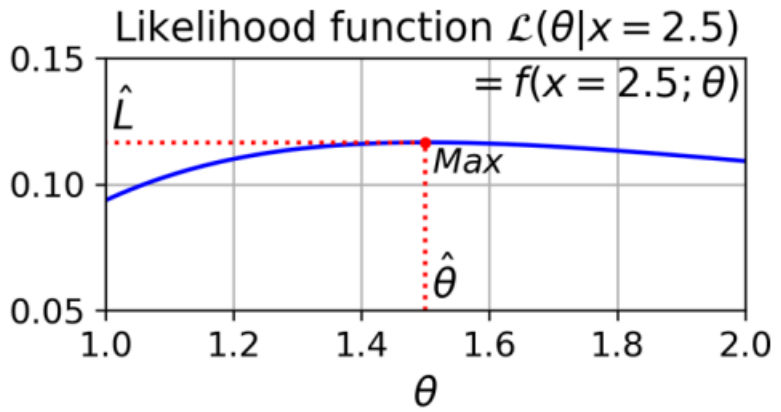
- Given a statistical model with some parameters **θ**, the word "probability" is used to describe how plausible a future outcome **x** is (knowing the parameter values **θ**), while the word "likelihood" is used to describe how plausible a particular set of parameter values **θ** are, after the outcome **x** is known.
- To estimate the probability distribution of a future outcome *x*, you need to set the model parameter θ. For example, if you set θ to 1.3 (the horizontal line), you get the probability density function $f(x; \theta=1.3)$ shown. Say you want to estimate the probability that *x* will fall between –2 and +2. You must calculate the integral of the PDF on this range (i.e., the surface of the shaded region).

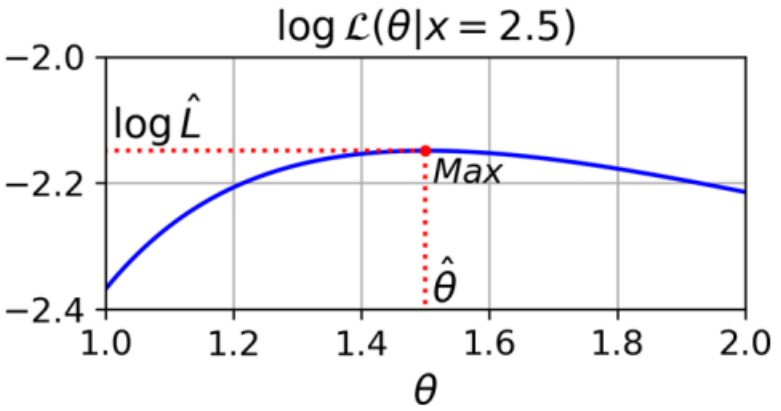




But what if you don't know θ, and instead if you have observed a single instance x=2.5 (the vertical line in the upper-left plot)? In this case, you get the likelihood function $\mathcal{L}(\theta|x=2.5)=f(x=2.5;\theta)$, represented in the plot.



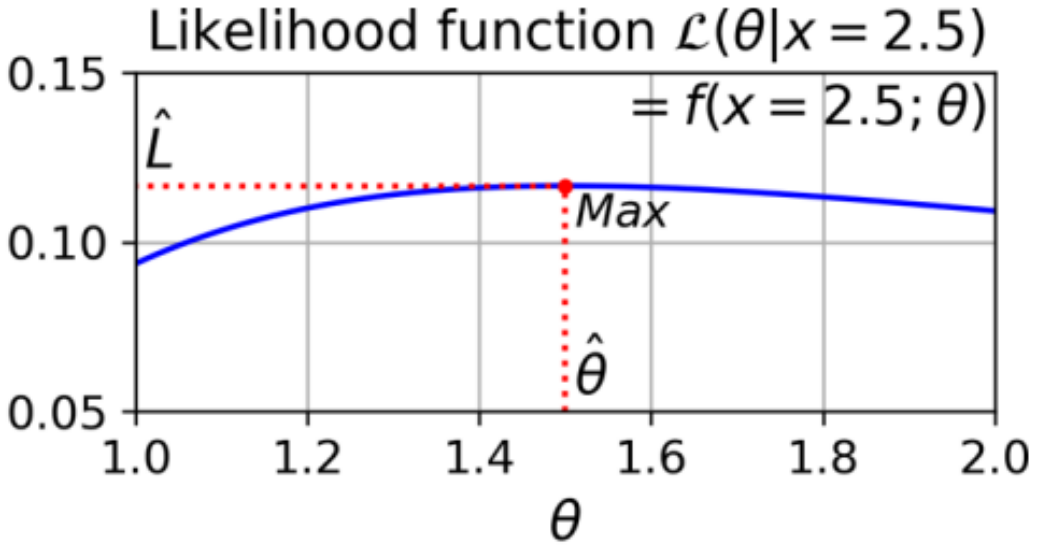




- In short, the PDF is a function of x (with θ fixed), while the likelihood function is a function of θ (with x fixed).
- It is important to understand that the likelihood function is *not* a probability distribution: if you integrate a probability distribution over all possible values of x, you always get 1; but if you integrate the likelihood function over all possible values of θ, the result can be any positive value.

- Given a dataset **X**, a common task is to try to estimate the most likely values for the model parameters. To do this, you must find the values that maximize the likelihood function, given **X**. In this example, if you have observed a single instance x=2.5, the

maximum likelihood estimate (MLE) of θ is θ =1.5.



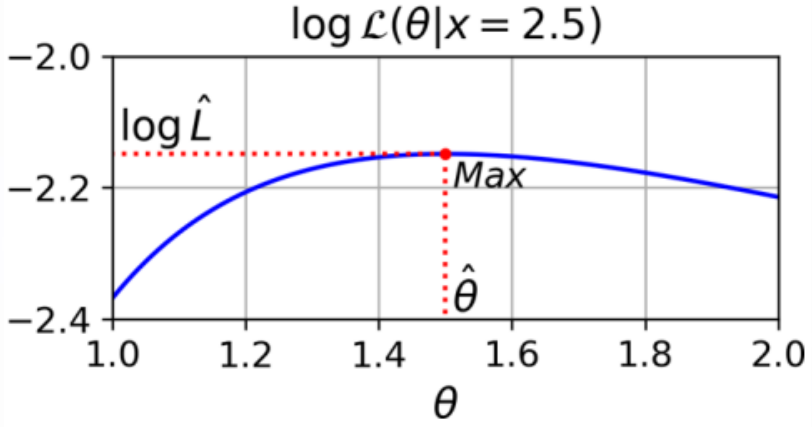
- If a prior probability distribution P over θ exists, it is possible to take it into account by maximizing $\mathcal{L}(\theta|x)\mathrm{P}(\theta)$ rather than just maximizing $\mathcal{L}(\theta|x)$.
- This is called *maximum a-posteriori* (MAP) estimation.
- Since MAP constrains the parameter values, you can think of it as a regularized version of MLE.

$$P(\theta|x)P(x) = \mathcal{L}(\theta|x)P(\theta)$$

$$P(\theta|x) = \frac{\mathcal{L}(\theta|x)P(\theta)}{P(x)} \xrightarrow{P(x)=cte} \mathcal{L}(\theta|x)P(\theta)$$

$P(\theta|x)$: Posterior probability

- Notice that maximizing the likelihood function is equivalent to maximizing its logarithm (represented in the figure).
- It is generally easier to maximize the log likelihood. For example, if you observed several independent instances $x^{(1)}$ to $x^{(m)}$, you would need to find the value of θ that maximizes the product of the individual likelihood functions.
- To maximize the sum (not the product) of the log likelihood functions, thanks to the magic of the logarithm which con- verts products into sums: $\log(ab)=\log(a)+\log(b)$.



Once you have estimated $\hat{\theta}$, the value of θ that maximizes the likelihood function, then you are ready to compute $\hat{L} = \mathscr{L}(\hat{\theta}, \mathbf{X})$, which is the value used to compute the AIC and BIC; you can think of it as a measure of how well the model fits the data.

To compute the BIC and AIC, call the `bic()` and `aic()` methods:

```
>>> gm.bic(X)
8189.74345832983
>>> gm.aic(X)
8102.518178214792
```

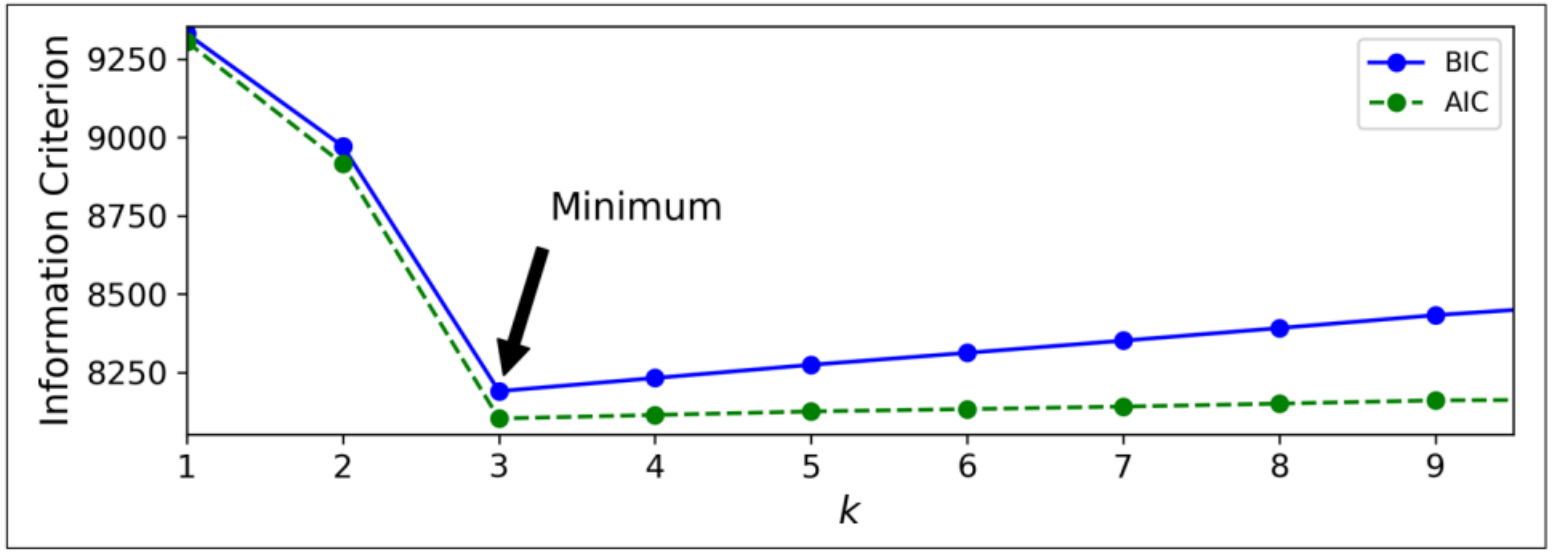


Figure 9-21. AIC and BIC for different numbers of clusters k

Bayesian Gaussian Mixture Models

- Rather than manually searching for the optimal number of clusters, you can use the `BayesianGaussianMixture` class, which is capable of giving weights equal (or close) to zero to unnecessary clusters.
- Set the number of clusters `n_components` to a value that you have good reason to believe is greater than the optimal number of clusters (this assumes some minimal knowledge about the problem at hand), and the algorithm will eliminate the unnecessary clusters automatically. For example, let's set the number of clusters to 10 and see what happens:

```
>>> from sklearn.mixture import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10)
>>> bgm.fit(X)
>>> np.round(bgm.weights_, 2)
array([0.4 , 0.21, 0.4 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

Gaussian mixture models work great on clusters with ellipsoidal shapes, but if you try to fit a dataset with different shapes, you may have bad surprises. For example, let's see what happens if we use a Bayesian Gaussian mixture model to cluster the moons dataset (see Figure 9-24).

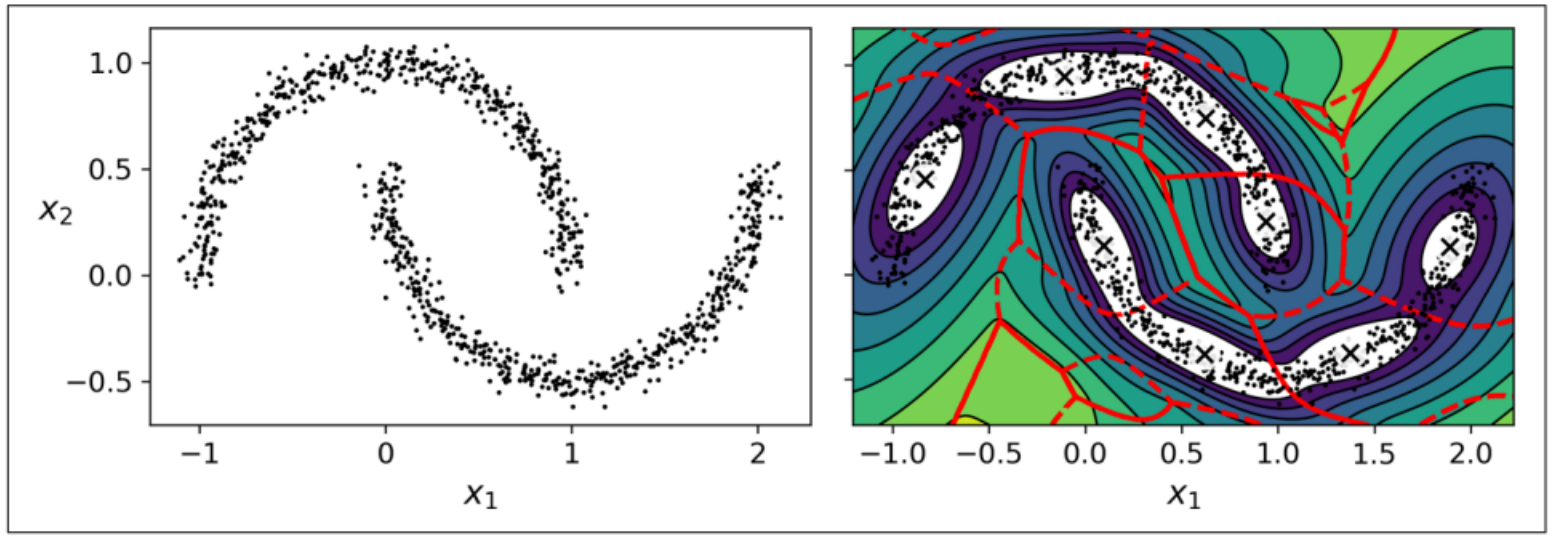


Figure 9-24. Fitting a Gaussian mixture to nonellipsoidal clusters

- The algorithm desperately searched for ellipsoids, so it found eight different clusters instead of two.
- The density estimation is not too bad, so this model could perhaps be used for anomaly detection, but it failed to identify the two moons