Chapter 9: Unsupervised Learning Techniques

- **Dimensionality Reduction** (seen in previous chapter)
- **Clustering**: group similar instances together into clusters
- **Anomaly Detection**: 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 trend in a time series
- **Density Estimation**: estimating the probability density function (PDF) of the random process that generated the dataset. Defines clusters as continous gregions of high density

1. Clustering

- Goal is to group similar instances together into clusters.
- Clustering is a great tool for data analysis, computer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction etc

1.1 Clustering Applications

- **For customer segmentation**
  Cluster customers based on their purchases and their activity on your website. Useful to understand who are your customers are and adapt your products and marketing campaigns to each segment. Can also be useful in recomender systems to suggest content that other uers in the same clusters enjoyed.

- **Data Analysis**
  When analyzing a new dataset, can be helpful to run a clustering algorithm and then analyze each cluster separately.

- **Dimensionality Reduction Technique**
  Once dataset has been clustered, we can measure how well an instance fits into a cluster (affinity).

- **Anomaly Detection (Outlier Detection**)
  Any instance that has a low affinity to all the clusters is likely to be an anomaly. Useful in detecting defects in manufacting or for fraud detection.

- **Semi-Supervised Learning**
  If we only have a few labels, we can perform clustering and propagate the lables to all

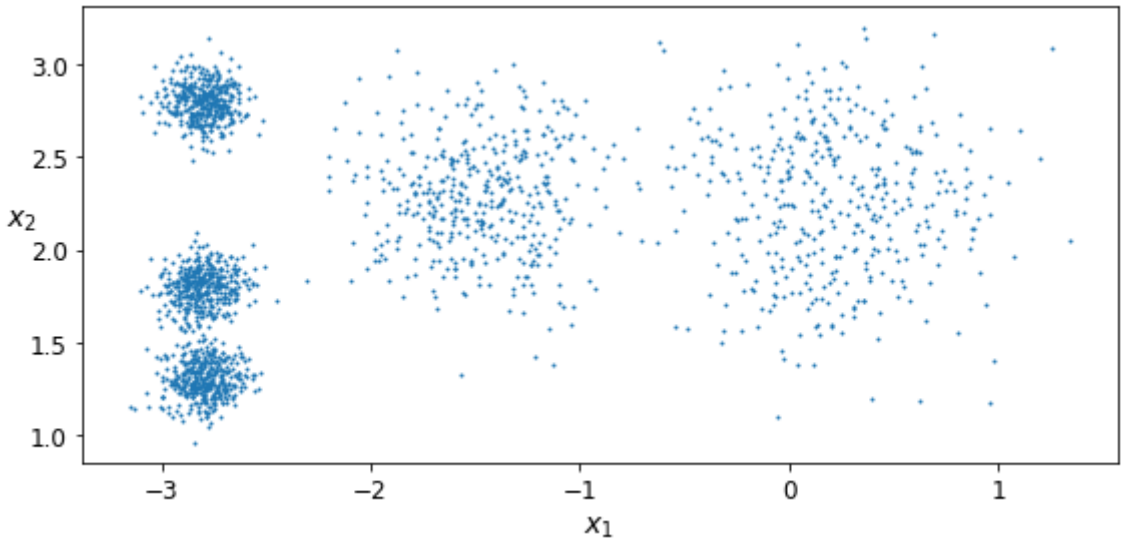
instances in the same cluster. This technique greatly increase the number of labels available for subsequent supervised learning algorithm.

- **Search Engines**
  Search engines let you search for images that are similar to a reference image. We first apply a clustering algorithm to all the images. Similar iamges would end up in same clluster. When a user provide a reference image, we just find the image's cluster and then return all the image from that cluster.

- **Segment an image**
  By clustering pixels according to their color, then replace each pixel's colour with the mean colour of its cluster, we can reduce the number of different colors in the image. Image segmentation is used in many object detection and tracking systems and makes it easier to detect the contour of each object.

1.2 K-Means (Lloyd-Forgt)

- Sometimes referred to as Lloyd-Forgy
- K-Means capable of clustering dataset very quickly and efficiently
- 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.
- It is important to scale the input features before running K-Means, or the clusters will be very stretched and K-Means will perform poorly.
- Scaling features does not guarantee that all clusters will be nice and spherical but it generally improves things.



In [ ]:
```
from sklearn.datasets import make_blobs
import numpy as np

blob_centers = np.array(
    [[ 0.2,  2.3],
     [-1.5 ,  2.3],
     [-2.8,  1.8],
     [-2.8,  2.8],
```

```
    [-2.8,  1.3]])
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])

X, y = make_blobs(n_samples=2000, centers=blob_centers,
                  cluster_std=blob_std, random_state=7)
```

In [ ]:
```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k) # Want K clusters
y_pred = kmeans.fit_predict(X)
```

In [ ]:
```
y_pred
```

Out[ ]:
```
array([4, 0, 1, ..., 2, 1, 0])
```

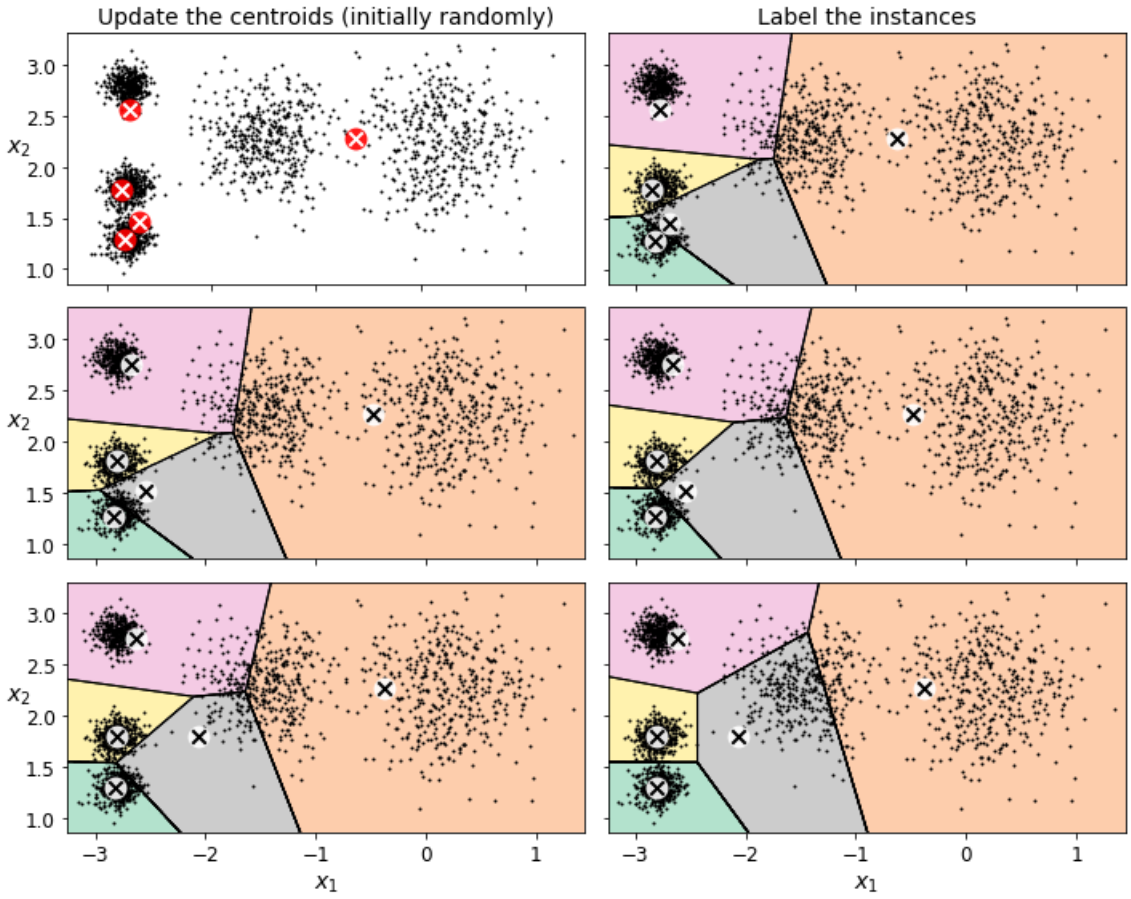
In [ ]:
```
# five centroids that the algorithm found
kmeans.cluster_centers_
```

Out[ ]:
```
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

1.2.1 K-Means Algorithm

- Start by placing centroids randomly (pick k instances at random and using their locations as centroids)
- Label the instances by assigning them to the closest centroid.
- Update the centroid and label the instances again until the centroids stop moving
- The algorithm is guaranteed to converge in a finite number of steps. But it may not be the optimum solution. This depends on the initial centroid initialization.

If data has no clustering structure, the computational complexity may increase exponentially witht he number of instances. If data has clustering structure, the computational complexity is generally linear wrt number of instances, number of clusters and number of dimensions.



1.2.2 Hard Clustering and Soft Clustering

- Hard Clustering - assign each instance to a single cluster
- Soft Clustering - assign each instance a score per cluster.

The score can be the distance between the instance and the centroid. It can also be a similarity score, such as Gaussian Radial Basis Function.

In [ ]:
```
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
```

In [ ]:
```
# Soft Clustering example.
kmeans.transform(X_new)
```

Out[ ]:
```
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]])
```

Example above, first instance is located at a distance of 2.81 from the first centroid, 0.33 from the second centroid, 2.90 from the third centroid and so on. If we have high-dimensional dataset and we transform the data this way, we will have k-dimensional dataset. This transformation can be very efficient nonlinear dimensionality reduction technique.

1.2.3 Improving Centroid Initialization

To mitigate the risk of obtaining suboptimal solution, we can improve centroid initilization.

1. If we know where the centroids should be, we can set the `init` hyperparameter.

In [ ]:
```
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)
```

2. Run the algorithm multiple times with different random initialization and keep the best solution.

- Another solution is to run the algorithm multiple times with different random initialization and keep the best solution.
- n_init hyperparameter controls the number of random initializations
- Scikit-Learn runs 10 times by default and keeps the best solution measured by model's inertia.
- Inertia: the measn squared distance between each instance and its closest centroid.

In [ ]:
```
kmeans.fit(X)
```

Out[ ]:
```
KMeans(init=array([[-3,  3],
       [-3,  2],
       [-3,  1],
       [-1,  2],
       [ 0,  2]]),
       n_clusters=5, n_init=1)
```

In [ ]:
```
kmeans.inertia_
```

Out[ ]:
```
211.59853725816836
```

3. K-Means++

- Another improvement is the K-Means++ algorithm, proposed in 2006.
- 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 additional computation is worth it because it drastically reduce the number of times the algorhtm needs to be run to find the optimal solution

The KMeans class uses this initialization by default. If we want the original method, we can set the `init` hyperparameter to random though we rarely need to do this.

1.2.4 Accelerated K-Means

- Accelerates the algorhtm by avoiding many unnecessary distance calculations.
- This is the algorithm the KMeans class uses by default.
- We can force it to use the original algorithm by setting `algorithm` hyperparameter to `"full"` although we probably will never need to.

1.2.5 Mini-Batch K-Means

- Another variant of K-Means algorithm.
- 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 and possible to cluster huge datasets that do not fit in memory.

In [ ]:
```
from sklearn.cluster import MiniBatchKMeans

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

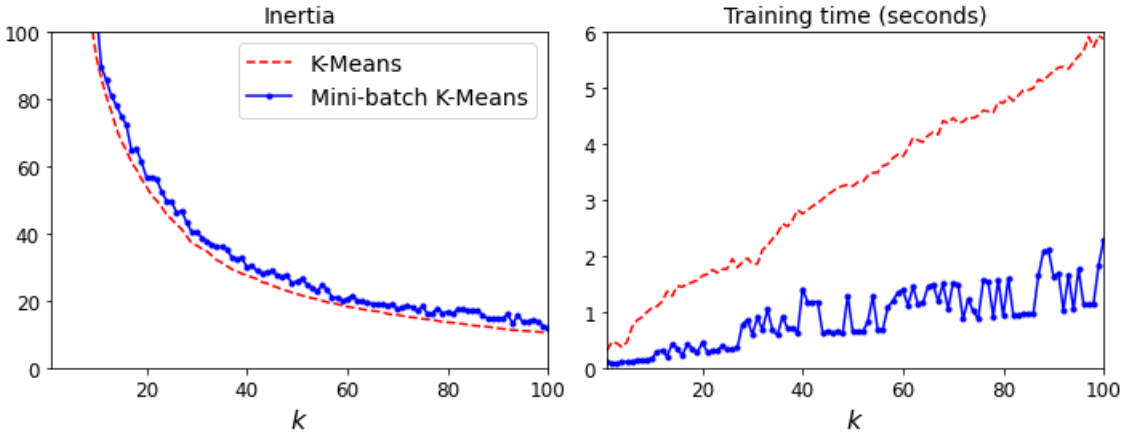
```
c:\Users\joann\Anaconda3\envs\tensorflow-gpu\lib\site-packages\sklearn\cluster\_kmeans.py:1043: UserWarning: MiniBatchKMeans is known to have a memory leak on Windows with MKL, when there are less chunks than available threads. You can prevent it by setting batch_size >= 3072 or by setting the environment variable OMP_NUM_THREADS=4
  warnings.warn(
```

Out[ ]:
```
MiniBatchKMeans(n_clusters=5)
```

In [ ]:
```
minibatch_kmeans.inertia_
```

Out[ ]:
```
224.68584873118633
```

But the inertia is generally slightly worse especially when numher of clusters increase.

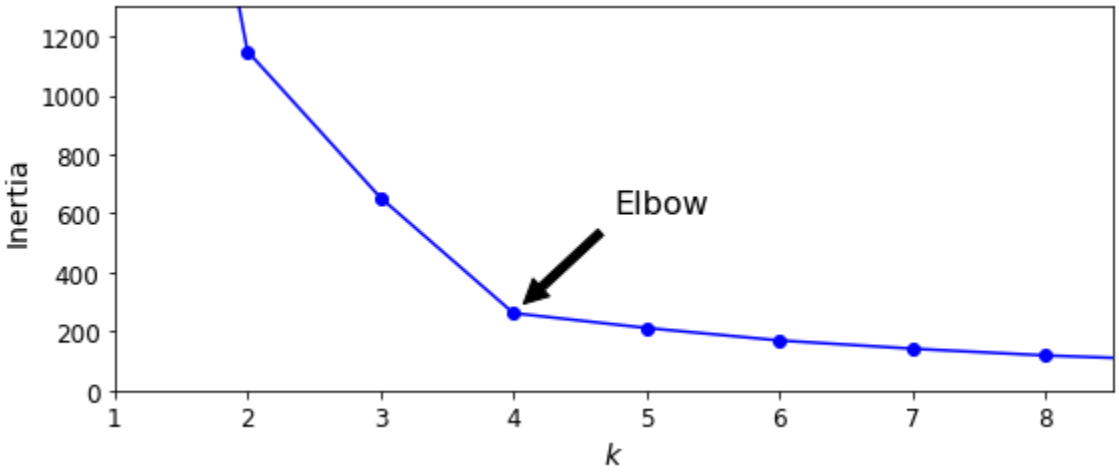


If dataset does not fit in memory, we can use the `memmap` class.

- We can pass one mini-batch at a time to the `partial_fit()` method.

1.2.6 Finding Optimal Number of Clusters using Silhouette Coefficient

- The inertia is not a good performance metric when trying to choose k beacuse it keeps getting lower as we increase k.



- A more precise approach but also more computational expensive is to use the sihouette score, which is the mean sihouette coefficient over all the instances.

Sihouette Score = $\frac{b-a}{max(a,b)}$. Score will be between -1 and +1.

- a is the mean distance to the other instances in the same cluster (i.e mean intra-cluster distance)
- b is the mean nearest-cluster distance (i.e mean distance to the instances of the next closest cluster)

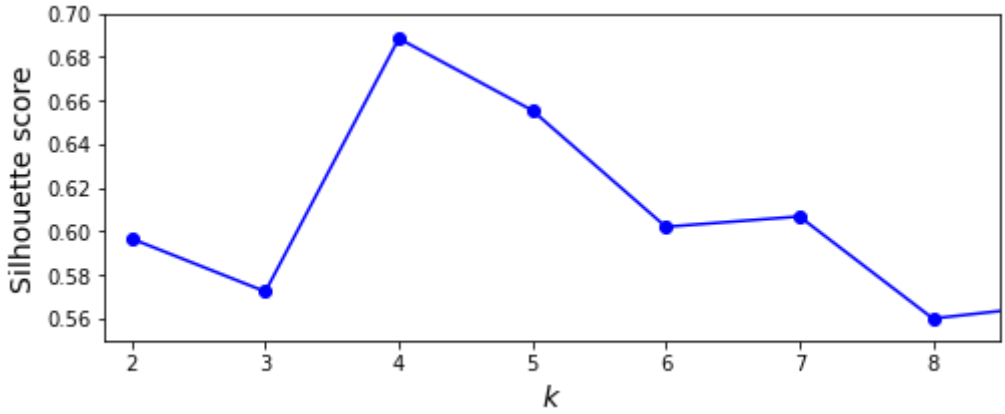
In [ ]:
```
from sklearn.metrics import silhouette_score
silhouette_score(X, kmeans.labels_)
```

Out[ ]:
```
0.655517642572828
```

Coefficient close to +1 means that the intance is well inside its own cluster and far from other clusters.
Coefficient close to 0 means that it is close to a cluster boundary
Coefficient close to -1 means that the instance may have been assigned to the wrong cluster

In [ ]:
```
import matplotlib.pyplot as plt

kmeans_per_k = [KMeans(n_clusters=k, random_state=42).fit(X)
                for k in range(1, 10)]

silhouette_scores = [silhouette_score(X, model.labels_)
                     for model in kmeans_per_k[1:]]

plt.figure(figsize=(8, 3))
plt.plot(range(2, 10), silhouette_scores, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Silhouette score", fontsize=14)
plt.axis([1.8, 8.5, 0.55, 0.7])
plt.show()
```

```
c:\Users\joann\Anaconda3\envs\tensorflow-gpu\lib\site-packages\sklearn\cluster\_km
eans.py:1036: UserWarning: KMeans is known to have a memory leak on Windows with M
KL, when there are less chunks than available threads. You can avoid it by setting
the environment variable OMP_NUM_THREADS=8.
  warnings.warn(
```

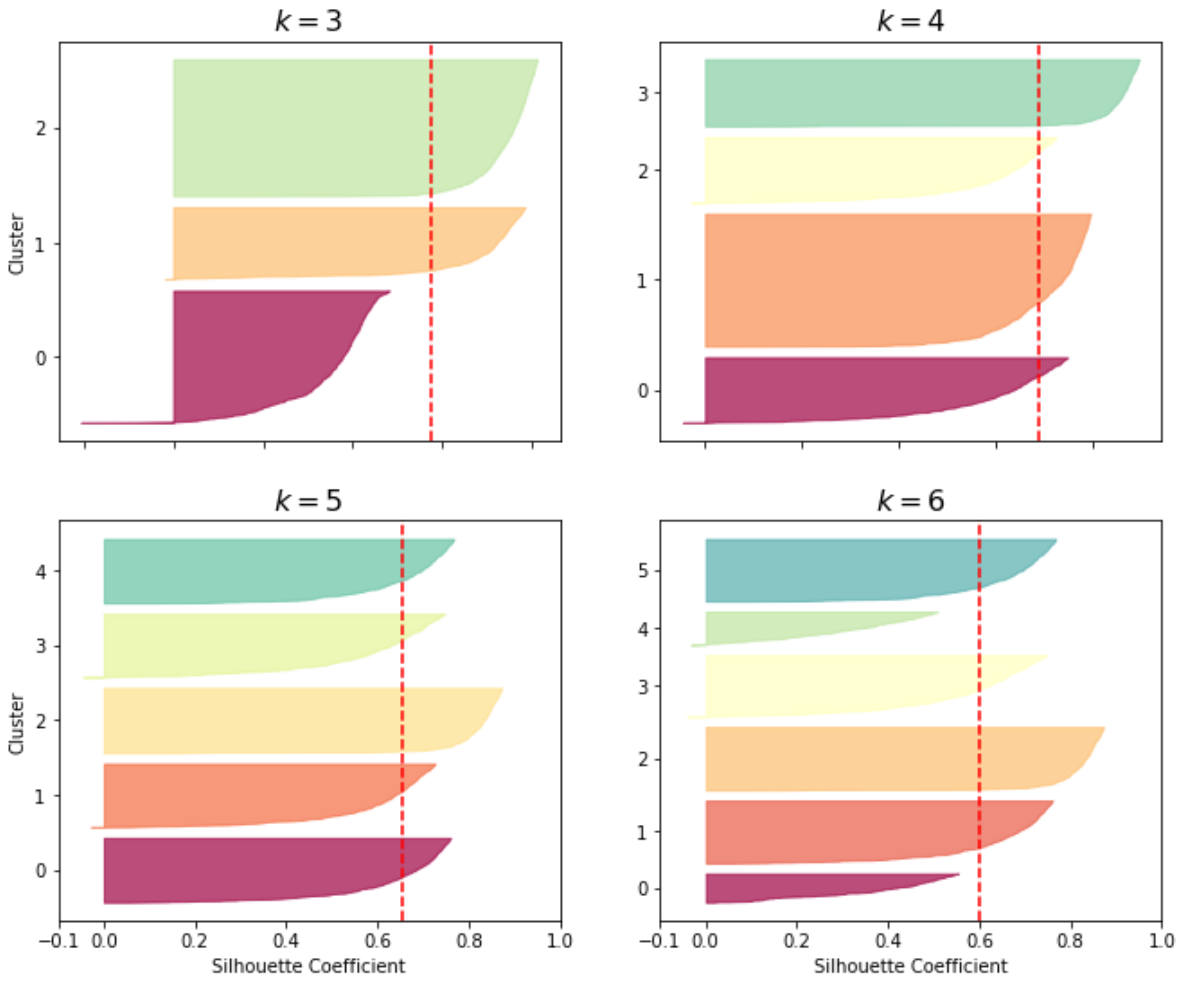


1.2.7 Silhouette Diagram

- An even more informative visualization is obtained when we plot every instance's silhouette coefficient, sorted by the cluster they are assigned to and by the value of the coefficient.
- The shape's height indicates the number of instances the cluster contains.
- The shape's width represents the sorted silhouette coefficients of the instances in the cluster. Wider is better.
- The dashed line indicates the mean silhouette coefficient.

In [ ]:
```
from sklearn.metrics import silhouette_samples
from matplotlib.ticker import FixedLocator, FixedFormatter
import matplotlib as mpl

plt.figure(figsize=(11, 9))

for k in (3, 4, 5, 6):
    plt.subplot(2, 2, k - 2)

    y_pred = kmeans_per_k[k - 1].labels_
    silhouette_coefficients = silhouette_samples(X, y_pred)

    padding = len(X) // 30
    pos = padding
    ticks = []
    for i in range(k):
        coeffs = silhouette_coefficients[y_pred == i]
        coeffs.sort()

        color = mpl.cm.Spectral(i / k)
        plt.fill_betweenx(np.arange(pos, pos + len(coeffs)), 0, coeffs,
                          facecolor=color, edgecolor=color, alpha=0.7)
        ticks.append(pos + len(coeffs) // 2)
        pos += len(coeffs) + padding

    plt.gca().yaxis.set_major_locator(FixedLocator(ticks))
    plt.gca().yaxis.set_major_formatter(FixedFormatter(range(k)))
    if k in (3, 5):
        plt.ylabel("Cluster")

    if k in (5, 6):
        plt.gca().set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
        plt.xlabel("Silhouette Coefficient")
    else:
        plt.tick_params(labelbottom=False)
```

```
    plt.axvline(x=silhouette_scores[k - 2], color="red", linestyle="--")
    plt.title("$k={}$".format(k), fontsize=16)
```

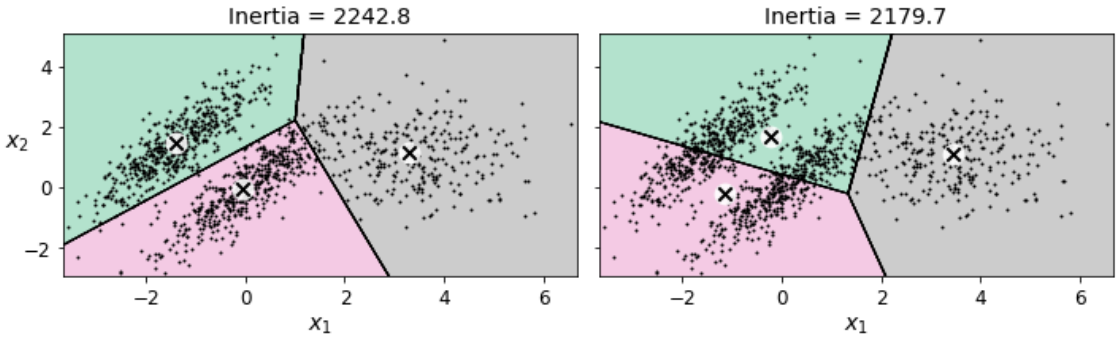


When most of the instances in a cluster have a lower coefficient than the mean silhouette score for that cluster, the cluster is bad since it means its instances are much too close to other clusters.

- k=3 and k=6 are bad clusters.
- k=4 and k=5 are good clusters since most instances extend beyond the dashed line.
- When k=4, the cluster at index 1 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 k=5, k=5 is a better cluster size.

1.2.8 Limitations of K-Means

- Does not behave well when the clusters have varying sizes, different densities, or nonspherical shapes.



- It is important to scale the input features before running K-Means, or the clusters will be very stretched and K-Means will perform poorly.
- Scaling features does not guarantee that all clusters will be nice and spherical but it generally improves things.

1.3 Using Clustering for Image Segmentation

- **Image Segmentation**: partition an image into multiple segments
- **Semantic Segmentation**: all pixels that are part of the same object type get assigned to the same segment
- **Instance Segmentation**: all pixels that are part of the same individual object are assigned to the same segment
- **Color Segmentation**: assign pixels to the same segment if they have similar color.

In [ ]:
```
# Download the ladybug image
import urllib.request

images_path = os.path.join(".", "images", "unsupervised_learning")
os.makedirs(images_path, exist_ok=True)
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
filename = "ladybug.png"
print("Downloading", filename)
url = DOWNLOAD_ROOT + "images/unsupervised_learning/" + filename
urllib.request.urlretrieve(url, os.path.join(images_path, filename))
```

Out[ ]:
```
Downloading ladybug.png
('.\\images\\unsupervised_learning\\ladybug.png',
 <http.client.HTTPMessage at 0x2290709abe0>)
```

In [ ]:
```
from matplotlib.image import imread
image = imread(os.path.join(images_path, filename))
image.shape #(height, width, number of color channels)
```

Out[ ]:
```
(533, 800, 3)
```

There are 3 color channels, in this case RGB. This means that for each pixel, there is a 3D vector containing the intensities of red, green and blue each between 0.0 and 1.0 (or between 0 and 255 if we use `imageio.imread()`)

- For each color, it looks for the mean color of the pixel's color cluster.

- Example: all shades of green may be replaced with the same light green color (assuming the mean color of the green cluster is light green).
- Then it reshapes this long list of colors to get the same shape as the original image.

In [ ]:
```
X = image.reshape(-1, 3) #-1 refers to the inferred value from the length of the ar
kmeans = KMeans(n_clusters=8, random_state=42).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

In [ ]:
```
len(X) # height x width
```

Out[ ]:
```
426400
```

In [ ]:
```
segmented_imgs = []
n_colors = (10, 8, 6, 4, 2)
for n_clusters in n_colors:
    kmeans = KMeans(n_clusters=n_clusters, random_state=42).fit(X)
    segmented_img = kmeans.cluster_centers_[kmeans.labels_]
    segmented_imgs.append(segmented_img.reshape(image.shape))
```

In [ ]:
```
plt.figure(figsize=(10,5))
plt.subplots_adjust(wspace=0.05, hspace=0.1)

plt.subplot(231)
plt.imshow(image)
plt.title("Original image")
plt.axis('off')

for idx, n_clusters in enumerate(n_colors):
    plt.subplot(232 + idx)
    plt.imshow(segmented_imgs[idx])
    plt.title("{} colors".format(n_clusters))
    plt.axis('off')

plt.show()
```

1.4 Using Clustering for Preprocessing

- Clustering can be an efficient approach to dimensionality reduction, in particular as a

preprocessing step before a supervised learning algorithm.

In [ ]:
```
# Load Data
from sklearn.datasets import load_digits
X_digits, y_digits = load_digits(return_X_y=True)

# Split into training and testing set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits, random_sta

# Fit a logisitc regression mode
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(multi_class="ovr", solver="lbfgs", max_iter=5000, ran
log_reg.fit(X_train, y_train)

log_reg.score(X_test, y_test)
```

Out[ ]:
```
0.9688888888888889
```

- First cluster the training set into 50 clusters and replace the images with their distances to these 50 clusters then apply Logistic

In [ ]:
```
# Try adding K-Means as a preprocessing step
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression(multi_class="ovr", solver="lbfgs", max_iter=500
])

pipeline.fit(X_train, y_train)

pipeline.score(X_test, y_test)
```

Out[ ]:
```
0.9777777777777777
```

- Finding a good k value is simpler here because we can just the result of the classification

In [ ]:
```
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)
```

```
Fitting 3 folds for each of 98 candidates, totalling 294 fits
[CV] END ..............................kmeans__n_clusters=2; total time=   0.2s
[CV] END ..............................kmeans__n_clusters=2; total time=   0.2s
[CV] END ..............................kmeans__n_clusters=2; total time=   0.3s
[CV] END ..............................kmeans__n_clusters=3; total time=   0.2s
[CV] END ..............................kmeans__n_clusters=3; total time=   0.2s
[CV] END ..............................kmeans__n_clusters=3; total time=   0.2s
[CV] END ..............................kmeans__n_clusters=4; total time=   0.3s
[CV] END ..............................kmeans__n_clusters=4; total time=   0.3s
[CV] END ..............................kmeans__n_clusters=4; total time=   0.2s
[CV] END ..............................kmeans__n_clusters=5; total time=   0.3s
[CV] END ..............................kmeans__n_clusters=5; total time=   0.4s
[CV] END ..............................kmeans__n_clusters=5; total time=   0.3s
[CV] END ..............................kmeans__n_clusters=6; total time=   0.4s
[CV] END ..............................kmeans__n_clusters=6; total time=   0.4s
[CV] END ..............................kmeans__n_clusters=6; total time=   0.4s
[CV] END ..............................kmeans__n_clusters=7; total time=   0.5s
[CV] END ..............................kmeans__n_clusters=7; total time=   0.5s
[CV] END ..............................kmeans__n_clusters=7; total time=   0.6s
[CV] END ..............................kmeans__n_clusters=8; total time=   0.8s
[CV] END ..............................kmeans__n_clusters=8; total time=   0.7s
[CV] END ..............................kmeans__n_clusters=8; total time=   0.7s
[CV] END ..............................kmeans__n_clusters=9; total time=   0.8s
[CV] END ..............................kmeans__n_clusters=9; total time=   0.7s
[CV] END ..............................kmeans__n_clusters=9; total time=   0.7s
[CV] END .............................kmeans__n_clusters=10; total time=   0.9s
[CV] END .............................kmeans__n_clusters=10; total time=   0.9s
[CV] END .............................kmeans__n_clusters=10; total time=   0.9s
[CV] END .............................kmeans__n_clusters=11; total time=   1.1s
[CV] END .............................kmeans__n_clusters=11; total time=   1.1s
[CV] END .............................kmeans__n_clusters=11; total time=   1.2s
[CV] END .............................kmeans__n_clusters=12; total time=   1.3s
[CV] END .............................kmeans__n_clusters=12; total time=   1.5s
[CV] END .............................kmeans__n_clusters=12; total time=   1.3s
[CV] END .............................kmeans__n_clusters=13; total time=   2.1s
[CV] END .............................kmeans__n_clusters=13; total time=   2.3s
[CV] END .............................kmeans__n_clusters=13; total time=   2.4s
[CV] END .............................kmeans__n_clusters=14; total time=   1.9s
[CV] END .............................kmeans__n_clusters=14; total time=   2.1s
[CV] END .............................kmeans__n_clusters=14; total time=   2.1s
[CV] END .............................kmeans__n_clusters=15; total time=   2.3s
[CV] END .............................kmeans__n_clusters=15; total time=   2.3s
[CV] END .............................kmeans__n_clusters=15; total time=   2.2s
[CV] END .............................kmeans__n_clusters=16; total time=   2.5s
[CV] END .............................kmeans__n_clusters=16; total time=   2.6s
[CV] END .............................kmeans__n_clusters=16; total time=   2.4s
[CV] END .............................kmeans__n_clusters=17; total time=   2.8s
[CV] END .............................kmeans__n_clusters=17; total time=   2.6s
[CV] END .............................kmeans__n_clusters=17; total time=   2.6s
[CV] END .............................kmeans__n_clusters=18; total time=   3.1s
[CV] END .............................kmeans__n_clusters=18; total time=   3.0s
[CV] END .............................kmeans__n_clusters=18; total time=   2.7s
[CV] END .............................kmeans__n_clusters=19; total time=   3.3s
[CV] END .............................kmeans__n_clusters=19; total time=   3.2s
[CV] END .............................kmeans__n_clusters=19; total time=   3.2s
[CV] END .............................kmeans__n_clusters=20; total time=   3.4s
[CV] END .............................kmeans__n_clusters=20; total time=   3.0s
[CV] END .............................kmeans__n_clusters=20; total time=   3.1s
[CV] END .............................kmeans__n_clusters=21; total time=   3.4s
[CV] END .............................kmeans__n_clusters=21; total time=   3.3s
[CV] END .............................kmeans__n_clusters=21; total time=   3.3s
[CV] END .............................kmeans__n_clusters=22; total time=   3.7s
[CV] END .............................kmeans__n_clusters=22; total time=   3.6s
[CV] END .............................kmeans__n_clusters=22; total time=   3.8s
```

```
[CV] END ..............................kmeans__n_clusters=23; total time=   3.7s
[CV] END ..............................kmeans__n_clusters=23; total time=   3.5s
[CV] END ..............................kmeans__n_clusters=23; total time=   3.5s
[CV] END ..............................kmeans__n_clusters=24; total time=   3.6s
[CV] END ..............................kmeans__n_clusters=24; total time=   3.9s
[CV] END ..............................kmeans__n_clusters=24; total time=   3.7s
[CV] END ..............................kmeans__n_clusters=25; total time=   3.9s
[CV] END ..............................kmeans__n_clusters=25; total time=   4.0s
[CV] END ..............................kmeans__n_clusters=25; total time=   3.7s
[CV] END ..............................kmeans__n_clusters=26; total time=   3.9s
[CV] END ..............................kmeans__n_clusters=26; total time=   4.1s
[CV] END ..............................kmeans__n_clusters=26; total time=   4.5s
[CV] END ..............................kmeans__n_clusters=27; total time=   4.2s
[CV] END ..............................kmeans__n_clusters=27; total time=   4.4s
[CV] END ..............................kmeans__n_clusters=27; total time=   3.8s
[CV] END ..............................kmeans__n_clusters=28; total time=   4.1s
[CV] END ..............................kmeans__n_clusters=28; total time=   4.7s
[CV] END ..............................kmeans__n_clusters=28; total time=   4.6s
[CV] END ..............................kmeans__n_clusters=29; total time=   4.7s
[CV] END ..............................kmeans__n_clusters=29; total time=   5.1s
[CV] END ..............................kmeans__n_clusters=29; total time=   4.3s
[CV] END ..............................kmeans__n_clusters=30; total time=   4.6s
[CV] END ..............................kmeans__n_clusters=30; total time=   4.6s
[CV] END ..............................kmeans__n_clusters=30; total time=   4.4s
[CV] END ..............................kmeans__n_clusters=31; total time=   4.9s
[CV] END ..............................kmeans__n_clusters=31; total time=   4.4s
[CV] END ..............................kmeans__n_clusters=31; total time=   3.9s
[CV] END ..............................kmeans__n_clusters=32; total time=   5.2s
[CV] END ..............................kmeans__n_clusters=32; total time=   4.5s
[CV] END ..............................kmeans__n_clusters=32; total time=   4.1s
[CV] END ..............................kmeans__n_clusters=33; total time=   4.5s
[CV] END ..............................kmeans__n_clusters=33; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=33; total time=   4.3s
[CV] END ..............................kmeans__n_clusters=34; total time=   4.4s
[CV] END ..............................kmeans__n_clusters=34; total time=   4.6s
[CV] END ..............................kmeans__n_clusters=34; total time=   4.4s
[CV] END ..............................kmeans__n_clusters=35; total time=   4.6s
[CV] END ..............................kmeans__n_clusters=35; total time=   5.0s
[CV] END ..............................kmeans__n_clusters=35; total time=   5.3s
[CV] END ..............................kmeans__n_clusters=36; total time=   4.7s
[CV] END ..............................kmeans__n_clusters=36; total time=   4.5s
[CV] END ..............................kmeans__n_clusters=36; total time=   5.0s
[CV] END ..............................kmeans__n_clusters=37; total time=   5.1s
[CV] END ..............................kmeans__n_clusters=37; total time=   4.6s
[CV] END ..............................kmeans__n_clusters=37; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=38; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=38; total time=   5.3s
[CV] END ..............................kmeans__n_clusters=38; total time=   5.1s
[CV] END ..............................kmeans__n_clusters=39; total time=   5.0s
[CV] END ..............................kmeans__n_clusters=39; total time=   4.8s
[CV] END ..............................kmeans__n_clusters=39; total time=   4.7s
[CV] END ..............................kmeans__n_clusters=40; total time=   4.6s
[CV] END ..............................kmeans__n_clusters=40; total time=   5.5s
[CV] END ..............................kmeans__n_clusters=40; total time=   5.1s
[CV] END ..............................kmeans__n_clusters=41; total time=   5.1s
[CV] END ..............................kmeans__n_clusters=41; total time=   4.9s
[CV] END ..............................kmeans__n_clusters=41; total time=   4.9s
[CV] END ..............................kmeans__n_clusters=42; total time=   5.0s
[CV] END ..............................kmeans__n_clusters=42; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=42; total time=   4.8s
[CV] END ..............................kmeans__n_clusters=43; total time=   5.0s
[CV] END ..............................kmeans__n_clusters=43; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=43; total time=   5.1s
[CV] END ..............................kmeans__n_clusters=44; total time=   5.1s
```

```
[CV] END ..............................kmeans__n_clusters=44; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=44; total time=   5.2s
[CV] END ..............................kmeans__n_clusters=45; total time=   5.3s
[CV] END ..............................kmeans__n_clusters=45; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=45; total time=   5.2s
[CV] END ..............................kmeans__n_clusters=46; total time=   5.2s
[CV] END ..............................kmeans__n_clusters=46; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=46; total time=   5.0s
[CV] END ..............................kmeans__n_clusters=47; total time=   5.1s
[CV] END ..............................kmeans__n_clusters=47; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=47; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=48; total time=   5.2s
[CV] END ..............................kmeans__n_clusters=48; total time=   5.2s
[CV] END ..............................kmeans__n_clusters=48; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=49; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=49; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=49; total time=   5.0s
[CV] END ..............................kmeans__n_clusters=50; total time=   5.3s
[CV] END ..............................kmeans__n_clusters=50; total time=   5.2s
[CV] END ..............................kmeans__n_clusters=50; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=51; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=51; total time=   5.5s
[CV] END ..............................kmeans__n_clusters=51; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=52; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=52; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=52; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=53; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=53; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=53; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=54; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=54; total time=   5.3s
[CV] END ..............................kmeans__n_clusters=54; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=55; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=55; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=55; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=56; total time=   5.5s
[CV] END ..............................kmeans__n_clusters=56; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=56; total time=   6.4s
[CV] END ..............................kmeans__n_clusters=57; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=57; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=57; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=58; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=58; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=58; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=59; total time=   5.2s
[CV] END ..............................kmeans__n_clusters=59; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=59; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=60; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=60; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=60; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=61; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=61; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=61; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=62; total time=   5.5s
[CV] END ..............................kmeans__n_clusters=62; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=62; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=63; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=63; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=63; total time=   4.7s
[CV] END ..............................kmeans__n_clusters=64; total time=   5.3s
[CV] END ..............................kmeans__n_clusters=64; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=64; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=65; total time=   5.5s
[CV] END ..............................kmeans__n_clusters=65; total time=   6.2s
```

```
[CV] END ..............................kmeans__n_clusters=65; total time=   5.2s
[CV] END ..............................kmeans__n_clusters=66; total time=   5.5s
[CV] END ..............................kmeans__n_clusters=66; total time=   6.4s
[CV] END ..............................kmeans__n_clusters=66; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=67; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=67; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=67; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=68; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=68; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=68; total time=   6.4s
[CV] END ..............................kmeans__n_clusters=69; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=69; total time=   6.9s
[CV] END ..............................kmeans__n_clusters=69; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=70; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=70; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=70; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=71; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=71; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=71; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=72; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=72; total time=   6.7s
[CV] END ..............................kmeans__n_clusters=72; total time=   5.1s
[CV] END ..............................kmeans__n_clusters=73; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=73; total time=   6.6s
[CV] END ..............................kmeans__n_clusters=73; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=74; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=74; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=74; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=75; total time=   5.5s
[CV] END ..............................kmeans__n_clusters=75; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=75; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=76; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=76; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=76; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=77; total time=   6.6s
[CV] END ..............................kmeans__n_clusters=77; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=77; total time=   5.1s
[CV] END ..............................kmeans__n_clusters=78; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=78; total time=   6.8s
[CV] END ..............................kmeans__n_clusters=78; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=79; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=79; total time=   6.5s
[CV] END ..............................kmeans__n_clusters=79; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=80; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=80; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=80; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=81; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=81; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=81; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=82; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=82; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=82; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=83; total time=   6.8s
[CV] END ..............................kmeans__n_clusters=83; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=83; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=84; total time=   6.4s
[CV] END ..............................kmeans__n_clusters=84; total time=   6.0s
[CV] END ..............................kmeans__n_clusters=84; total time=   6.6s
[CV] END ..............................kmeans__n_clusters=85; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=85; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=85; total time=   5.5s
[CV] END ..............................kmeans__n_clusters=86; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=86; total time=   6.5s
[CV] END ..............................kmeans__n_clusters=86; total time=   5.4s
```

```
[CV] END ..............................kmeans__n_clusters=87; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=87; total time=   6.8s
[CV] END ..............................kmeans__n_clusters=87; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=88; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=88; total time=   6.8s
[CV] END ..............................kmeans__n_clusters=88; total time=   5.7s
[CV] END ..............................kmeans__n_clusters=89; total time=   6.7s
[CV] END ..............................kmeans__n_clusters=89; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=89; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=90; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=90; total time=   6.4s
[CV] END ..............................kmeans__n_clusters=90; total time=   5.2s
[CV] END ..............................kmeans__n_clusters=91; total time=   5.6s
[CV] END ..............................kmeans__n_clusters=91; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=91; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=92; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=92; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=92; total time=   5.4s
[CV] END ..............................kmeans__n_clusters=93; total time=   6.5s
[CV] END ..............................kmeans__n_clusters=93; total time=   6.8s
[CV] END ..............................kmeans__n_clusters=93; total time=   5.8s
[CV] END ..............................kmeans__n_clusters=94; total time=   6.4s
[CV] END ..............................kmeans__n_clusters=94; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=94; total time=   5.9s
[CV] END ..............................kmeans__n_clusters=95; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=95; total time=   6.4s
[CV] END ..............................kmeans__n_clusters=95; total time=   5.0s
[CV] END ..............................kmeans__n_clusters=96; total time=   6.5s
[CV] END ..............................kmeans__n_clusters=96; total time=   5.3s
[CV] END ..............................kmeans__n_clusters=96; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=97; total time=   6.2s
[CV] END ..............................kmeans__n_clusters=97; total time=   6.6s
[CV] END ..............................kmeans__n_clusters=97; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=98; total time=   6.5s
[CV] END ..............................kmeans__n_clusters=98; total time=   6.3s
[CV] END ..............................kmeans__n_clusters=98; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=99; total time=   6.1s
[CV] END ..............................kmeans__n_clusters=99; total time=   6.5s
[CV] END ..............................kmeans__n_clusters=99; total time=   6.2s
```

Out[ ]:
```
GridSearchCV(cv=3,
             estimator=Pipeline(steps=[('kmeans', KMeans(n_clusters=50)),
                                       ('log_reg',
                                        LogisticRegression(max_iter=5000,
                                                           multi_class='ovr',
                                                           random_state=42))]),
             param_grid={'kmeans__n_clusters': range(2, 100)}, verbose=2)
```

In [ ]:
```
grid_clf.best_params_
```

Out[ ]:
```
{'kmeans__n_clusters': 54}
```

In [ ]:
```
grid_clf.score(X_test, y_test)
```

Out[ ]:
```
0.98
```

2. 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.
- Also useful for data analysis and visualization.

2.1 DBSCAN

The algorithm works well if

- clusters are dense enough
- clusters are well separated by low-density region

DBSCAN works the following way:

- The algorithm defines clusters as continuous regions of high density
- For each instance, the algorithm counts how many instances are location within a small distance ϵ from it. This region is called the instance's ϵ neighbourhood
- If an instance has at least `min_samples` instances in its ϵ neighbourhood (including itself), then it is considered a *core instance* (located in dense region)
- All instances in the neighbourhood of a core instances belong to the same cluster.
- This neighbourhood may include other core instances.
- Therefore, a long sequence of neighbouring core isntances forms a single cluster.
- Any instance that is not a core instance and does not have one in its neighbourhood is considered an anomaly.

In [ ]:
```
from sklearn.datasets import make_moons
from sklearn.cluster import DBSCAN

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

Out[ ]:
```
DBSCAN(eps=0.05)
```

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

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

- Cluster index = -1 means they are considered as anomalies by the algorithm.
- We can identify the indices of the core instances using `core_sample_indices_`
- We can identify the core instances themselves using `components_`

In [ ]:
```
len(dbscan.core_sample_indices_)
```

Out[ ]:
```
808
```

In [ ]:
```
dbscan.core_sample_indices_
```

Out[ ]:
```
array([  0,   4,   5,   6,   7,   8,  10,  11,  12,  13,  14,  16,  17,
        18,  19,  20,  21,  22,  23,  24,  25,  26,  28,  29,  30,  31,
        32,  33,  34,  36,  38,  39,  41,  42,  44,  45,  47,  49,  50,
        51,  52,  53,  54,  55,  56,  58,  59,  61,  63,  64,  65,  66,
        67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,  79,
        80,  81,  83,  84,  85,  87,  88,  89,  90,  91,  93,  94,  96,
        97,  98, 102, 103, 104, 105, 106, 107, 108, 109, 110, 113, 114,
       115, 116, 117, 119, 120, 122, 123, 124, 125, 126, 127, 128, 129,
       130, 135, 136, 139, 140, 141, 143, 144, 145, 146, 147, 148, 149,
       150, 152, 153, 154, 155, 156, 157, 158, 159, 161, 162, 163, 164,
       165, 166, 167, 168, 169, 170, 172, 173, 174, 175, 176, 177, 178,
       179, 181, 182, 183, 185, 186, 187, 188, 189, 191, 193, 194, 195,
       196, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 209, 210,
       211, 212, 213, 214, 215, 216, 217, 218, 219, 221, 222, 223, 224,
       226, 228, 229, 230, 232, 233, 234, 235, 236, 238, 239, 240, 241,
       242, 243, 245, 246, 247, 248, 249, 250, 251, 252, 253, 255, 256,
       257, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271,
       272, 273, 274, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285,
       287, 288, 289, 290, 291, 293, 294, 295, 296, 297, 300, 301, 303,
       304, 305, 308, 309, 310, 311, 313, 315, 317, 318, 319, 320, 321,
       322, 323, 324, 327, 328, 329, 330, 332, 333, 335, 339, 340, 341,
       342, 343, 346, 347, 348, 349, 351, 352, 353, 354, 355, 356, 358,
       360, 361, 362, 363, 364, 365, 366, 367, 368, 370, 371, 372, 373,
       374, 375, 377, 378, 379, 380, 381, 382, 384, 385, 387, 388, 389,
       390, 391, 392, 393, 394, 395, 397, 398, 399, 400, 401, 402, 403,
       404, 405, 406, 409, 411, 412, 413, 414, 415, 416, 417, 418, 419,
       420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432,
       433, 435, 436, 437, 438, 440, 441, 442, 443, 444, 445, 446, 447,
       448, 449, 450, 451, 452, 453, 454, 456, 457, 458, 459, 461, 462,
       463, 464, 467, 468, 469, 471, 472, 473, 474, 475, 476, 477, 478,
       479, 480, 483, 484, 485, 486, 487, 488, 489, 491, 492, 493, 495,
       496, 497, 498, 499, 501, 502, 503, 504, 505, 506, 507, 508, 509,
       510, 511, 512, 513, 514, 515, 516, 518, 519, 520, 521, 523, 524,
       525, 526, 528, 529, 530, 531, 532, 533, 535, 536, 537, 538, 539,
       540, 541, 542, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553,
       554, 555, 556, 557, 559, 560, 562, 563, 564, 565, 566, 568, 569,
       570, 572, 573, 574, 575, 576, 578, 579, 580, 583, 584, 585, 586,
       588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 599, 600, 601,
       602, 603, 604, 605, 607, 610, 611, 614, 615, 616, 617, 620, 624,
       625, 627, 628, 629, 631, 633, 634, 635, 636, 637, 638, 639, 640,
       641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 652, 655, 656,
       657, 661, 662, 663, 664, 666, 667, 670, 671, 672, 673, 675, 676,
       677, 678, 679, 680, 681, 682, 684, 685, 686, 688, 689, 690, 691,
       692, 694, 695, 696, 697, 698, 703, 704, 705, 706, 708, 709, 710,
       711, 712, 713, 714, 716, 717, 718, 719, 721, 722, 723, 724, 726,
       729, 730, 731, 733, 735, 736, 737, 738, 739, 740, 741, 742, 743,
       744, 745, 746, 748, 749, 750, 751, 752, 753, 754, 756, 757, 758,
       759, 760, 761, 762, 763, 765, 766, 768, 770, 772, 773, 774, 775,
       776, 777, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 791,
       792, 793, 794, 795, 796, 797, 798, 799, 800, 802, 803, 804, 805,
       806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818,
       819, 820, 821, 822, 824, 825, 826, 827, 828, 829, 830, 831, 832,
       835, 836, 837, 838, 839, 840, 841, 842, 843, 845, 846, 848, 849,
       850, 851, 852, 853, 854, 855, 857, 858, 860, 861, 862, 863, 864,
       865, 867, 868, 870, 871, 873, 877, 878, 879, 880, 882, 883, 884,
       885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897,
       898, 899, 902, 903, 904, 905, 906, 907, 908, 910, 912, 913, 916,
       918, 919, 920, 921, 922, 923, 925, 926, 928, 929, 930, 931, 932,
       933, 934, 935, 937, 938, 939, 940, 941, 942, 943, 944, 945, 947,
       948, 949, 951, 952, 953, 954, 956, 958, 959, 960, 961, 962, 963,
       964, 965, 966, 967, 968, 969, 970, 972, 974, 975, 976, 978, 979,
       980, 982, 983, 984, 985, 986, 987, 988, 990, 992, 993, 995, 997,
       998, 999], dtype=int64)
```

In [ ]:
```
dbscan.components_
```

Out[ ]:
```
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]])
```

2.1.1 Predicting clusters

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier

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

Out[ ]:
```
KNeighborsClassifier(n_neighbors=50)
```

In [ ]:
```
X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
knn.predict(X_new)
```

Out[ ]:
```
array([6, 0, 3, 2], dtype=int64)
```

In [ ]:
```
knn.predict_proba(X_new)
```

Out[ ]:
```
array([[0.24, 0.  , 0.  , 0.  , 0.  , 0.76],
       [1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.3 , 0.7 , 0.  , 0.  ],
       [0.  , 0.  , 1.  , 0.  , 0.  , 0.  ]])
```

2.1.2 Summary of DBSCAN

Advantages

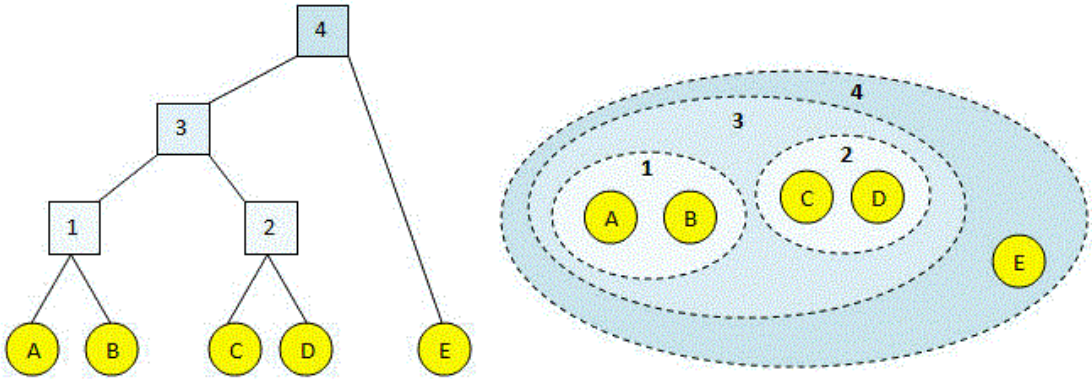
- DBSCAN is capable of identifying any number of clusters of any shape
- Is robust to outliers
- Has only two hyperparameters: `eps` and `min_samples`

Disadvantages

- If Density varies significantly across the clusters, it can be impossible for it to capture all the clusters properly.

3. Other Clustering Algorithms

3.1 Agglomerative Clustering

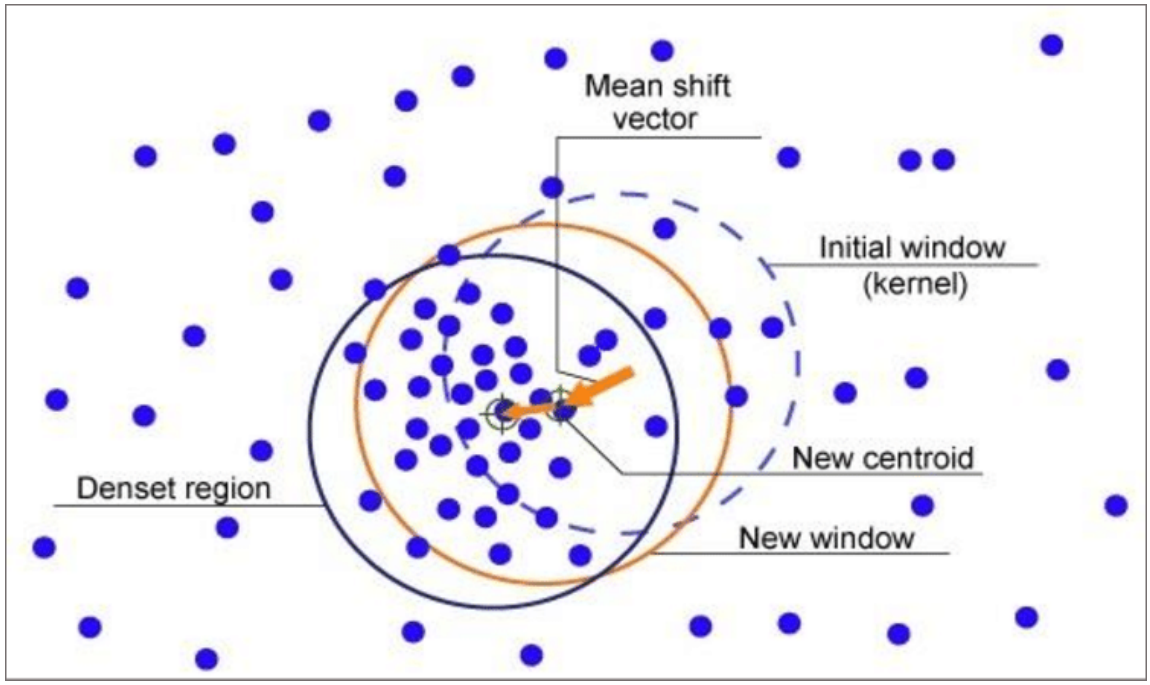


- A hierarchy of clusters is built from the bottom up
- At each iteration, agglomerative clustering connects the nearest pair of clusters.
- Produces a flexible and informative cluster tree instead of forcing you to choose a particular cluster scale
- Can be used with any pairwise distance

3.2 Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH)

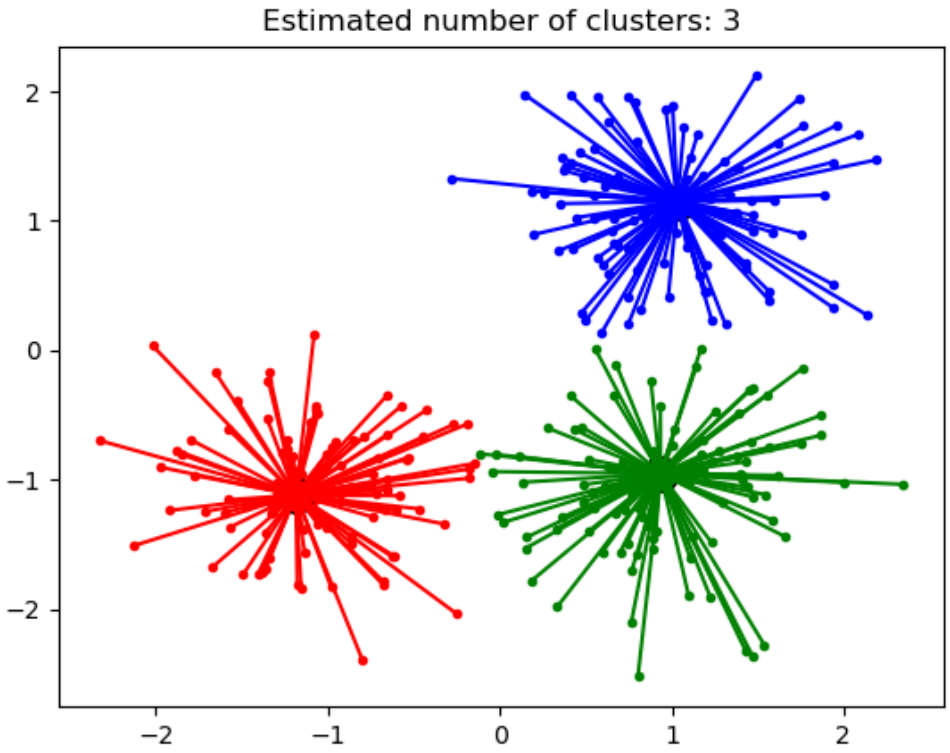
- Designed for very large datasets and can be faster than batch K-Means as long as number of feature < 20.
- A clustering algorithm that can cluster large datasets by first generating a small and compact summary of the large dataset that retains as much information as possible.
- During training, it builds a tree structure containing just enough information to quickly assign each new instance to a cluster without having to store all the instances in the tree.
- This approach allows it to use limited memory when handling huge datasets.

3.3 Mean-Shift



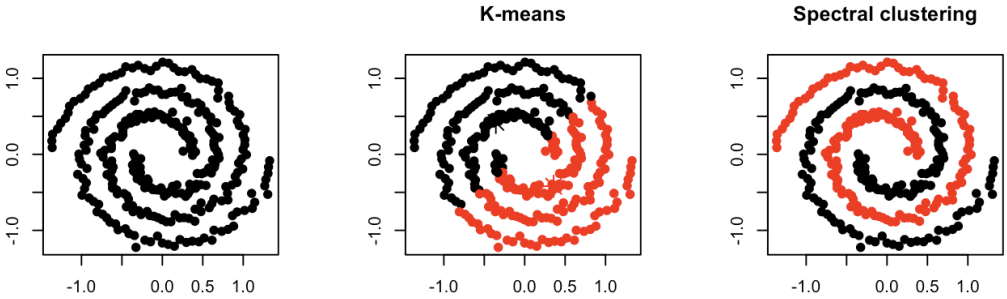
- Algorithm starts by placing a circle centered on each instance.
- For each circle, it computes the mean of all the instances located within it and it shifts the circle so that it is centered on the mean
- Then it iterates this mean-shifting step until all the circles stop moving.
- Mean-shift shfits the circles in the direction of higher density, until each of them has found a lock density meximum.
- Finally, all instances whose circle have settled int he same place are assigned to the same cluster.
- Like DBSCAN, it can find any number of clusters of any shape and has very few hyperparameters and relies on local density estimaion.
- Unlike DBSCAN, it tends to chop clusters into pieces when they have internal density variations.
- But the computation complexity is $O(m^2)$ so not suited for large datasets

3.4 Affinity Propagation



- Algorithm uses a voting system where instances vote for similar instances to be their representatives.
- Once algorithm converges, each representative and its voters form a cluster.
- Affinity propagation can detect any number of clusters of different sizes.
- But the computation complexity is $O(m^2)$ so not suited for large datasets

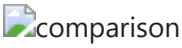
3.5 Spectral Clustering



- Algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (reduces dimensionality)
- Then it uses another clustering algorithm in this low-dimensional space (Scikit-Learn uses K-Means).
- Spectral clustering can capture complex cluster structures
- Also good to be used to cut graphs (eg: to identify clusters of friends on a social network)
- But it does not scale well to large numbers of instances

- It does not behave well when the clusters have very different sizes.

Comparison of all clustering algorithms



4. Gaussian Mixtures

- Can be used for density estimation, clustering and anomaly detection
- Gaussian mixture model (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions who parameters are unknown
- All instances generated from a single gaussian distribution form a cluster that looks like an ellipsoid.
- Each cluster can have a different ellipsoidal shape, size, density and orientation.
- When we observe an instance, we know it was generated from one of the Gaussian distributions but not told which one and do not know what the parameters are.

4.1 Clustering with Gaussian Mixtures

In [ ]:
```
X1, y1 = make_blobs(n_samples=1000, centers=((4, -4), (0, 0)), random_state=42)
X1 = X1.dot(np.array([[0.374, 0.95], [0.732, 0.598]]))
X2, y2 = make_blobs(n_samples=250, centers=1, random_state=42)
X2 = X2 + [6, -8]
X = np.r_[X1, X2]
y = np.r_[y1, y2]
```

In [ ]:
```
from sklearn.mixture import GaussianMixture

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

Out[ ]:
```
GaussianMixture(n_components=3, n_init=10, random_state=42)
```

The parameters that the algorithm estimated are:

In [ ]:
```
gm.weights_
```

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

In [ ]:
```
gm.means_
```

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

In [ ]:
```
gm.covariances_
```

Out[ ]:
```
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]]])
```

We can check whether the algorithm converged and how many iterations it took

In [ ]:
```
gm.converged_
```

Out[ ]:
```
True
```

In [ ]:
```
gm.n_iter_
```

Out[ ]:
```
4
```

We can use the model to predict which cluster each instance belongs to (hard clustering) or the probabilities that it came from each cluster (soft clustering).

In [ ]:
```
# Hard Clustering
gm.predict(X)
```

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

In [ ]:
```
# Soft Clustering
gm.predict_proba(X)
```

Out[ ]:
```
array([[9.76741808e-01, 6.78581203e-07, 2.32575136e-02],
       [9.82832955e-01, 6.76173663e-04, 1.64908714e-02],
       [7.46494398e-05, 9.99923327e-01, 2.02398402e-06],
       ...,
       [4.26050456e-07, 2.15512941e-26, 9.99999574e-01],
       [5.04987704e-16, 1.48083217e-41, 1.00000000e+00],
       [2.24602826e-15, 8.11457779e-41, 1.00000000e+00]])
```

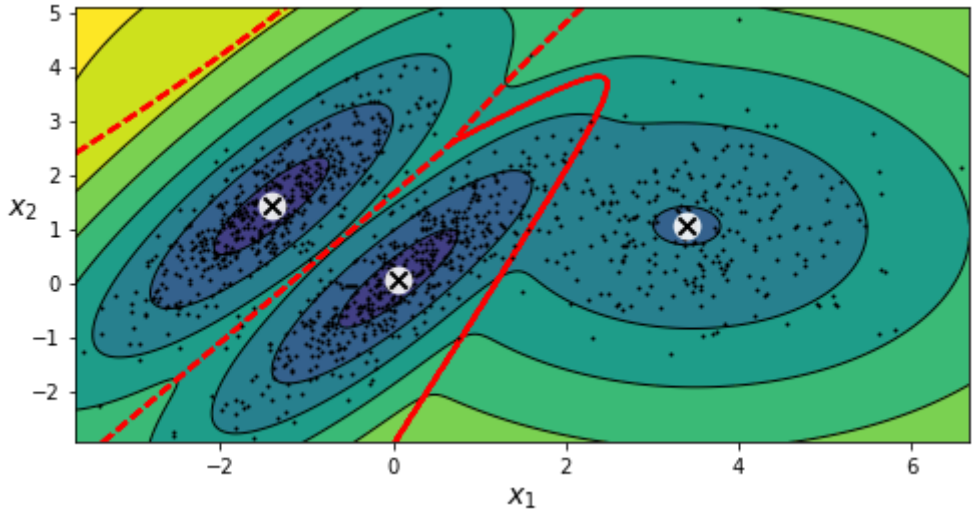
Plotting the resulting decision boundary and density contours

In [ ]:
```
def plot_centroids(centroids, weights=None, circle_color='w', cross_color='k'):
    if weights is not None:
        centroids = centroids[weights > weights.max() / 10]
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='o', s=35, linewidths=8,
                color=circle_color, zorder=10, alpha=0.9)
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='x', s=2, linewidths=12,
                color=cross_color, zorder=11, alpha=1)
```

In [ ]:
```
from matplotlib.colors import LogNorm
import matplotlib.pyplot as plt

def plot_gaussian_mixture(clusterer, X, resolution=1000, show_ylabels=True):
    mins = X.min(axis=0) - 0.1
    maxs = X.max(axis=0) + 0.1
    xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], resolution),
                         np.linspace(mins[1], maxs[1], resolution))
    Z = -clusterer.score_samples(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
```

```
    plt.contourf(xx, yy, Z,
                 norm=LogNorm(vmin=1.0, vmax=30.0),
                 levels=np.logspace(0, 2, 12))
    plt.contour(xx, yy, Z,
                norm=LogNorm(vmin=1.0, vmax=30.0),
                levels=np.logspace(0, 2, 12),
                linewidths=1, colors='k')

    Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contour(xx, yy, Z,
                linewidths=2, colors='r', linestyles='dashed')

    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)
    plot_centroids(clusterer.means_, clusterer.weights_)

    plt.xlabel("$x_1$", fontsize=14)
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
        plt.tick_params(labelleft=False)
```

In [ ]:
```
plt.figure(figsize=(8, 4))

plot_gaussian_mixture(gm, X)

plt.show()
```



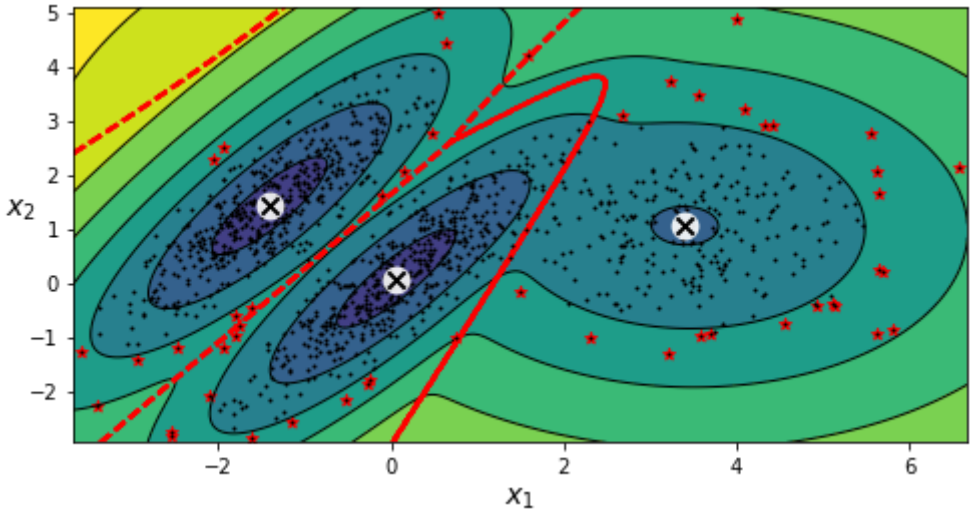
4.2 Anomaly Detection with Gaussian Mixtures

- Anomaly Detection is the task of detecting instances that deviate strongly from the norm
- Using Gaussian Mixtures, any instance located in the low-density region can be considered an anomaly.

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

In [ ]:
```
plt.figure(figsize=(8, 4))

plot_gaussian_mixture(gm, X)
```

```
plt.scatter(anomalies[:, 0], anomalies[:, 1], color='r', marker='*')
plt.ylim(top=5.1)

plt.show()
```



5. Other Algorithms for Anomaly and Novelty Detection

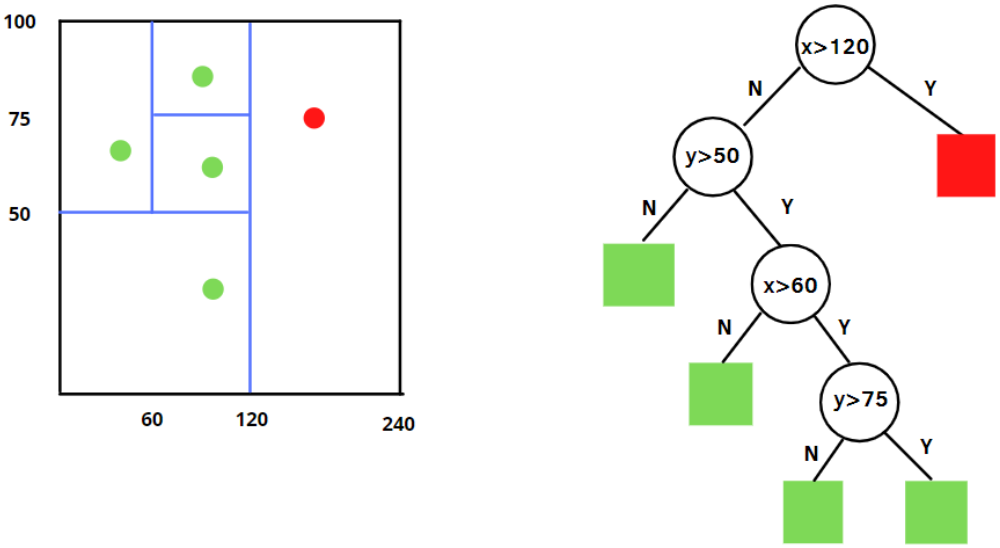
5.1 PCA

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

5.2 Fast-MCD

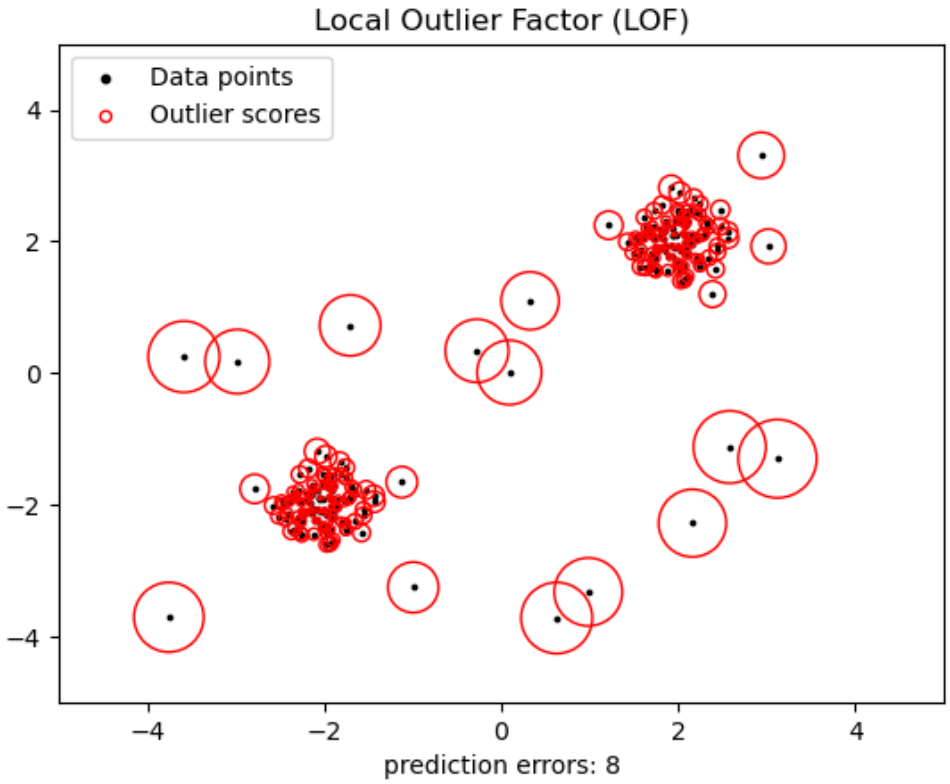
- Assumes that the normal instances are generated from a single Gaussian Distribution.
- Also assumes that the dataset is contaminated with outliers that were not generated from this Gaussian distribution.
- When algorithm estimates the parameters of the Gaussian Distribution, it is careful to ignore the instances that are most likely outliers.

5.3 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 picks a random threshold value to split the dataset in two.
- The dataset gets chopped gradually into pieces until all instances end up isolated from other instances.
- Anomalies are usually far from other instances so on avereage, they tend to get isolated in fewer steps than normal instances

5.4 Local Outlier Factor



- Algorithm compares the density of instances around a given instance to the density around its neighbours.
- An anomaly is often more isolated than it *k* nearest neighbours.