Unsupervised Learning Techniques

- The vast majority of the available data is unlabeled: we have the input features **X,** but we do not have the labels **Y**
- Most common Unsupervised Learning tasks and algorithms:
  - Dimensionality Reduction (CH8)
  - 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. These instances are called **anomalies**, or **outliers**, while the normal instances are called **inliers**.
    - Anomaly detection is useful in a wide variety of applications, such as fraud detection, detecting defective products in manufacturing, identifying new trends in time series, or removing outliers from a dataset before training another model
  - 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

- **Clustering** is the task identifying similar instances and assigning them to **clusters**, or groups of similar instances
- **Clustering** is used in wide variety of applications:
  - **Customer Segmentation**
    - You can cluster your customers based on their purchases and their activity on your website.
    - **Customer segmentation** can be useful in recommender systems to suggest content that other users in the same cluster enjoyed.
  - **Data Analysis**
    - When you analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.
  - **Dimensionality Reduction Technique**
    - Once a dataset has been clustered, it is usually possible to measure each instance's **affinity**(any measure of how well an instance fits into a cluster)with each cluster.
    - Each instance's **feature** vector **x** can then be replaced with the vector of its cluster **affinities**. If there are **k** clusters, then this vector is **k-dimensional**.
    - The new vector is typically much lower-dimensional than the original feature vector, but it can preserve enough information for further processing.
  - **Anomaly Detection**
    - Any instance that has a low affinity to all the clusters is likely to be an anomaly
  - **Feature Engineering**
    - The cluster affinities can often be useful as extra features
  - **Semi-supervised Learning**
    - If you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster.
  - **Search Engines**
    - Some search engines let you search for images that are similar to a reference image.

- **Image Segmentation**
  - By clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster,
- There is no universal definition of what a cluster is:
  - some algorithms look for instances centered around a particular point, called a **Centeroid**
  - others look for continuous regions of densely packed instances
  - others are hierarchical, looking for clusters of clusters

K-Means

- The **k-means** algorithm is a simple algorithm capable of clustering some kind of dataset very quickly and efficiently, often in just a few iterations.
- **K-means** clusterer tries 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)
```

- Note that you have to specify the no. of clusters **K** that the algorithm must find, it is not that easy to decide
- Each instance is assigned to one of the **K** clusters, in the context of clustering an instance's **label** is the index of the cluster that this instance gets assigned to by the algorithm
- 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
```

```
>>> 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

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

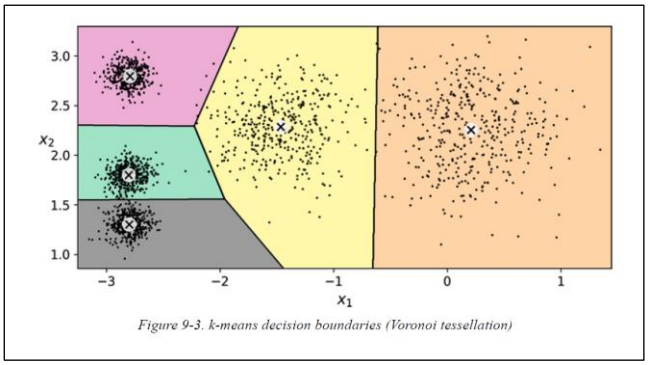


Figure 9-3. k-means decision boundaries (Voronoi tessellation)

- the **k-Means** algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid.
- A few instances were probably mislabeled(especially near the boundary between the top-left cluster and the central cluster)
- Instead of using **Hard clustering**(assigning each instance to a single cluster
- We can use **Soft clustering,** by giving each instance a score per cluster, the score can be the distance between the instance and the centroid; conversely, it can be a similarity score(or **affinity**), such as **Gaussian RBF**
- **K-Means** class uses the **transform()** method to measure the distance from each instance to every centroid

```
>>> kmeans.transform(X_new).round(2)
array([[2.81, 0.33, 2.9 , 1.49, 2.89],
       [5.81, 2.8 , 5.85, 4.48, 5.84],
       [1.21, 3.29, 0.29, 1.69, 1.71],
       [0.73, 3.22, 0.36, 1.55, 1.22]])
```

- If you have a high-dimensional dataset and you transform it this way, you end up with a k-dimensional dataset, this transformation can be a very efficient nonlinear dimensionality reduction technique.
- Alternatively, you can use these distances as extra features to train another model.

The K-Means Algorithm

- How does K-Means Algorithm work?
  - First, you aren't given neither the labels nor the centroids
  - You can Start by placing the centroids randomly (e.g., by picking k instances at random from the dataset and using their locations as centroids)
  - Then label the instances, update the centroids, and so on until the centroids stop moving
  - The algorithm is guaranteed to converge in a finite number of steps(usually quite small)
  - That's because the mean squared distance between the instances and their closest centroids can only go down at each step, and since it cannot be negative, it's guaranteed to converge.
  - The Computational Complexity of K-Means is **O(MKN)**, **m** is no. of instances, **K** is no. of clusters and **n** is no. of dimensions, However if the data doesn't have a clustering structure , the complexity can increase exponentially(NP-hard)
- 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.

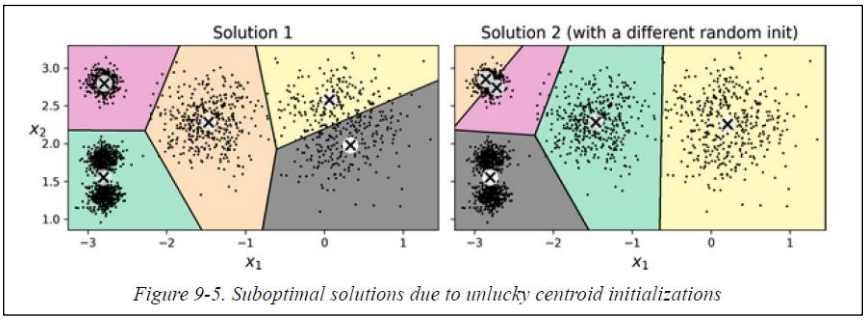


Figure 9-5. Suboptimal solutions due to unlucky centroid initializations

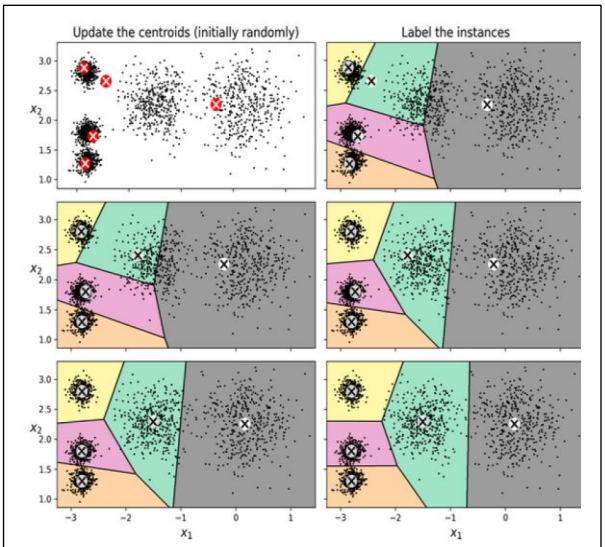


Figure 9-4. The k-means algorithm

Centroid initialization methods

- How to improve the centroid initialization and mitigate the risk of getting bag initializations?
  1. If you know approximately where the centroids should be(running a clustering algorithm earlier), you can set the **init** hyperparameter to a **Numpy** array containing the list of centroids and set **n_init = 1**

```
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, random_state=42)
kmeans.fit(X)
```

  2. Another solution is to run the algorithm multiple times with different random initializations and keep the best solution.
     a. The number of random initializations is controlled by the **n_init** hyperparameter(default = 10)
     b. When you call **fit(),** the algorithms runs **n_init** times, and scikit-learn keeps the best solution

c. To pick the best solution, it uses a performance metric called **inertia,** which is the mean squared distance between each instance and its closest centroid, the algorithm keeps the model with the lowest **inertia**
d. You can access model's **inertia** via **inertia_** attribute
e. **Score()** methods returns the negative **inertia** "greater is better"

3. Using **K-Means++,** They introduced a smarter initialization step that tends to select centroids that are distant from one another, and it makes the k-means algorithm much less likely to converge to a suboptimal solution. that the additional computation required for the smarter initialization step is well worth it because it makes it possible to drastically reduce the number of times the algorithm needs to be run to find the optimal solution.
    a. Take one centroid $\mathbf{c}^{(1)}$, chosen uniformly at random from the dataset.
    b. Take a new centroid $\mathbf{c}^{(i)}$, choosing an instance $\mathbf{x}^{(i)}$ with probability $D(\mathbf{x}^{(i)})^2 / \sum_{j=1}^{m} D(\mathbf{x}^{(j)})^2$, where $\mathbf{D(x^{(i)})}$ is the distance between the instance $\mathbf{x}^{(i)}$ and the closest centroid that was already chosen.
    c. This probability distribution ensures that instances farther away from already chosen centroids are much more likely to be selected as centroids.
    d. Repeat the previous step until all **k** centroids have been chosen.

- The K-Means uses this initialization method by default.

Accelerated k-means and mini-batch k-means

1. Accelerated k-means
    a. By setting **algorithm = "elkan",** On some large datasets with many clusters, the algorithm can be accelerated by avoiding many unnecessary distance calculations.
    b. This is achieved by exploiting the triangle inequality (i.e., that a straight line is always the shortest distance between two points) and by keeping track of lower and upper bounds for distances between instances and centroids.
    c. **Elkan's** algorithm does not always accelerate training, and sometimes it can even slow down training significantly; it depends on the dataset.
    d. Also, it is memory intensive
2. mini-batch k-means
    a. 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.
    b. This speeds up the algorithm (typically by a factor of three to four) and makes it possible to cluster huge datasets that do not fit in memory.
    c. Scikit-Learn uses **MiniBatchKMeans** class, which you can use just like the Kmeans class

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5, random_state=42)
minibatch_kmeans.fit(X)
```

    d. If the dataset does not fit in memory, you can use the **memmap** class
    e. Alternatively, you can pass one mini-batch at a time to the **partial_fit()** method, but this will require much more work, since you will need to perform multiple initializations and select the best one yourself.

- Although the mini-batch k-means algorithm is much faster than the regular k-means algorithm, its inertia is generally slightly worse.

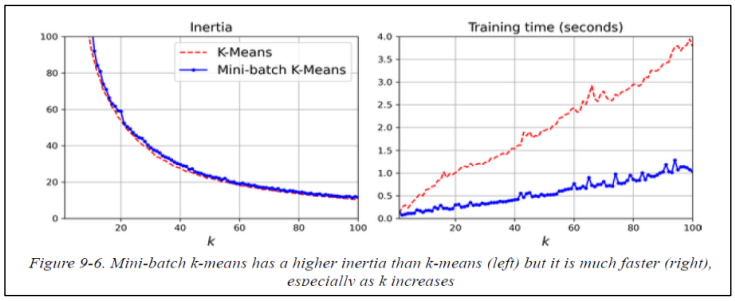


Figure 9-6. Mini-batch k-means has a higher inertia than k-means (left) but it is much faster (right), especially as k increases

Finding the optimal number of cluster

- It is not easy to know how to set the number of clusters **K**
- The inertia is not a good performance metric when trying to choose k because it keeps getting lower as we increase **k**.

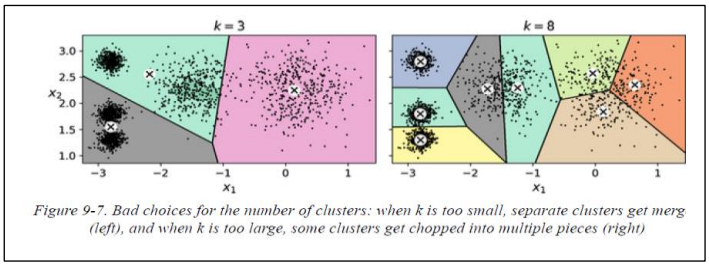


Figure 9-7. Bad choices for the number of clusters: when k is too small, separate clusters get merg (left), and when k is too large, some clusters get chopped into multiple pieces (right)

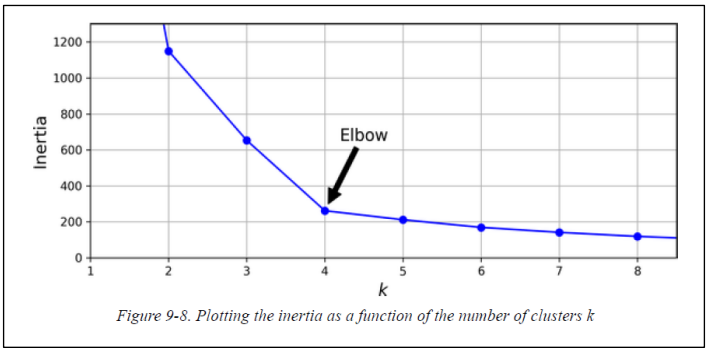


Figure 9-8. Plotting the inertia as a function of the number of clusters k

- A more precise (but also more computationally expensive) approach is to use the **silhouette score**, which is the mean silhouette coefficient over all the instances.
- An instance's silhouette coefficient is equal to **(b – a) / max(a, b),** where **a** is the mean distance to the other instances in the same cluster (i.e., the mean intra-cluster distance) and **b** is the mean nearest-cluster distance (i.e., the mean distance to the instances of the next closest cluster, defined as the one that minimizes **b**, excluding the instance's own cluster).
- The silhouette coefficient can vary between **–1** and **+1**. A coefficient close to **+1** means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to **0** means that it is close to a cluster boundary; finally, a coefficient close to **–1** means that the instance may have been assigned to the wrong cluster.
- To compute the silhouette score, you can use Scikit-Learn's **silhouette_score()** function

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

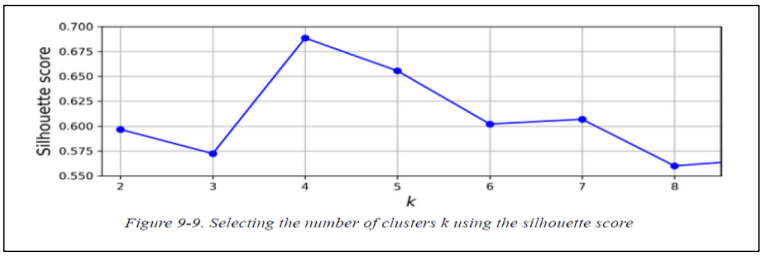


Figure 9-9. Selecting the number of clusters k using the silhouette score

- An even more informative visualization is obtained when we plot every instance's **silhouette coefficient**, sorted by the clusters they are assigned to and by the value of the coefficient. This is called a **silhouette diagram**
- Each diagram contains one knife shape per cluster. The shape's height indicates the number of instances in the cluster, and its width represents the sorted silhouette coefficients of the instances in the cluster (wider is better). The vertical dashed lines represent the mean silhouette score for each number of clusters.
- 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.

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.

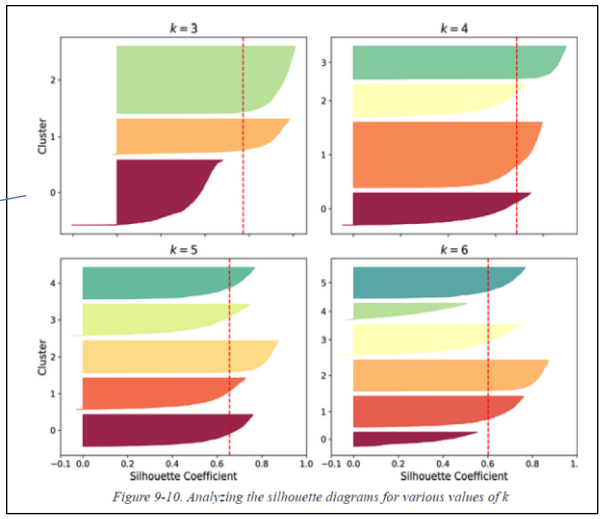


Figure 9-10. Analyzing the silhouette diagrams for various values of k

Limits of K-Means

- Despite its many merits, most notably being fast and scalable, k-means is not perfect.
- it is necessary to run the algorithm several times to avoid suboptimal solutions, plus you need to specify the number of clusters.
- k-means does not behave very well when the clusters have varying sizes, different densities, or nonspherical shapes.
- Fig 9-11, neither of these solutions is any good. The solution on the left is better, but it still chops off 25% of the middle cluster and assigns it to the cluster on the right.
- The solution on the right is just terrible, even though its inertia is lower.
- It is important to scale the input features before you run k-means, or the clusters may be very stretched, and k-means will perform poorly.

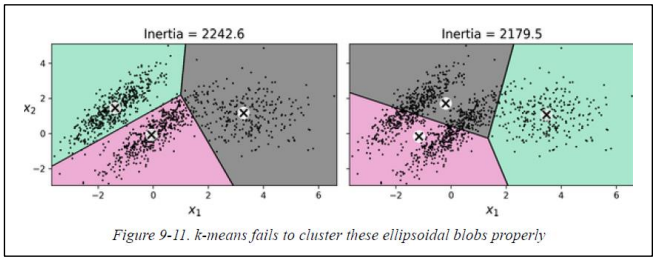


Figure 9-11. k-means fails to cluster these ellipsoidal blobs properly

Using Clustering for Image Segmentation

- **Image segmentation** is the task of partitioning an image into multiple segments. There are several variants:
  - **color segmentation**, pixels with a similar color get assigned to the same segment.
  - **semantic segmentation**, all pixels that are part of the same object type get assigned to the same segment.(one segment containing all the “pedestrians”)
  - **instance segmentation**, all pixels that are part of the same individual object are assigned to the same segment.(a different segment for each pedestrian)
- how to do **Color Segmentation** with K-Means?
  - First, let’s load the image using Matplotlib’s **imread()** function

```
>>> from matplotlib.image import imread  # you could also use `imageio.imread()`
>>> image = imread(os.path.join("images","clustering","ladybug.png"))
>>> image.shape
(533, 800, 3)
```

  - The image is represented as a 3D array: the first dimension’s size is the height, the second is the width, and the third is the number of color channels, in this case red, green, and blue (RGB)
  - reshape the array to get a long list of RGB colors, then cluster these colors using K-Means with eight clusters.
  - It creates a **segmented_img** array containing the nearest cluster center for each pixel
  - lastly it reshapes this array to the original image shape.

```
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)
```

When you use less than 8 clusters, notice that the ladybug’s flashy red color fails to get a cluster of its own: it gets merged with colors from the environment. This is due to the fact that the ladybug is quite small, much smaller than the rest of the image, so even though its color is flashy, K-Means fails to dedicate a cluster to it: as mentioned earlier, K-Means prefers clusters of similar sizes.

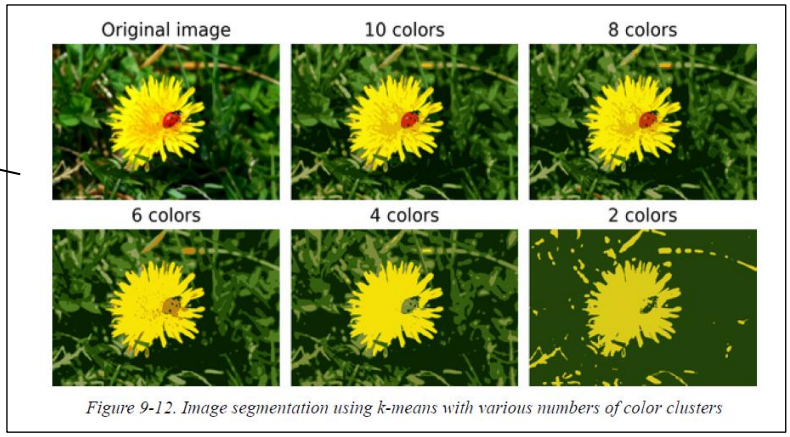


Figure 9-12. Image segmentation using k-means with various numbers of color clusters

Using Clustering for Preprocessing

- Clustering can be an efficient approach to dimensionality reduction, in particular as a preprocessing step before a supervised learning algorithm.

```
from sklearn.datasets import load_digits

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

Now, let's 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, let's fit a Logistic Regression model:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)
```

Let's evaluate its accuracy on the test set:

```
>>> log_reg.score(X_test, y_test)
0.9666666666666667
```

```
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)
0.9822222222222222
```

- We almost divided the error rate by a factor of 2!
- Although it is tempting to define the number of clusters to 10, since there are 10 different digits, it is unlikely to perform well, because there are several different ways to write each digit.
- 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.

```
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 best value for *k*, and the performance of the resulting pipeline:

```
>>> grid_clf.best_params_
{'kmeans__n_clusters': 90}
>>> grid_clf.score(X_test, y_test)
0.9844444444444445
```

Using Clustering for Semi-Supervised Learning

- A use case for clustering in semi-supervised learning when we have plenty of unlabeled instances and very few labeled instances.

train a logistic regression model on a sample of 50 labeled instances

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

What is the performance of this model on the test set?

```
>>> log_reg.score(X_test, y_test)
0.8266666666666667
```

- it is a good idea to label representative instances rather than just random instances, since it is often costly and painful to label instances
- What would happen if we used **label propagation,** propagating the labels to all other instances in the same cluster?

```
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)
0.9288888888888889
```

cluster the training set into 50 clusters, then for each cluster let's find the image closest to the centroid. (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]
```

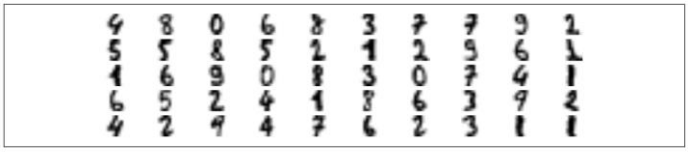


Figure 9-13. Fifty representative digit images (one per cluster)

Now 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])
```

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.9244444444444444
```

- We got a tiny little accuracy boost, 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.
- What would happen if we only propagate the labels to 20% of the instances that are close to the centroids?

```
percentile_closest = 20

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

Now let's train the model again on this partially propagated dataset:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.9422222222222222
```

  - the propagated labels are actually pretty good, their accuracy is very close to 99%,

```
>>> np.mean(y_train_partially_propagated == y_train[partially_propagated])
0.9896907216494846
```

- **Active Learning,** is When a human expert interacts with the learning algorithm, providing labels when the algorithm needs them.
  - There are many different strategies for active learning, but one of the most common ones is called **uncertainty sampling**:
    - The model is trained on the labeled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
    - The instances for which the model is most uncertain (i.e., when its estimated probability is lowest) must be labeled by the expert.
    - Then you just iterate this process again and again, until the performance improvement stops being worth the labeling effort.
  - Other strategies include labeling the instances that would result in the largest model change, or the largest drop in the model's validation error, or the instances that different models disagree on.

DBSCAN

- **DBSCAN is** a popular clustering algorithm that illustrates a different approach based on local density estimation. This approach allows the algorithm to identify clusters of arbitrary shapes.
- This algorithm defines clusters as continuous regions of high density.
  - 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 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**.
- This algorithm works well if all the clusters are dense enough, and they are well separated by low-density regions.
- Scikit-Learn uses **DBSCAN** class

```
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**, this 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.
- 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]])
```

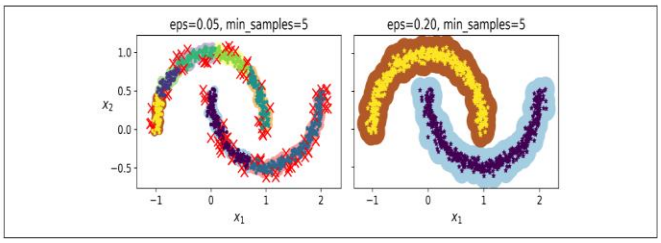


Figure 9-14. DBSCAN clustering using two different neighborhood radiuses

- In Fig 9-14, on the left; it identified quite a lot of anomalies, plus 7 different clusters.
- If we widen each instance's neighborhood by increasing eps to 0.2, we get the clustering on the right, which looks perfect.
- **DBSCAN** class does not have a **predict()** method, although it has a **fit_predict()** method.
- The rationale for this decision is that several classification algorithms could make sense here, and it is easy enough to train one, like using **KNeighborsClassifier.**

```
from sklearn.neighbors import KNeighborsClassifier

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

```
>>> 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 them on the core instances, but we could also have chosen to train them on all the instances, or all but the anomalies: this choice depends on the final task.
- Notice that since there is no anomaly in the KNN's training set, the classifier always chooses a cluster, even when that cluster is far away.
- We use the **kneighbors()** method of the **KneighborsClassifier** to introduce a maximum distance, in which case the instances that are far away from both clusters are classified as anomalies.
- It takes a set of instances, and it returns the distances and the indices of the k-nearest neighbors in the training set

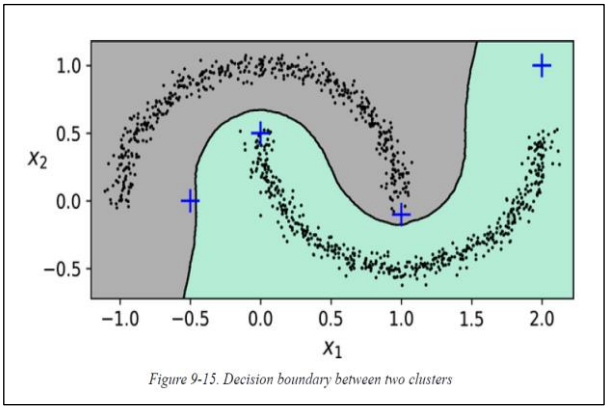


Figure 9-15. Decision boundary between two clusters

```
>>> 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])
```

- DBSCAN is a very 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**).
- DBSCAN can struggle to capture all the clusters properly If the density varies significantly across the clusters, or if there's no sufficiently low-density region around some clusters
- its computational complexity is roughly $\mathbf{O(m^2n)}$, so it does not scale well to large datasets.
- **Hierarchical DBSCAN (HDBSCAN**), is usually better than DBSCAN at finding clusters of varying densities.

Other Clustering Algorithms

- **Agglomerative clustering**
  - A hierarchy of clusters is built from the bottom up, at each iteration, **agglomerative clustering** connects the nearest pair of clusters (starting with individual instances)
  - It can scale nicely to large numbers of instances if you provide a connectivity matrix, which is a sparse m × m matrix that indicates which pairs of instances are neighbors (e.g., returned by **sklearn.neighbors.kneighbors_graph()**).
  - Without a connectivity matrix, the algorithm does not scale well to large datasets.
- **BIRCH**
  - **BIRCH** algorithm was designed specifically for very large datasets, and it can be faster than batch k-means, with similar results, as long as the number of features is not too large (<20).
- **Mean-Shift**
  - This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean.
  - **Mean-shift** shifts the circles in the direction of higher density, until each of them has found a local density maximum. Finally, all the instances whose circles have settled in the same place (or close enough) are assigned to the same cluster.
  - Computational complexity is $\mathbf{O(m^2)}$, not suited for large datasets
- **Affinity propagation**
  - This algorithm uses a voting system, Where instances vote for similar instances to be their representatives, and one the algorithm converges, each representative and its voters form a cluster.
  - Computational complexity is $\mathbf{O(m^2)}$, not suited for large datasets
- **Spectral Clustering**
  - This algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (i.e., it reduces the matrix's dimensionality), then it uses another clustering algorithm in this low-dimensional space
  - It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

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.
- 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.
- There are several GMM variants. The one implemented in **GaussianMixture** class; you must know in advance the number k of Gaussian distributions.
- How is the dataset is assumed to have been generated?
  - The Dataset **X** is assumed to be generated through this probabilistic process
    1. For each instance, a cluster is picked randomly from among **K** clusters
    2. The probability of choosing the $\mathbf{j^{th}}$ cluster is define by the cluster's weight $\boldsymbol{\varphi}^{(i)}$

3. The index of the cluster chosen for the **ith** instance is noted **$z^{(i)}$**
4. if **$z^{(i)}$ = j**, it means **ith** instance has been assigned to **jth** cluster
5. the location **$x^{(i)}$** of this instance is sampled randomly from the gaussian distribution with **mean $\mu^{(i)}$** and **covariance matrix $\Sigma^{(j)}$** . This is noted **$x \sim \mathcal{N}(\mu^{(j)}, \Sigma^{(j)})$.**

- Here is how to interpret this Generative process:
  - The **circles** represent random variables.
  - The **squares** represent fixed values (i.e., parameters of the model).
  - The large rectangles are called **plates**: they indicate that their content is repeated several times.
  - The number indicated at the bottom right-hand side of each plate indicates how many times its content is repeated
  - Each variable **$z^{(i)}$** is drawn from the **categorical distribution** with weights **φ**.

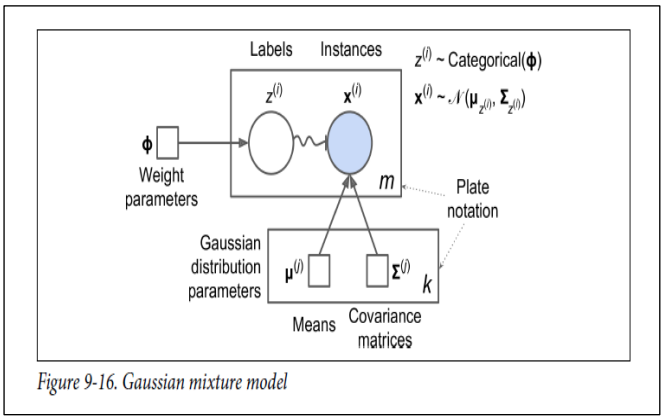


Figure 9-16. Gaussian mixture model

  - Each variable **$x^{(i)}$** is drawn from the **normal distribution** with the **mean** and **covariance** matrix defined by its cluster **$z^{(i)}$.**
  - The **solid arrows** represent conditional dependencies, also when an arrow crosses a plate boundary, it means that it applies to all the repetitions of that plate
  - The **squiggly arrow** from **$z^{(i)}$** to **$x^{(i)}$** represents a switch: depending on the value of **$z^{(i)}$,** the instance **$x^{(i)}$** will be sampled from a different Gaussian distribution. if **$z^{(i)}$=j,** then **$x^{(i)}$ $\mathcal{N}(\mu^{(j)}, \Sigma^{(j)})$.**
  - **Shaded nodes** indicate that the value is known, so in this case only the random variables **$x^{(i)}$** have known values: they are called **observed variables**. The unknown random variables **$z^{(i)}$** are called **latent variables**.
- What can you do with such a model?
  - 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 **$\Sigma^{(k)}$**.

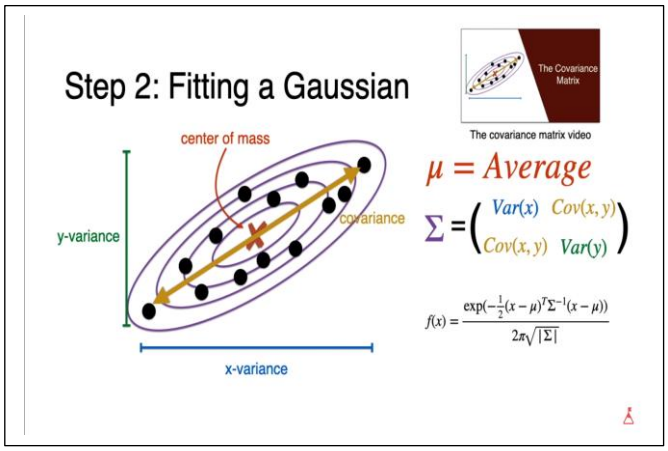
```
from sklearn.mixture import GaussianMixture

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

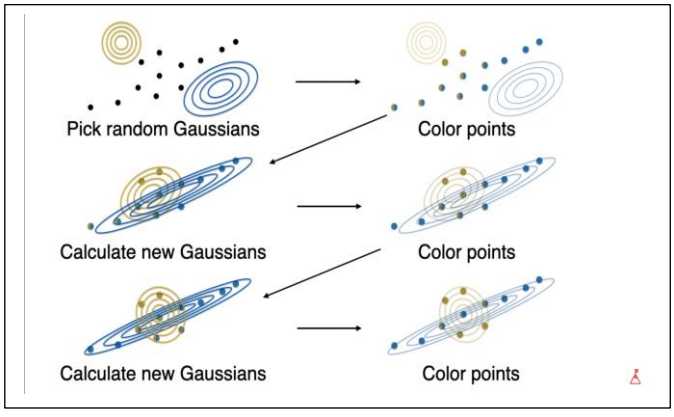
Let's look at the parameters that the algorithm estimated:

```
>>> 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 ]],
```



- How the **GaussianMixture** class works?
  - This class relies on the **Expectation-Maximization (EM)** algorithm
  - First it initializes the cluster parameters randomly, then it repeats two steps until convergence
    - first assigning instances to clusters (this is called the **expectation** step)
    - then updating the clusters (this is called the **maximization** step).



- in the context of clustering, you can think of **EM** as a generalization of **Kmeans** which not only finds the cluster centers ($\mu^{(1)}$ **to** $\mu^{(k)}$), but also their size, shape, and orientation ($\Sigma^{(1)}$ **to** $\Sigma^{(k)}$), as well as their relative weights ($\varphi^{(1)}$ **to** $\varphi^{(k)}$)
- **EM** uses **soft** cluster assignments rather than **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 weighed 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.

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

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

- Now 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**).

```
>>> gm.predict(X)
array([0, 0, 1, ..., 2, 2, 2])
>>> gm.predict_proba(X).round(3)
array([[0.977, 0.   , 0.023],
       [0.983, 0.001, 0.016],
       [0.   , 1.   , 0.   ],
       ...,
       [0.   , 0.   , 1.   ],
       [0.   , 0.   , 1.   ],
       [0.   , 0.   , 1.   ]])
```

- A Gaussian mixture model is a **generative model**, meaning you can sample new instances from it.

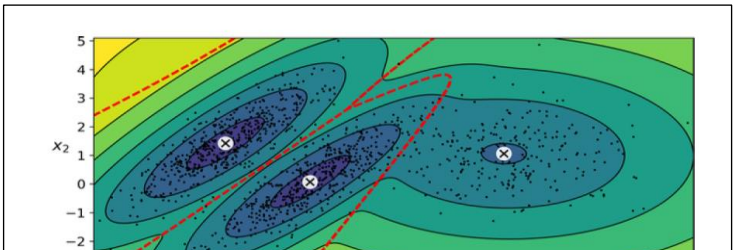
```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([[-0.86944074, -0.32767626],
       [ 0.29836051,  0.28297011],
       [-2.8014927 , -0.09047309],
       [ 3.98203732,  1.49951491],
       [ 3.81677148,  0.53095244],
       [ 2.84104923, -0.73858639]])
>>> y_new
array([0, 0, 1, 2, 2, 2])
```

They are ordered by cluster index

- It is 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).round(2)
array([-2.61, -3.57, -3.33, ..., -3.51, -4.4 , -3.81])
```

- What is the challenges you need to deal with?



- unfortunately, real-life data is not always so Gaussian and low-dimensional
- When there are many dimensions, or many clusters, or a few instances, **EM** can struggle to converge to the optimal solution.
- You may need to reduce the difficulty of the task by limiting the no. of parameters the algorithm has to learn.
- One way to do this is to limit the range of shapes and orientations that the clusters can have, 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).
  - "**full**" (default)
    - Each cluster can take on any shape, size, and orientation (it has its own unconstrained covariance matrix)

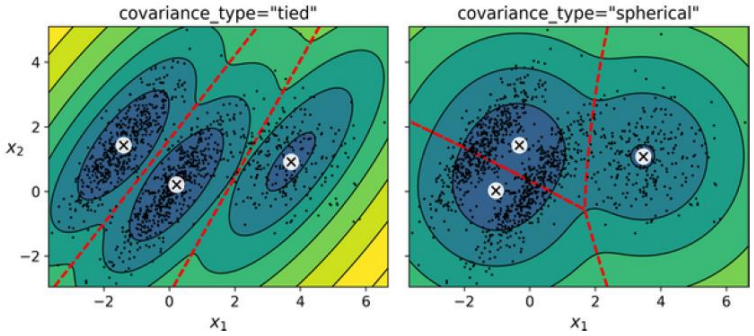


Figure 9-17. Gaussian mixtures for tied clusters (left) and spherical clusters (right)

- 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**, and 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 **$O(kmn^2 + kn^3)$**

Anomaly Detection Using Gaussian Mixtures

- **Anomaly detection** (also called **outlier detection**) is the task of detecting instances that deviate strongly from the norm. These instances are of course called **anomalies** or **outliers**, while the normal instances are called **inliers**.
- How to apply anomaly detections using gaussian mixtures?
  - any instance located in a low-density region can be considered an anomaly.
  - You must define what density threshold you want to use.

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

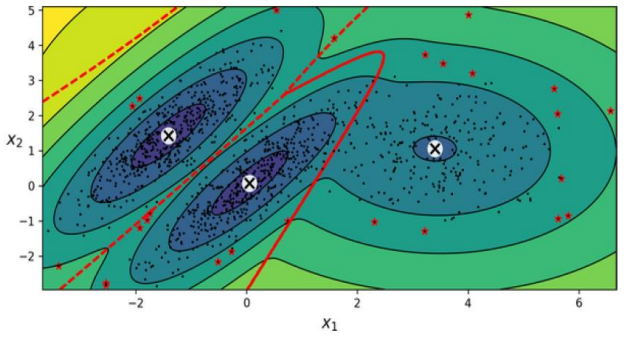


Figure 9-18. Anomaly detection using a Gaussian mixture model

- What if the dataset has too many outliers?
  - **Gaussian mixture** models try to fit all the data, including the **outliers**; if you have too many of them this will bias the model's view of "**normality**", and some outliers may wrongly be considered as normal.
  - If this happens, you can try to fit the model once, use it to detect and remove the most extreme outliers, then fit the model again on the cleaned-up dataset
  - Another approach is to use robust covariance estimation methods like **EllipticEnvelope** class
- A closely related task is **novelty detection**: it differs from anomaly detection in that the algorithm is assumed to be trained on a "clean" dataset, uncontaminated by outliers, whereas anomaly detection does not make this assumption.

Selecting the Number of Clusters

- To select the right no. of clusters for a gaussian mixtures, 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)**

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

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

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

- 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 (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)
8189.747000497186
>>> gm.aic(X)
8102.521720382148
```

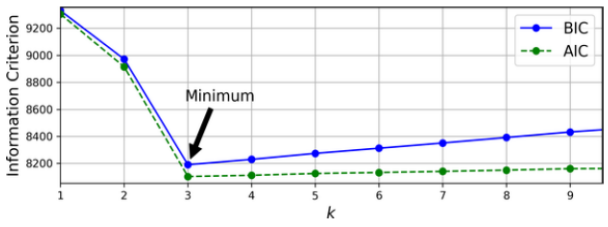
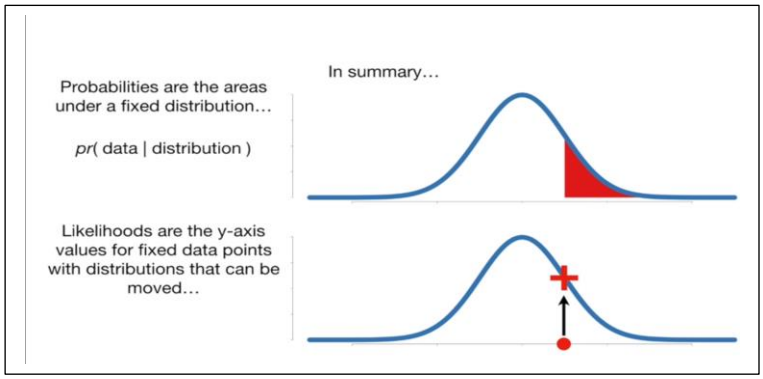


Figure 9-20. AIC and BIC for different numbers of clusters k

Likelihood Function

- The terms "**probability**" and "**likelihood**" have very different meanings in statistics.
- 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**)
- the word "**likelihood**" is used to describe how plausible a particular set of parameter values **θ** are, after the outcome **x** is known.



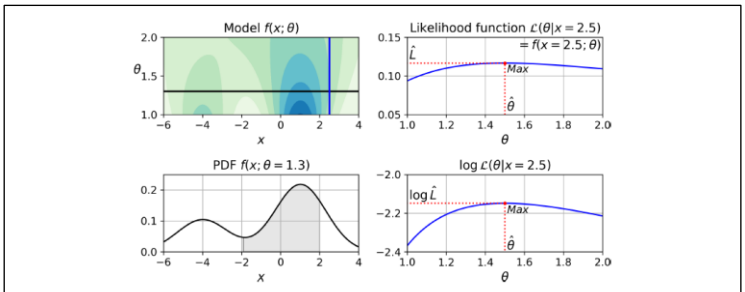


Figure 9-20. A model's parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right) and a log likelihood function (lower right)

- the **PDF** is a function of **x** (with **θ** fixed) while the **likelihood** function is a function of **θ** (with **x** fixed)
- **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.
- **maximum likelihood estimate (MLE),** Given a dataset X, a common task is to try to estimate the most likely values for the model parameters.
- **maximum a-posteriori (MAP)** estimation, Given a prior probability distribution g over **θ** exists, it is possible to take it into account by maximizing $\mathcal{L}(\theta \mid x)\, g(\theta)$ rather than just maximizing $\mathcal{L}(\theta \mid x)$.
- **MAP** constrains the parameter values; you can think of it as a regularized version of **MLE**.
- maximizing the **likelihood** function is equivalent to maximizing its **logarithm,** it is generally easier to maximize the **log likelihood**.
- it is equivalent, and much simpler, to maximize the sum (not the product) of the log likelihood functions, thanks to the magic of the logarithm which converts products into sums: **log(ab)=log(a)+log(b).**

Bayesian Gaussian Mixture Models

- **BayesianGaussianMixture class** is capable of giving weights equal (or close) to zero to unnecessary clusters, Rather than manually searching for the optimal number of clusters
- Just 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)

```
>>> from sklearn.mixture import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10, random_state=42)
>>> bgm.fit(X)
>>> np.round(bgm.weights_, 2)
array([0.4 , 0.21, 0.4 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

- the algorithm automatically detected that only 3 clusters are needed
- In this model, the cluster parameters (including the weights, means and covariance matrices) are not treated as fixed model parameters anymore, but as latent random variables.
- **z** now includes both the cluster parameters and the cluster assignments.
- **The Stick-Breaking process (SBP)** uses **Beta distribution**(model random values whose values lie within a fixed range)
- If the concentration **α** is high, then **ϕ** values will likely be close to **0** and the **SBP** generate many clusters

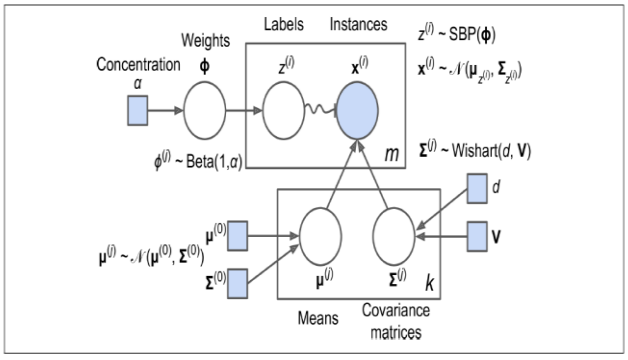


Figure 9-22. Bayesian Gaussian mixture model

- If the concentration **α** is low, then **ϕ** values will likely be close to **1** and the **SBP** generate few clusters
- The **Wishart distribution** is used to sample covariance matrices: the parameters **d** and **V** control the distribution of cluster shapes
- **weight_concentration_prior** hyperparameter, controls the prior belief about the no. of clusters

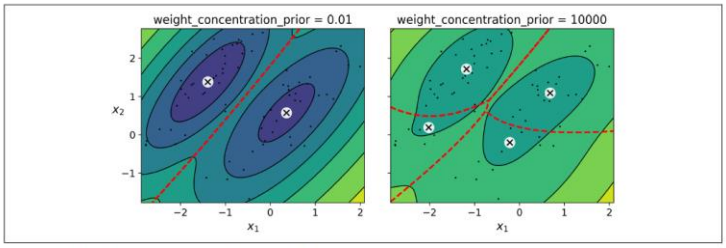
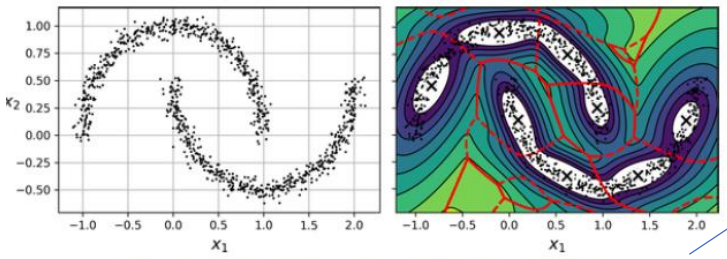


Figure 9-23. Using different concentration priors

- **Gaussian mixture models** work great on clusters with ellipsoidal shapes, they don't do so well with clusters of very different shapes.



using a Bayesian Gaussian mixture model to cluster the moons dataset

Figure 9-21. 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

Other Algorithms for Anomaly and Novelty Detection

- **PCA** (any dimensionality reduction technique with **inverse_transform()** method)
  - By comparing the reconstruction error of a normal instance with the reconstruction error of an anomaly, the latter will usually be much larger.
- **Fast-MCD (Minimum Covariance determinant)**
  - Implemented by the **EllipticEnvelope** class , for outlier detection
  - It assumes that the normal instances (**inliers**) are generated from a single Gaussian distribution (not a mixture).
  - It also assumes that the dataset is contaminated with **outliers** that were not generated from this Gaussian distribution.
  - When the algorithm 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.
- **Isolation Forest**
  - 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 (between the min and max values) 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.
- **Local Outlier Factor (LOF)**
  - Used for outlier detection
  - 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**
  - This algorithm is better suited for novelty detection
  - Operating on one class of instances, the one-class SVM algorithm tries to separate the instances in high-dimensional 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**.
  - You can tweak hyperparameters of a kernelized SVM, plus a **margin** hyperparameter that corresponds to the probability of a new instance being mistakenly considered as novel when it is in fact normal.
  - It works great, especially with high-dimensional datasets, but like all **SVMs** it does not scale to large datasets.

Exercises

- **How would you define clustering? Can you name a few clustering algorithms?**
- Clustering is the unsupervised task of grouping similar instances together, The notion of similarity depends on the task at hand. K-Means, DBSCAN, agglomerative clustering, BIRCH, Mean-Shift, affinity propagation, and spectral clustering.
- **What are some of the main applications of clustering algorithms?**
- Data analysis, customer segmentation, recommender systems, search engines , image segmentation, semi-supervised learning, dimensionality reduction, anomaly detection, and novelty detection.
- **Describe two techniques to select the right number of clusters when using k-means.**
- The elbow rule, Silhouette score
- **What is label propagation? Why would you implement it, and how?**
- Label propagation is a technique that consists in copying some or all of the labels from the labeled set instances to so similar unlabeled instances.
- It can greatly extend the number of labeled instances, and thereby allow a supervised algorithm to reach better performance (semi-supervised learning)
- Using clustering algorithm like k-means on all instances, then for each cluster find the most common label or the label of the most representative instance(the one closest to the centroid), and propagate it to the unlabeled instances in the same cluster
- **Can you name two clustering algorithms that can scale to large datasets? And two that look for regions of high density?**
- K-means and BIRCH scale well to large datasets, DBSCAN and Mean-shift look for regions of high density
- **Can you think of a use case where active learning would be useful? How would you implement it?**
- Active learning is useful whenever you have plenty of unlabeled instances but labeling is costly, in this case rather than randomly selecting instances to label, it is often preferable to perform active learning, where human experts interact with the learning algorithm, providing labels for specific instances when the algorithm requests them , A common approach is **uncertainty sampling**
- **What is the difference between anomaly detection and novelty detection?**
- In anomaly detection, the algorithm is trained on a dataset that may contain outliers, and the goal is typically to identify these outliers(within the training set), as well as outliers among new instances
- In novelty detection, the algorithm is trained on a dataset that is presumed to be “clean” and the objective is to detect novelties strictly among new instances
- One-class SVM works best for novelty detection , isolation forest works best for anomaly detection
- **What is a Gaussian mixture? What tasks can you use it for?**
- 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.
- Density estimation, clustering, and anomaly detection
- **Can you name two techniques to find the right number of clusters when using a Gaussian mixture model?**
- Choose the no. of clusters that minimizes the **BIC or AIC**
- Use **Bayesian Gaussian Mixture model**