4.
DBSCAN

DBSCAN

- The *density-based spatial clustering of applications with noise* algorithm defines clusters as continuous regions of high density:
  1. For each instance, count how many instances are located within a small distance ε from it. This region is called the instance's *ε-neighborhood*.
  2. 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.
  3. 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.
  4. Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

DBSCAN in Scikit-Learn

- DBSCAN works well if all the clusters are well separated by low-density regions.
- Let's test it on the moons dataset:

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

X, y = make_moons(n_samples=1000, noise=0.05, random_state=42)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

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

```
dbscan.labels_[:10]
```

```
array([ 0,  2, -1, -1,  1,  0,  0,  0,  2,  5], dtype=int64)
```

DBSCAN in Scikit-Learn

- The indices of the core instances are in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

```
dbscan.core_sample_indices_[:10]
```

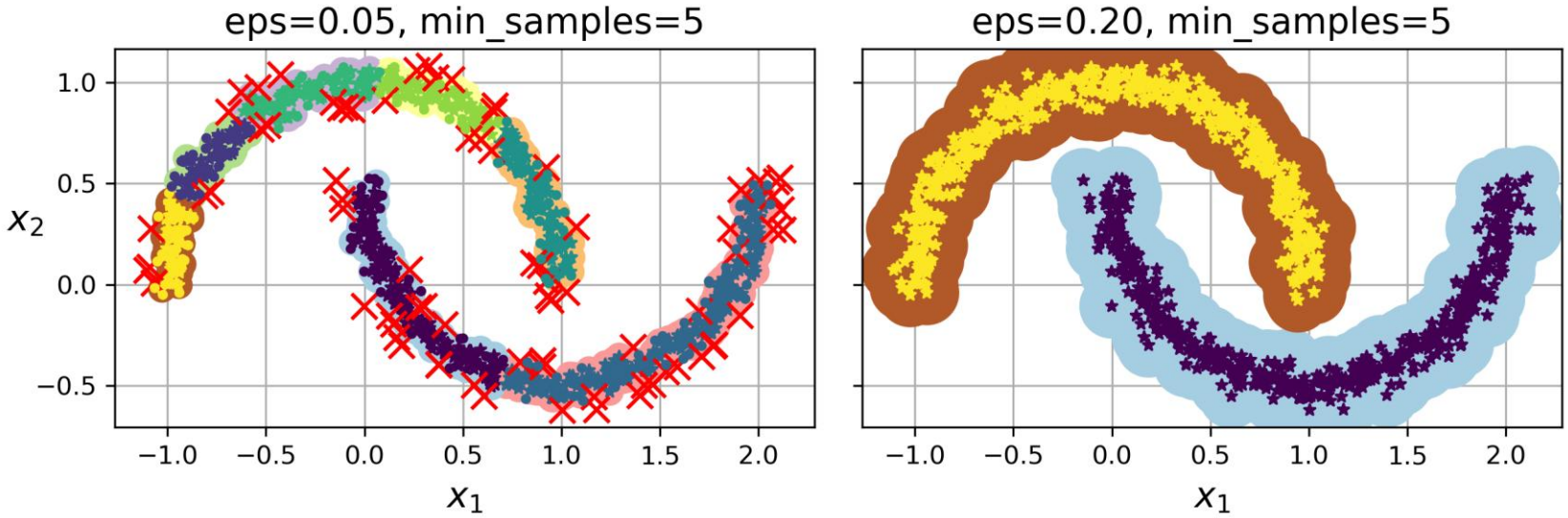
```
array([ 0,  4,  5,  6,  7,  8, 10, 11, 12, 13], dtype=int64)
```

```
dbscan.components_
```

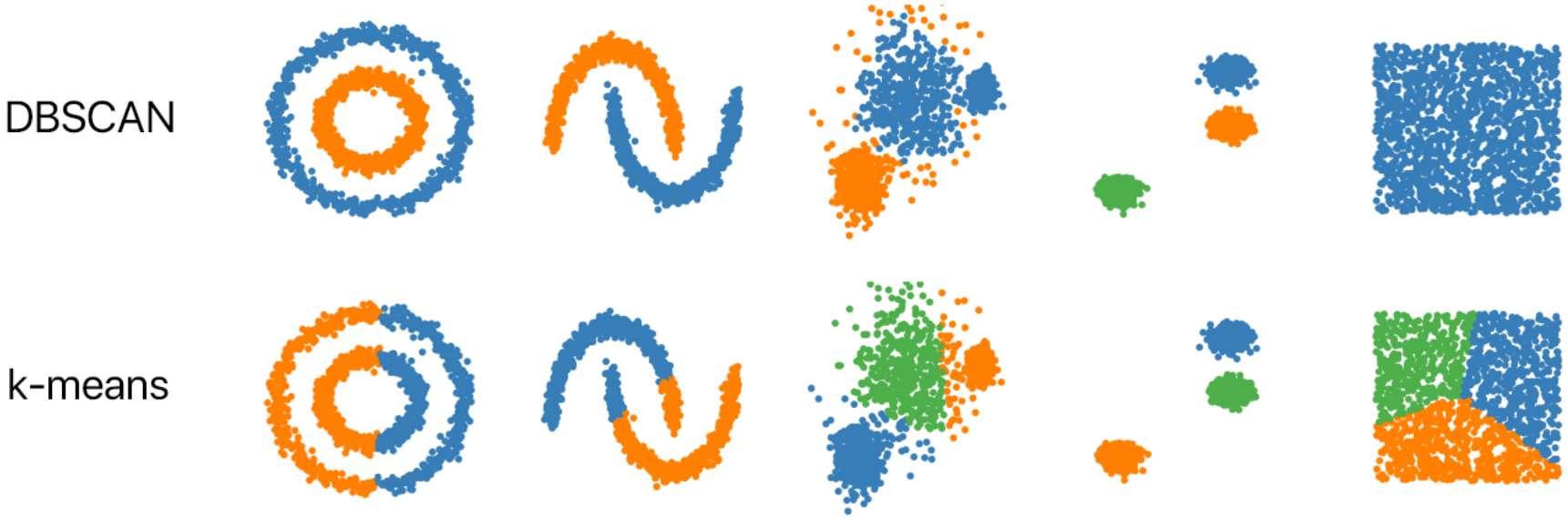
```
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       [ 0.58930337, -0.32137599],
       ...,
       [ 1.66258462, -0.3079193 ],
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```

Neighborhood Radius Effect

- DBSCAN clustering using two different neighborhood radiuses



DBSCAN vs. K-means



DBSCAN in Scikit-Learn

- The `DBSCAN` class does not have a `predict()` method, although it has a `fit_predict()` method.
  - It cannot predict which cluster a new instance belongs to.
- But we can implement it (e.g. use 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 clusters 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], dtype=int64)
```

```
knn.predict_proba(X_new)
```

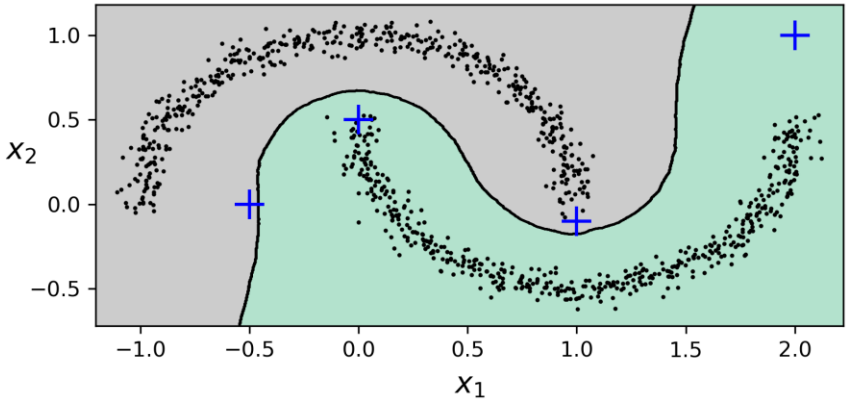
```
array([[0.18, 0.82],
       [1.  , 0.  ],
       [0.12, 0.88],
       [1.  , 0.  ]])
```

DBSCAN in Scikit-Learn

- You can introduce a maximum distance, in case the two instances that are far away from both clusters are classified as anomalies.

```
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], dtype=int64)
```

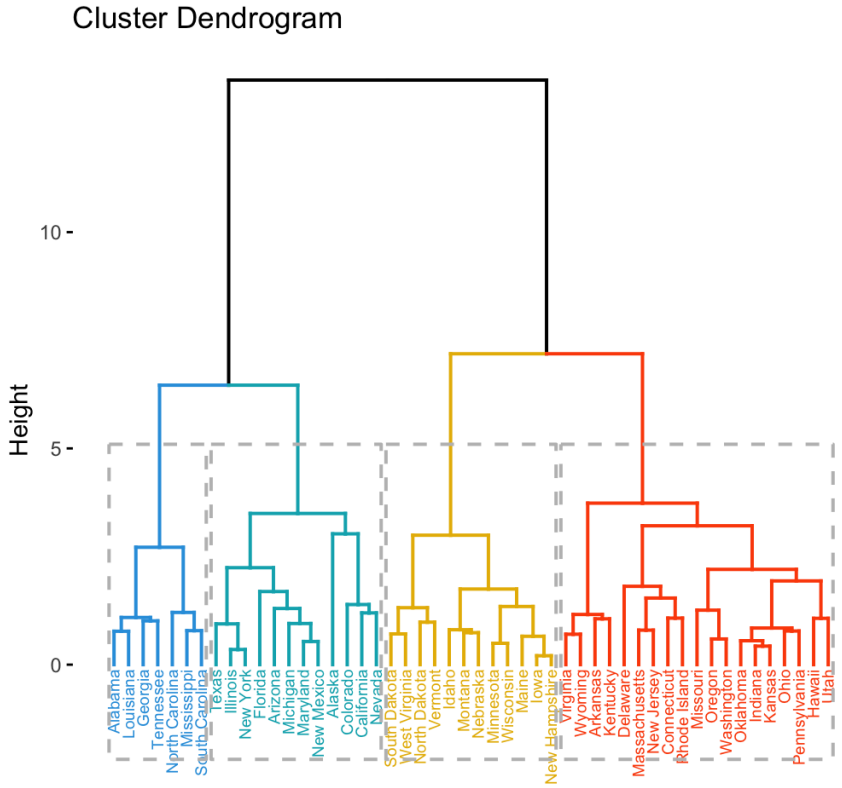


DBSCAN Pros and Cons

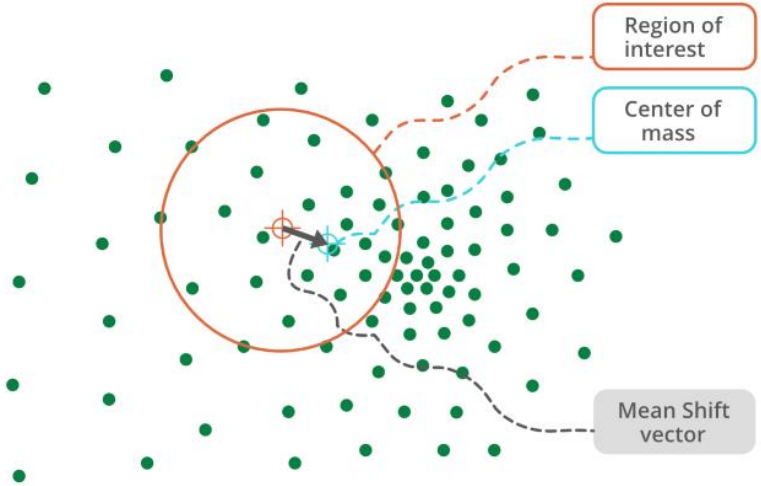
- DBSCAN is a simple yet powerful algorithm capable of identifying any number of clusters of any shape.
- It is robust to outliers, and it has just two hyperparameters (`eps` and `min_samples`).
- If the density varies significantly across the clusters, or if there's no sufficiently low-density region around some clusters, DBSCAN can struggle to capture all the clusters properly.
- Its computational complexity is roughly $O(m^2 n)$, so it does not scale well to large datasets.
- Also try *hierarchical DBSCAN* (HDBSCAN), which is usually better than DBSCAN at finding clusters of varying densities.

Other Clustering Algorithms

- *Agglomerative clustering* builds a hierarchy of clusters from the bottom up. At each iteration, the algorithm connects the nearest pair of clusters (starting with individual instances).
- *BIRCH* is designed specifically for very large datasets, and can be faster than batch k-means, if number of features is not too large (<20).
- *Mean-shift* places a circle centered on each instance; then for each circle it computes the mean of all the instances within it, and shifts the circle so that it is centered on the mean. It iterates this mean-shifting step until all the circles stop moving.
  - Computational complexity is $O(m^2 n)$ → not suited for large datasets.
  - Unlike DBSCAN, it tends to chop clusters into pieces when they have internal density variations.



Agglomerative clustering



Mean-shift clustering

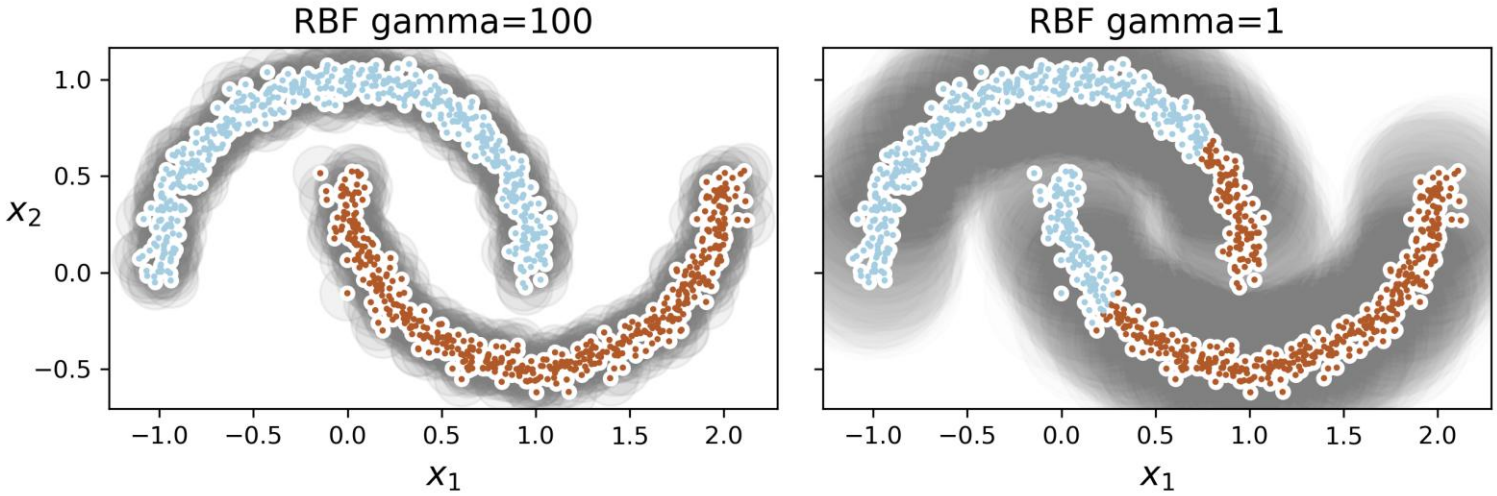
Other Clustering Algorithms

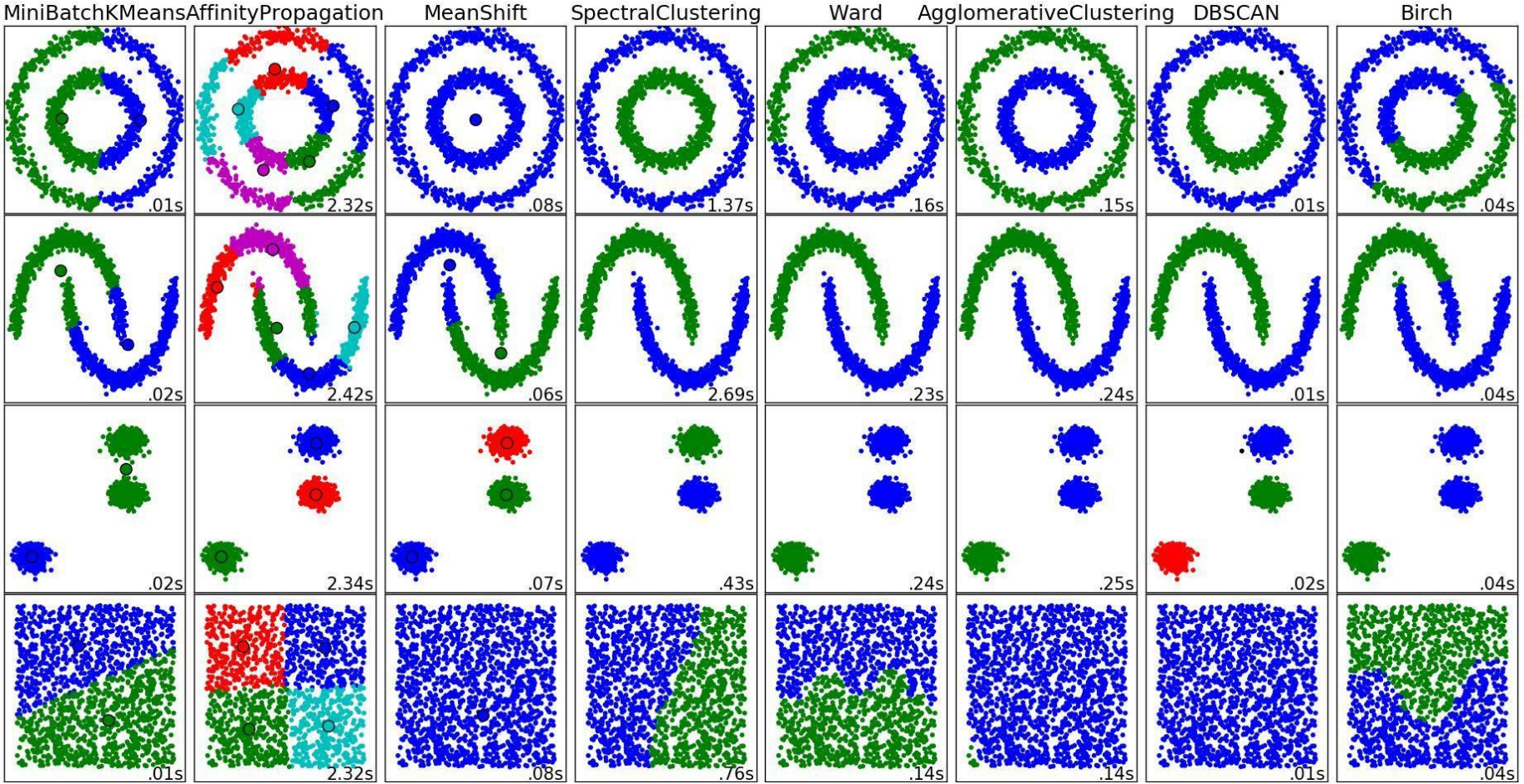
- *Affinity propagation*: instances repeatedly exchange messages until every instance has elected another instance (or itself) to represent it. These elected instances are called *exemplars*. Each exemplar and all the instances that elected it form one cluster.
  - Unlike k-means, you don't pick the number of clusters ahead of time.
  - It has a complexity of $O(m^2)$, so it is not suited for large datasets.
- *Spectral clustering* takes a similarity matrix between the instances and creates a low-dimensional embedding from it, then it uses another clustering algorithm in this low-dimensional space.
  - It can capture complex cluster structures, and be used to cut graphs.
  - It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

Spectral Clustering

```
from sklearn.cluster import SpectralClustering
```

```
sc1 = SpectralClustering(n_clusters=2, gamma=100, random_state=42)
sc1.fit(X)
```



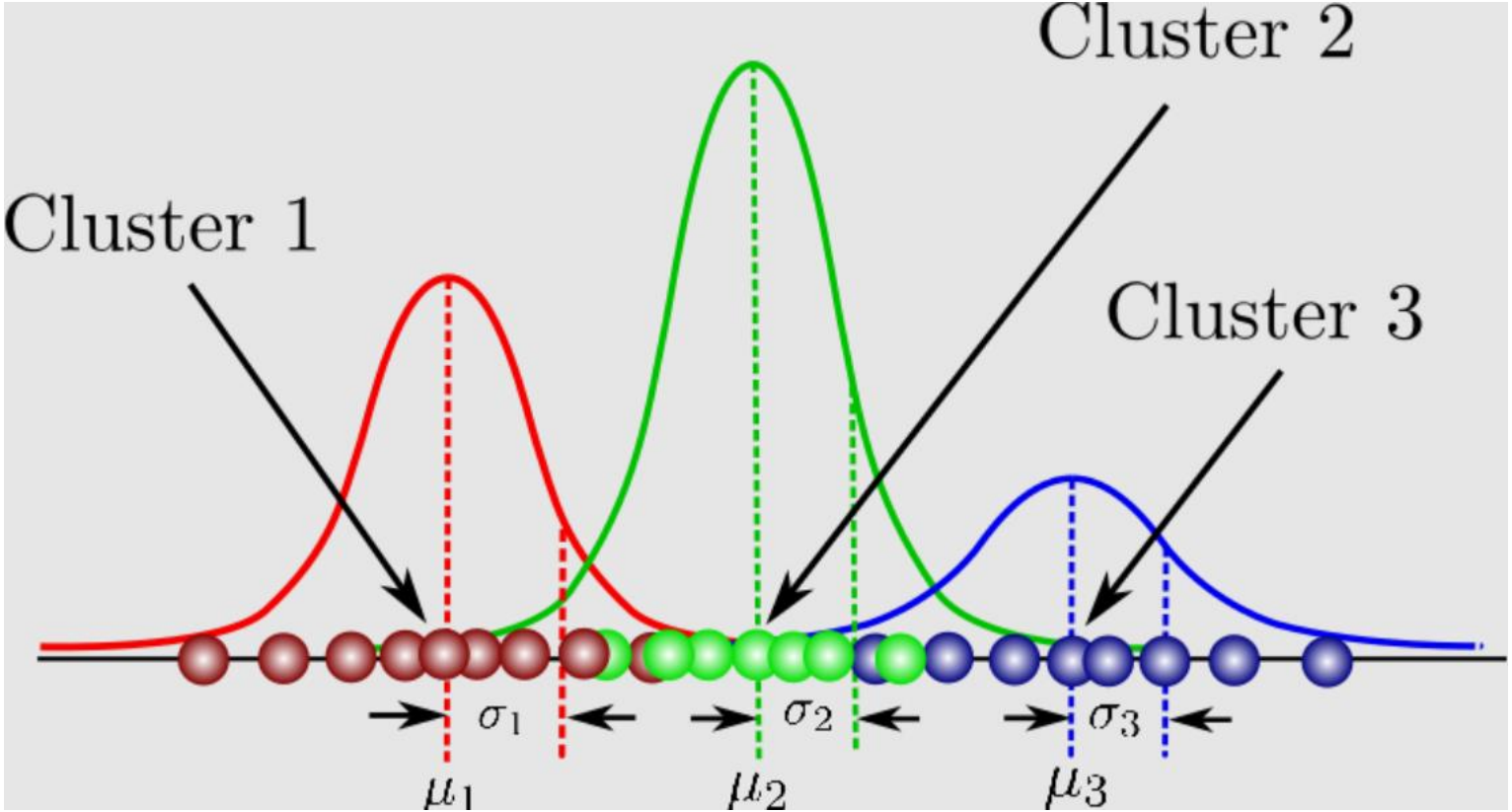

MiniBatchKMeans
AffinityPropagation
MeanShift
SpectralClustering
Ward
AgglomerativeClustering
DBSCAN
Birch
.01s
2.32s
.08s
1.37s
.16s
.15s
.01s
.04s
.02s
2.42s
.06s
2.69s
.23s
.24s
.01s
.04s
.02s
2.34s
.07s
.43s
.24s
.25s
.02s
.04s
.01s
2.32s
.08s
.76s
.14s
.14s
.01s
.04s

5.
Gaussian Mixture Models

Gaussian Mixture Model

- 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.
- All the instances generated from a single Gaussian distribution form a cluster that typically looks like an ellipsoid.
  - 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.

Gaussian Mixture Model

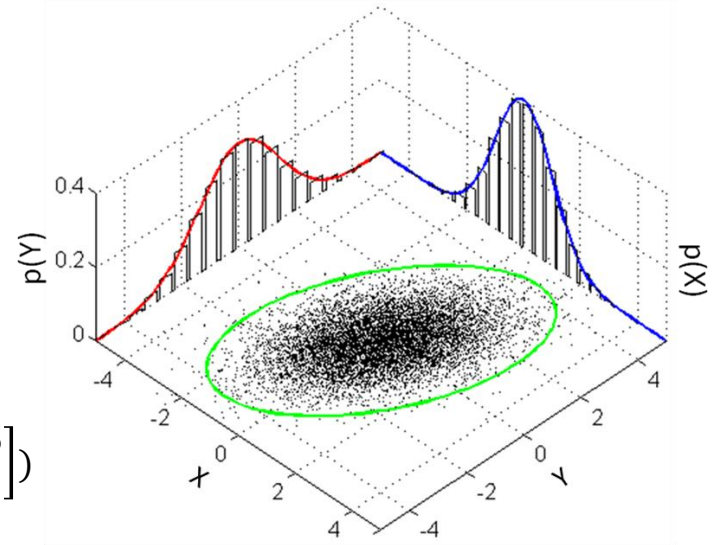


Multivariate Normal Distribution

- A generalization of normal distribution to higher dimensions:
  - $N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$: $\boldsymbol{\mu}$= mean vector, $\boldsymbol{\Sigma}$= covariance matrix

$$f(x,y) = \frac{1}{2\pi\sigma_X\sigma_Y\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2\left[1-\rho^2\right]}\left[\left(\frac{x-\mu_X}{\sigma_X}\right)^2 - 2\rho\left(\frac{x-\mu_X}{\sigma_X}\right)\left(\frac{y-\mu_Y}{\sigma_Y}\right) + \left(\frac{y-\mu_Y}{\sigma_Y}\right)^2\right]\right)$$

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_X \\ \mu_Y \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2 \end{pmatrix}$$



$$N(\boldsymbol{\mu} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \boldsymbol{\Sigma} = \begin{bmatrix} 1 & 3/5 \\ 3/5 & 2 \end{bmatrix})$$

Gaussian Mixture Model

- The simplest variant of GMM, implemented in the `GaussianMixture` class, assumes the number k of Gaussian distributions is known.
- The dataset $\mathbf{X}$ is assumed to have been generated through the following probabilistic process:
  - For each instance, a cluster is picked randomly from k clusters.
    - The probability of choosing the j-th cluster is the cluster's weight $\phi^{(j)}$.
    - The index of the cluster chosen for the i-th instance is noted $z^{(i)}$.
  - If the i-th instance was assigned to the j-th cluster (i.e., $z^{(i)} = j$), then 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 $\mathbf{x}^{(i)} \sim N(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)})$

Gaussian Mixture Model

- So what can you do with such a model?
  - Given the dataset $\mathbf{X}$, you want to start by estimating the weights $\boldsymbol{\phi}$ and all the distribution parameters $\boldsymbol{\mu}^{(1)}$ to $\boldsymbol{\mu}^{(k)}$ and $\boldsymbol{\Sigma}^{(1)}$ to $\boldsymbol{\Sigma}^{(k)}$.
  - Use Scikit-Learn's `GaussianMixture` class for this:

```
from sklearn.mixture import GaussianMixture
```

```
gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

```
gm.weights_
```

```
array([0.39025715, 0.40007391, 0.20966893])
```

```
gm.means_
```

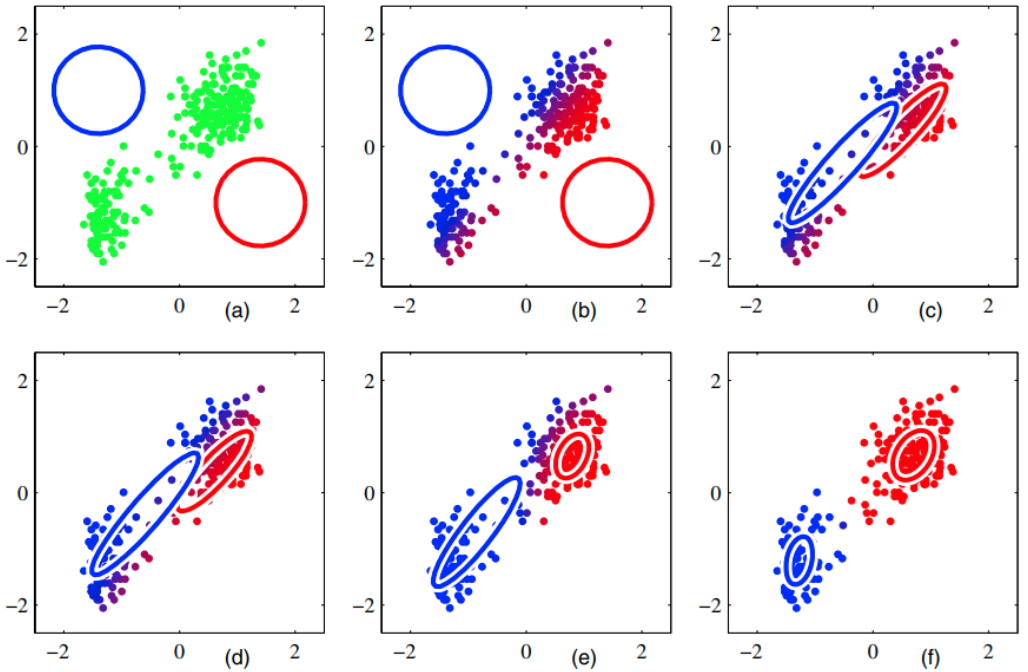
```
array([[ 0.05131611,  0.07521837],
       [-1.40763156,  1.42708225],
       [ 3.39893794,  1.05928897]])
```

```
gm.covariances_
```

```
array([[[ 0.68799922,  0.79606357],
        [ 0.79606357,  1.21236106]],

       [[ 0.63479409,  0.72970799],
        [ 0.72970799,  1.1610351 ]],

       [[ 1.14833585, -0.03256179],
        [-0.03256179,  0.95490931]]])
```

Expectation Maximization Algorithm

- `GaussianMixture` class relies on the *expectation-maximization* (EM) algorithm, which is similar to the k-means algorithm:
  - it initializes cluster parameters randomly.
  - it repeats two steps until convergence:
    1. assigning instances to clusters (this is called the *expectation step*)
    2. updating the clusters (this is called the *maximization step*)



Expectation Maximization Algorithm

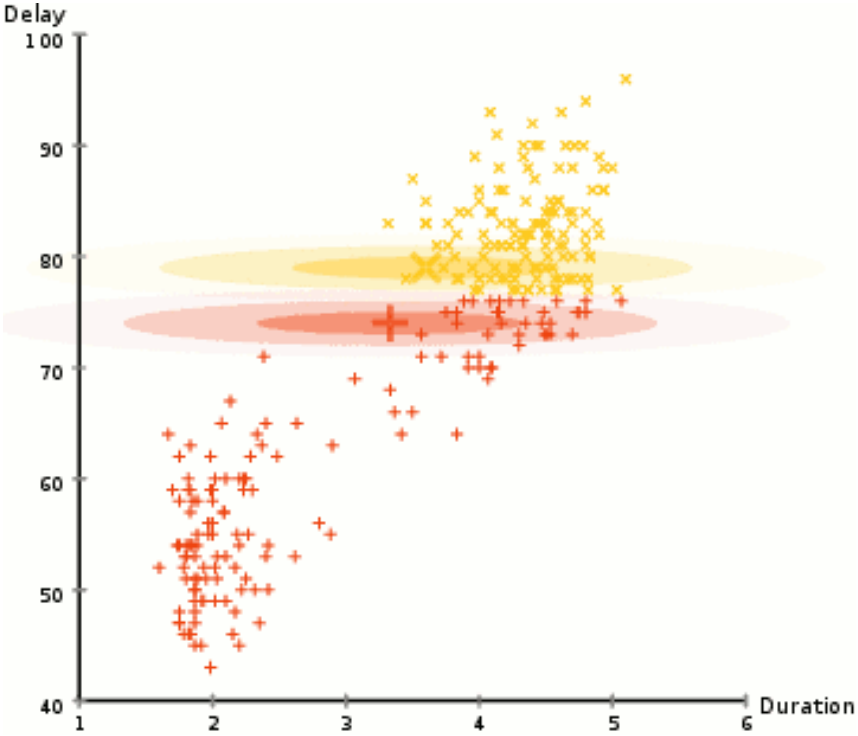
- You can think of EM as a generalization of k-means that not only finds the cluster centers ($\boldsymbol{\mu}^{(1)}$ to $\boldsymbol{\mu}^{(k)}$), but also their size, shape, and orientation ($\boldsymbol{\Sigma}^{(1)}$ to $\boldsymbol{\Sigma}^{(k)}$), and their relative weights ($\phi^{(1)}$ to $\phi^{(k)}$).

- Unlike k-means, though, EM uses soft cluster assignments, not hard assignments.

Soft Cluster Assignment

- In the *expectation step*, for each instance, the algorithm estimates the probability that it belongs to each cluster.
- In the *maximization step*, each cluster is updated using *all* instances in the dataset, with each instance weighted by the estimated probability that it belongs to that cluster.
  - These probabilities are called the *responsibilities* of the clusters for the instances.
- During the maximization step, each cluster's update will mostly be impacted by the instances it is most responsible for.
- Like k-means, EM can end up converging to poor solutions, so it needs to be run several times, keeping only the best solution.

EM updating the parameters of a GMM



Gaussian Mixture Model

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

```
gm.converged_
```

```
True
```

```
gm.n_iter_
```

```
4
```

- Having an estimate of the location, size, shape, orientation, and relative weight of each cluster, the model can:
  - assign each instance to the most likely cluster (hard clustering)
  - or estimate the probability that it belongs to a particular cluster (soft clustering).

Clustering using GMM

- Use the `predict()` method for hard clustering, or the `predict_proba()` method for soft clustering:

```
gm.predict(X)
```

```
array([2, 2, 0, ..., 1, 1, 1], dtype=int64)
```

```
gm.predict_proba(X).round(3)
```

```
array([[0.   , 0.023, 0.977],
       [0.001, 0.016, 0.983],
       [1.   , 0.   , 0.   ],
       ...,
       [0.   , 1.   , 0.   ],
       [0.   , 1.   , 0.   ],
       [0.   , 1.   , 0.   ]])
```

GMM as a Generative Model

- GMM is a *generative model*, meaning you can sample new instances from it:

```
X_new, y_new = gm.sample(6)
X_new
```

```
array([[-0.86944074, -0.32767626],
       [ 0.29836051,  0.28297011],
       [-2.8014927 , -0.09047309],
       [ 3.98203732,  1.49951491],
       [ 3.81677148,  0.53095244],
       [ 2.84104923, -0.73858639]])
```

```
y_new
```

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

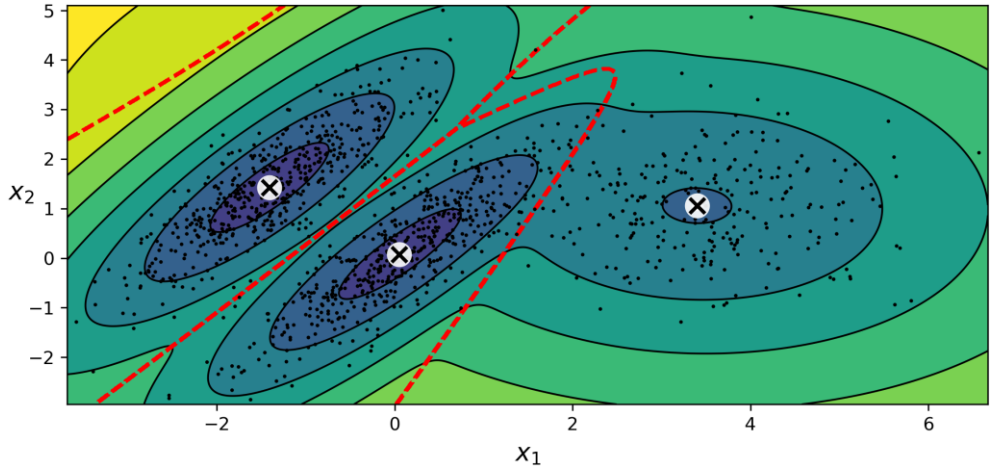
- It is possible to estimate the density of the model at any given location using the `score_samples()` method:

```
gm.score_samples(X).round(2)
```

```
array([-2.61, -3.57, -3.33, ..., -3.51, -4.4 , -3.81])
```

Density Contours

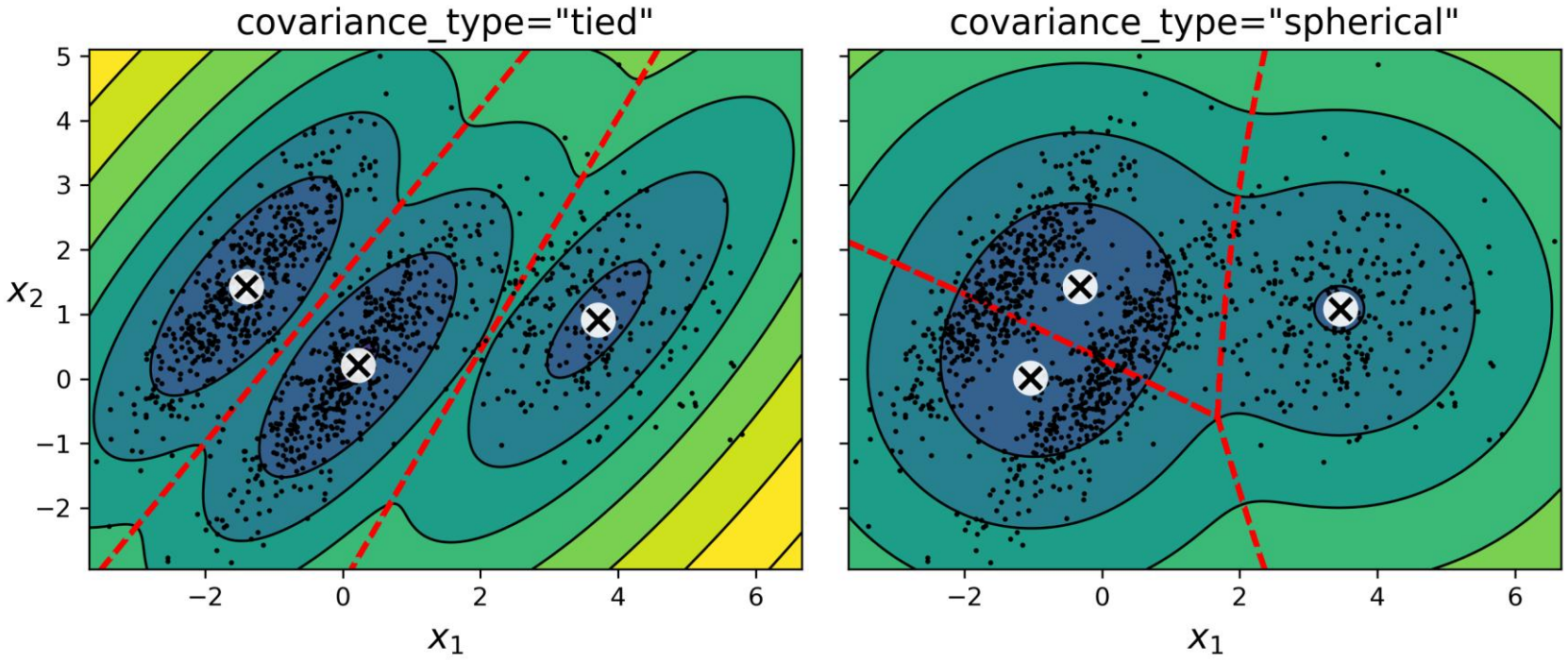
- If you compute the exponential of these scores, you get the value of the PDF at the location of the given instances.
- The cluster means, the decision boundaries (dashed lines), and the density contours of the trained GMM:



Covariance Type

- When there are many dimensions, or many clusters, or few instances, EM can struggle to converge to the optimal solution.
  - You need to reduce the difficulty of the task by limiting the number of parameters that the algorithm has to learn.
- To do this, limit the range of shapes and orientations that the clusters can have by imposing constraints on the covariance matrices.
- Set the `covariance_type` hyperparameter to:
  - "`spherical`": clusters must be spherical, but can have different diameters.
  - "`diag`": clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the coordinate axes.
  - "`tied`": clusters must have the same ellipsoidal shape, size, and orientation

Covariance Type



Complexity of GMM

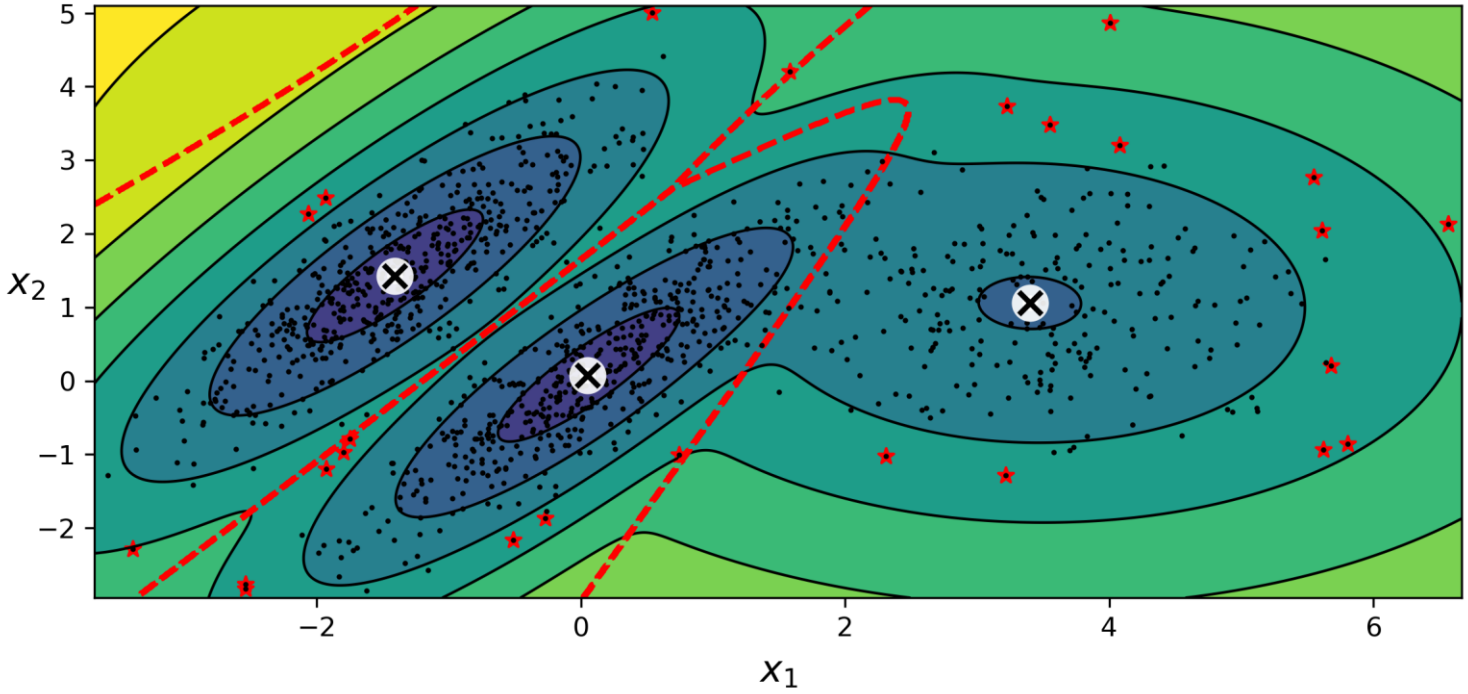
- The computational complexity of training a `GaussianMixture` model depends on:
  - the number of instances m
  - the number of dimensions n
  - the number of clusters k
  - the constraints on the covariance matrices
- If `covariance_type` is "`spherical`" or "`diag`", it is $O(kmn)$, assuming the data has a clustering structure.
- If `covariance_type` is "`tied`" or "`full`", it is $O(kmn^2 + kn^3)$, so it will not scale to large numbers of features.

Using GMM for Anomaly Detection

- In a GMM any instance located in a low-density region can be considered an anomaly.
- You must define what density threshold you want to use.
  - 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.
  - If you have too many false negatives (i.e., defective products that the system does not flag as defective), you can increase the threshold.
  - This is the usual precision/recall trade-off
- To identify the outliers using the second percentile lowest density as the threshold:

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 2)
anomalies = X[densities < density_threshold]
```

Using GMM for Anomaly Detection



Selecting the Number of Clusters

- With Gaussian mixtures, it is not possible to use the inertia or the silhouette score to select the appropriate number of clusters.
  - These metrics are not reliable when the clusters are not spherical or have different sizes.
- Find the model that minimizes a *theoretical information criterion*, such as the *Bayesian information criterion* (BIC) or the *Akaike information criterion* (AIC):

$$\mathrm{BIC} = \log(m)\, p - 2\log(\hat{\Phi})$$

$$\mathrm{AIC} = 2p - 2\log(\hat{\Phi})$$

  - m is the number of instances
  - p is the number of parameters learned by the model
  - $\hat{\Phi}$ is the maximized value of the *likelihood function* of the model

BIC and AIC

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

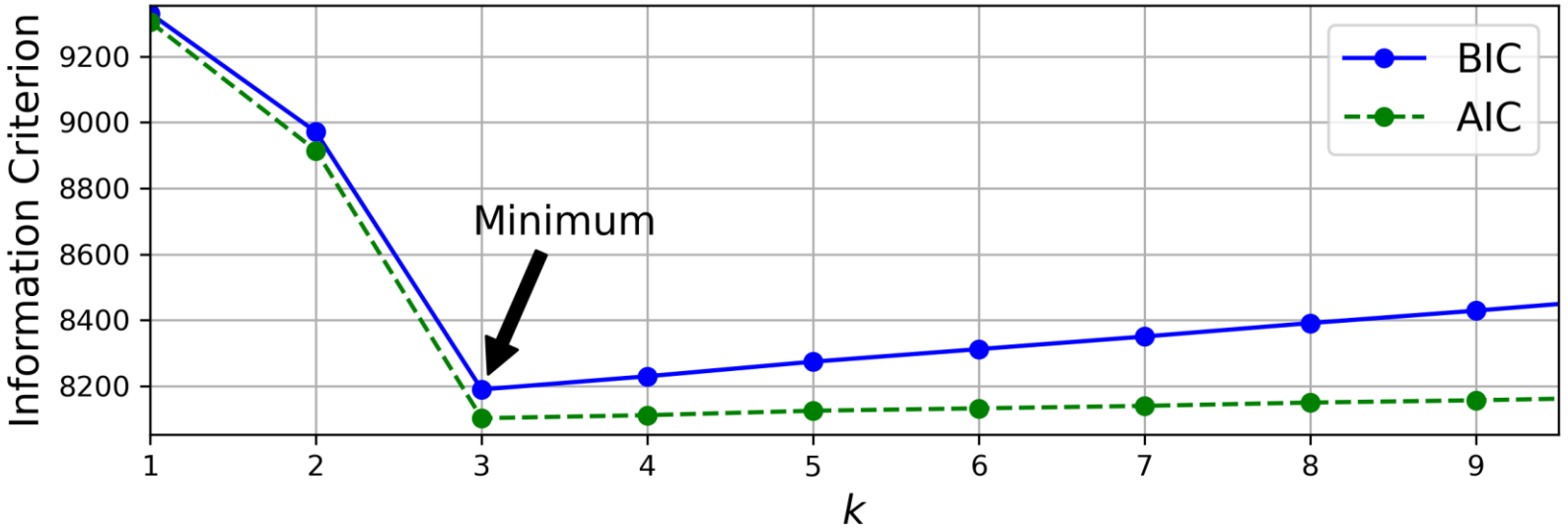
```
gm.bic(X)
```

```
8189.733705221638
```

```
gm.aic(X)
```

```
8102.5084251066
```

Selecting the Number of Clusters



Bayesian Gaussian Mixture Models

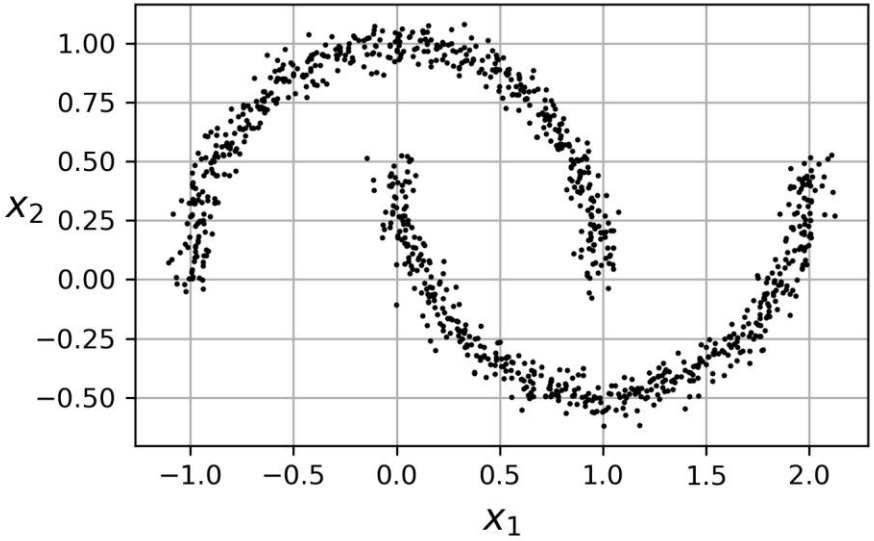
- Rather than manually searching for the optimal number of clusters, we can use the `BayesianGaussianMixture` class, which is capable of giving weights equal (or close) to zero to unnecessary clusters.
- Set the number of clusters `n_components` to a value that you have good reason to believe is greater than the optimal number of clusters and the algorithm will eliminate the unnecessary clusters automatically.

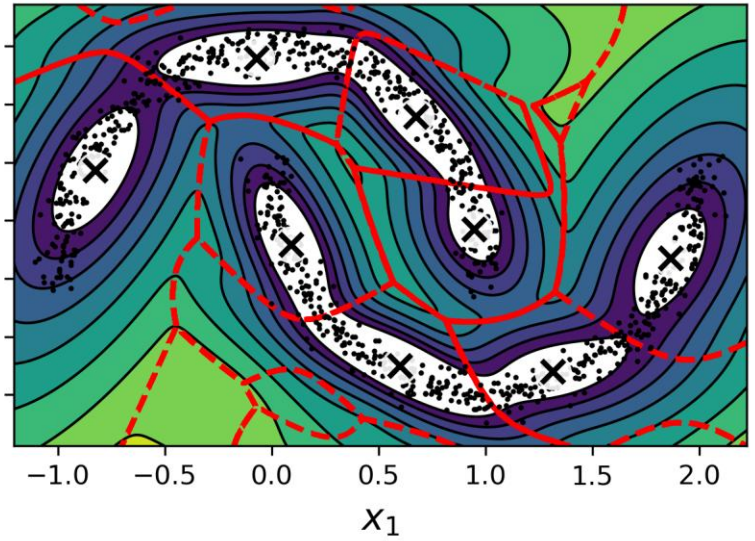
```
from sklearn.mixture import BayesianGaussianMixture

bgm = BayesianGaussianMixture(n_components=10, n_init=10, random_state=42)
bgm.fit(X)
bgm.weights_.round(2)
```

```
array([0.4 , 0.21, 0.4 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

Bayesian Gaussian Mixture Models

- Although GMMs work great on clusters with ellipsoidal shapes, they don't do so well with clusters of very different shapes.





6.
Other Algorithms for Anomaly Detection

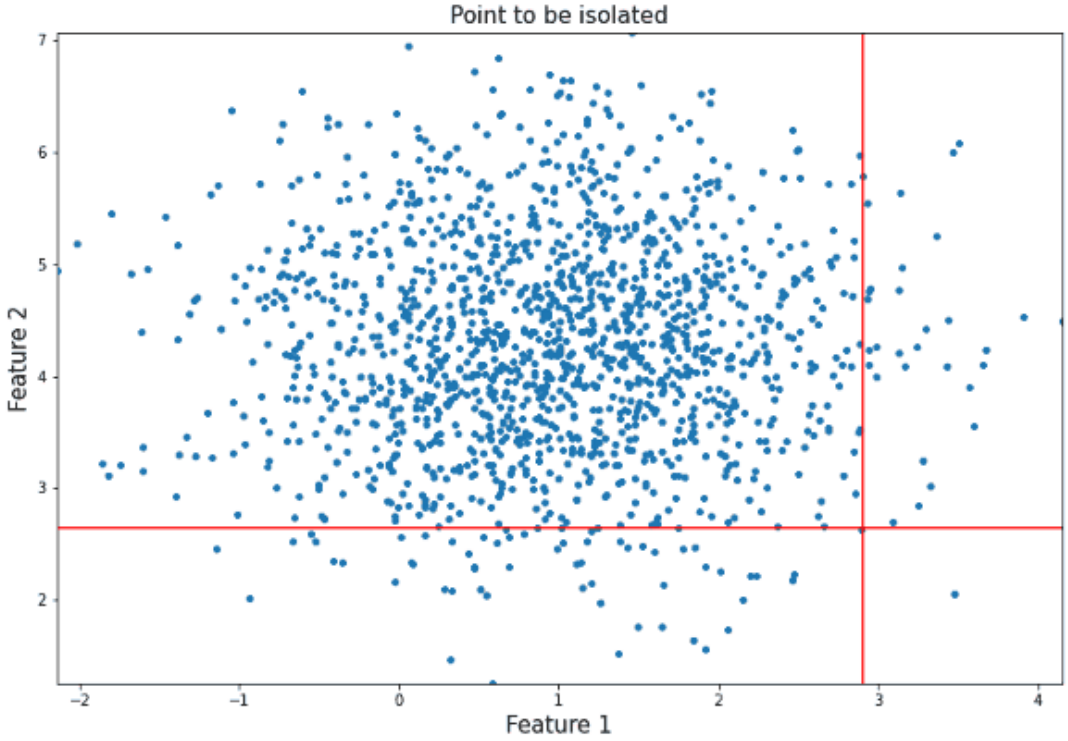
Fast-MCD

- *Fast-MCD (minimum covariance determinant)*: implemented by the `EllipticEnvelope` class, this algorithm is useful for outlier detection, in particular to clean up a dataset. It assumes that:
  1. The normal instances (inliers) are generated from a single Gaussian distribution (not a mixture).
  2. The dataset is contaminated with outliers that were not generated from this Gaussian distribution.
- When Fast-MCD estimates the parameters of the Gaussian distribution (i.e., the shape of the elliptic envelope around the inliers), it is careful to ignore the instances that are most likely outliers.
  - This technique gives a better estimation of the elliptic envelope and thus makes the algorithm better at identifying the outliers.

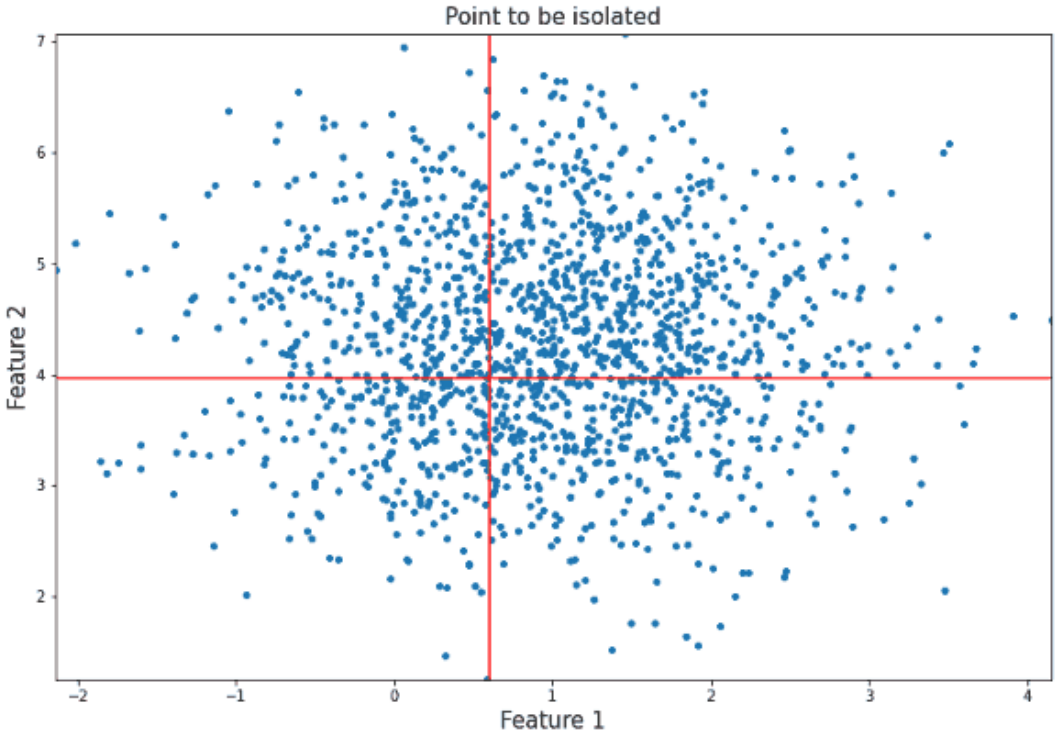
Isolation forest

- *Isolation forest* is an efficient algorithm for outlier detection, especially in high-dimensional datasets.
- The algorithm builds a random forest in which each decision tree is grown randomly: at each node, it picks a feature randomly, then it picks a random threshold value to split the dataset in two.
  - The dataset gradually gets chopped into pieces this way, until all instances end up isolated from the other instances.
- Anomalies are usually far from other instances, so on average (across all the decision trees) they tend to get isolated in fewer steps than normal instances.

Isolation Forest

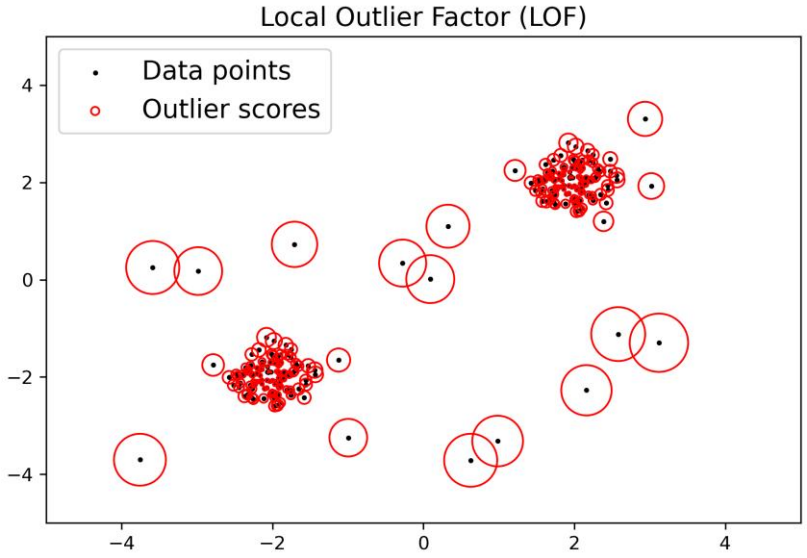


Isolation Forest



Local Outlier Factor

- *Local outlier factor (LOF)*: this algorithm compares the density of instances around a given instance to the density around its neighbors.
- An anomaly is often more isolated than its k-nearest neighbors.



One-class SVM

- *One-class SVM* is better suited for novelty detection.
- A kernelized SVM classifier separates two classes by mapping all the instances to a high-dimensional (HD) space, and separating the two classes using a linear SVM classifier.
- We have one class of instances, so the one-class SVM algorithm instead tries to separate the instances in HD space from the origin.
  - In the original space, this will correspond to finding a small region that encompasses all the instances.
  - If a new instance does not fall within this region, it is an anomaly.
- It works great, especially with high-dimensional datasets, but like all SVMs it does not scale to large datasets.

Dimensionality Reduction Techniques

- *PCA and other dimensionality reduction techniques with an* `inverse_transform()` *method:*

- If you compare the reconstruction error of a normal instance with the reconstruction error of an anomaly, the latter will usually be much larger.

- This is a simple and often quite efficient anomaly detection approach.