Clustering Algorithms: K-means and DBSCAN

羅右鈞 Yu-Chun Lo
howard.lo@nlplab.cc

前情提要：

本投影片跟 Ipython 的內容是一樣的，只是會將程式碼分頁呈現，並加上中文註解讓初學者易讀，有些過於冗長且非重點之程式碼就不會放在此投影片，建議邊讀此投影片，邊玩看看 Ipython。

K-means

K-means 演算法

K-means

The Algorithm

1. Randomly choose k centroids $C = \{c_1, c_2, \dots, c_k\}$ from the data points $X = \{x_1, x_2, \dots, x_n\} \in \mathbb{R}^D$.
2. For each data point x_i, find the nearest centroid c_j as its corresponding cluster using *sum of squared distance*

   $$D(x_i, c_j) = \sum_{i=1}^{n} \|x_i - c_j\|^2.$$

3. For each cluster, update its centroid by computing means value along the dimension of data points in the cluster.
4. Compute the displacement between the old and the new centroids and repeat steps 2 and 3 if the displacement is less than a threshold (converged).

首先先引入本教學會用到的套件

In [1]:
```
# Start from importing necessary packages.
import warnings
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

from IPython.display import display
from sklearn import metrics # for evaluations
from sklearn.datasets import make_blobs, make_circles # for generating experimental data
from sklearn.preprocessing import StandardScaler # for feature scaling
from sklearn.cluster import KMeans
from sklearn.cluster import DBSCAN

# make matplotlib plot inline (Only in Ipython).
warnings.filterwarnings('ignore')
%matplotlib inline
```

產生含有三個 clusters 的二維資料

In [2]:
```
# Generate data.
# `random_state` is the seed used by random number generator for reproducibility (default=None).
X, y = make_blobs(n_samples=5000,
                  n_features=2,
                  centers=3,
                  random_state=170)

# Print data like Ipython's cell output (Only in Ipython, otherwise use `print`).
display(X)
display(y)
```

```
array([[-4.01009423, -1.01473496],
       [ 1.00550526,  0.13163222],
       [ 2.06563121, -0.24527689],
       ...,
       [-5.09493013,  1.47160372],
       [-9.61459714, -4.91848716],
       [-7.72675795, -5.86656563]])
```

← 這是我們的訓練資料 X，共 5000 筆，每筆資料是二維，因此 X 是 (5000 x 2) 的矩陣。

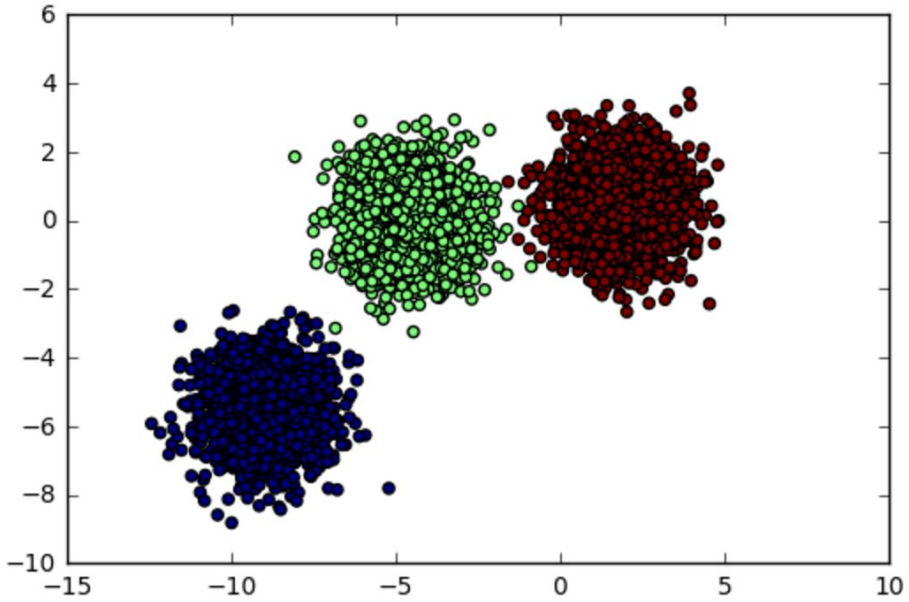
```
array([1, 2, 2, ..., 1, 0, 0])
```

← 這是我們每筆資料對應的 cluster label y (ground truth)，在此例有三個 clusters，因此 label = {0, 1, 2}，共 5000 筆，因此 y 是 5000 維的向量。

讓我們把資料視覺化一下，看看長什麼樣子。

In [3]:
```
# Plot the data distribution (ground truth) using matplotlib `scatter(axis-x, axis-y, color)`.
plt.scatter(X[:,0], X[:,1], c=y)
```

Out[3]:
```
<matplotlib.collections.PathCollection at 0x111d4c310>
```



In [4]:
```
# Perform K-means on our data (Train centroids)
kmeans = KMeans(n_clusters=3,
                n_init=3,
                init='random',
                tol=1e-4,
                random_state=170,
                verbose=True).fit(X)
```

```
Initialization complete
Iteration  0, inertia 93044.046
Iteration  1, inertia 11941.892
Iteration  2, inertia 9733.271
Converged at iteration 2
Initialization complete
Iteration  0, inertia 49491.222
Iteration  1, inertia 43229.847
Iteration  2, inertia 42926.507
Iteration  3, inertia 42574.503
Iteration  4, inertia 41999.156
Iteration  5, inertia 39853.967
Iteration  6, inertia 28163.261
Iteration  7, inertia 11338.225
Iteration  8, inertia 9733.783
Converged at iteration 8
Initialization complete
Iteration  0, inertia 35757.303
Iteration  1, inertia 10003.824
Iteration  2, inertia 9733.154
Converged at iteration 2
```

使用 scikit-learn 所提供的 KMeans 來跑我們的資料。

參數解釋

- n_clusters：你想在這資料上分幾群，也就是「選擇‘k”」啦。
- n_inits：你想讓 scikit-learn 的 k-means 演算法「跑幾次」，這邊我們是設定成3，意思就是它會跑3 次的 k-means，並且選擇最低 inertia 的那一次 (這邊的 inertia 目標就是 minimize sum of squared distance，distance 越小當然就越好囉！)，當作 k-means 的訓練完後的結果，也就是留著最好的 centroids。
- init：還記得 k-means 演算法一開始會隨機選擇centroids 嗎？這邊就是設定「怎麼選擇centroids 的演算法」，有 “random” 跟 “k-means++” (後面會解釋) 可以選擇。
- tol：k-means 收斂時距離門檻值的比例，比例越高會越早視為收斂結束。
- random_state：給一固定值可以讓我們一直跑出同樣的結果，通常debug 才會這麼用，一般來說不需要。
- verbose：設定 True 則會 print 出訓練 k-means 時的過程，反之則False。

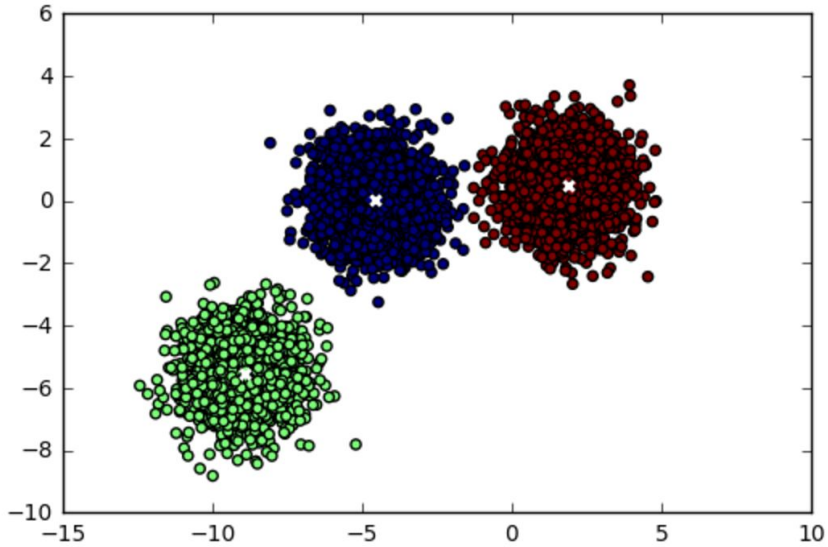
In [5]:
```
# Retrieve predictions and cluster centers (centroids).
display(kmeans.labels_)
display(kmeans.cluster_centers_)
```

```
array([0, 2, 2, ..., 0, 1, 1], dtype=int32)

array([[-4.55676387,  0.04603707],
       [-8.94710203, -5.51613184],
       [ 1.89450492,  0.5009336 ]])
```

← 使用 “kmeans.labels_” 來取得我們預測的結果，同樣是 5000 維的向量。
← 使用 “kmeans.cluster_centers_” 來取得訓練出來的 centroids，有三個 clusters，資料為二維，因此是 (3x2) 的矩陣。

In [6]:
```
# Plot the predictions.
plt.scatter(X[:,0], X[:,1], c=kmeans.labels_)
plt.scatter(kmeans.cluster_centers_[:,0],
            kmeans.cluster_centers_[:,1],
            c='w', marker='x', linewidths=2)
```

Out[6]:
```
<matplotlib.collections.PathCollection at 0x1125648d0>
```



← 把預測的結果視覺化出來，並且也順便把 centroids 給畫上去。

注意：每次預測出來的 cluster 雖然跟 ground truth 畫的 cluster 不同，但這只是因為 k-means 隨機選擇 centroids 的關係，所以有可能每次跑出來的結果中，同樣的一群資料被分群到的 cluster label 會不同，但是重點是看整體的分群結果喔！顏色不同其實也沒關係。

In [7]:
```
# We can make new predictions without re-run kmeans (simpily find nearest centroids).
X_new = np.array([[10,10], [-10, -10], [-5, 10]])
y_pred = kmeans.predict(X_new)

""" The below code is equivalent to:
y_pred = KMeans(...).fit_predict(X), but this needs to fit kmeans again.
"""

display(y_pred)
```

```
array([2, 1, 0], dtype=int32)
```

← 訓練完 k-means 之後，使用 “kmeans.predict()” 來預測新資料 X_new。

In [8]:
```
# We can get distances from data point to every centroid

""" The below code is equivalent to:
from sklearn.metrics.pairwise import euclidean_distances
euclidean_distances(X_new, kmeans.cluster_centers_)
"""

kmeans.transform(X_new)
```

Out[8]:
```
array([[ 17.63464636,  24.48965134,  12.48724601],
       [ 11.42592142,   4.60582976,  15.86659553],
       [  9.96382639,  16.01030799,  11.73739582]])
```

← 使用 “kmeans.transform()” 取得 X_new 中每個資料點到 每個 centroid 的距離。如果你有 M 個資料點，N 個 clusters，則你會有 (MxN) 的 矩陣，每個 row 對應至一個資料點到 N 個 clusters 的距離。

K-means++

K-means++ 演算法，更聰明初始化 centroid 的方式

A Smarter Way to Initialize Centroids: K-means++

Since *K-means* highly depends on the initialization of the centroids, the clustering results may be converged to a local minimum. We can address this by setting `init='kmeans++'` instead of `'random'`. *K-means++* initializes centroids in a smarter way to speed up convergence. The algorithm is as follows:

1. Randomly choose one centroid from the data points.
2. For each data point x_i, compute the distance $D(x_i, c_j)$ where c_j is nearest to x_i.
3. Randomly choose one new data point as a new centroid using *weighted probability distribution* proportional to $D(x_i, c_j)^2$.
4. Repeat steps 2 and 3 until k centroids have been chosen.
5. Now we have initialized centroids, run *K-means* algorithm.

簡單來說，kmeans++ 跟 kmeans 的差別是：
主要希望選到的 centroids 它們各自的距離越大越好，避免選到離彼此之間很近的 centroids。

In [9]:
```
# Perform K-means++ on our data.
kmeans_plus_plus = KMeans(n_clusters=3,
                          n_init=3,
                          init='k-means++',
                          tol=1e-4,
                          random_state=170,
                          verbose=True).fit(X)
```

```
Initialization complete
Iteration  0, inertia 14504.164
Iteration  1, inertia 9735.745
Iteration  2, inertia 9733.168
Converged at iteration 2
Initialization complete
Iteration  0, inertia 10751.002
Iteration  1, inertia 9733.462
Converged at iteration 1
Initialization complete
Iteration  0, inertia 12316.398
Iteration  1, inertia 9733.520
Converged at iteration 1
```

想使用 k-means++，你只要設定 init='k-means++' 就可以囉！很簡單吧？而且有沒有發現 k-means 每次在訓練的時候，收斂速度變快了呢？這要歸功於 k-means++ 一開始選了很好的 centroids，導致後面在 clustering 的時候就會很快！

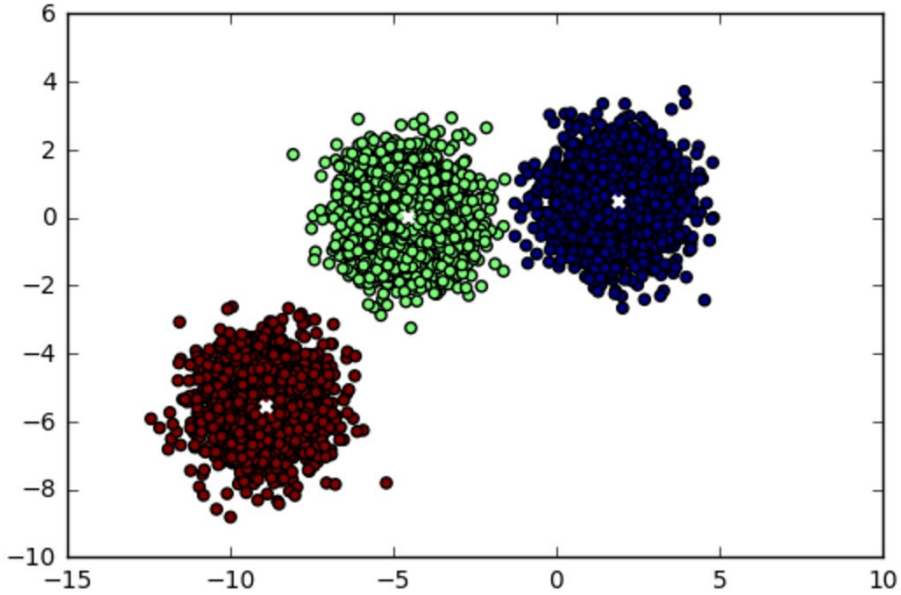
在 scikit-learn 中，你其實也不需要手動設定 init='k-means++'，因為它默認值就是如此囉！

You can see that *K-means++* converges much faster than *K-means*!

K-means++ 視覺化的預測結果

In [10]:
```
# Plot the predictions.
plt.scatter(X[:,0], X[:,1], c=kmeans_plus_plus.labels_)
plt.scatter(kmeans_plus_plus.cluster_centers_[:,0],
            kmeans_plus_plus.cluster_centers_[:,1],
            c='w', marker='x', linewidths=2)
```

Out[10]: <matplotlib.collections.PathCollection at 0x1127a4250>



請完成 “clustering-lab1.ipynb”

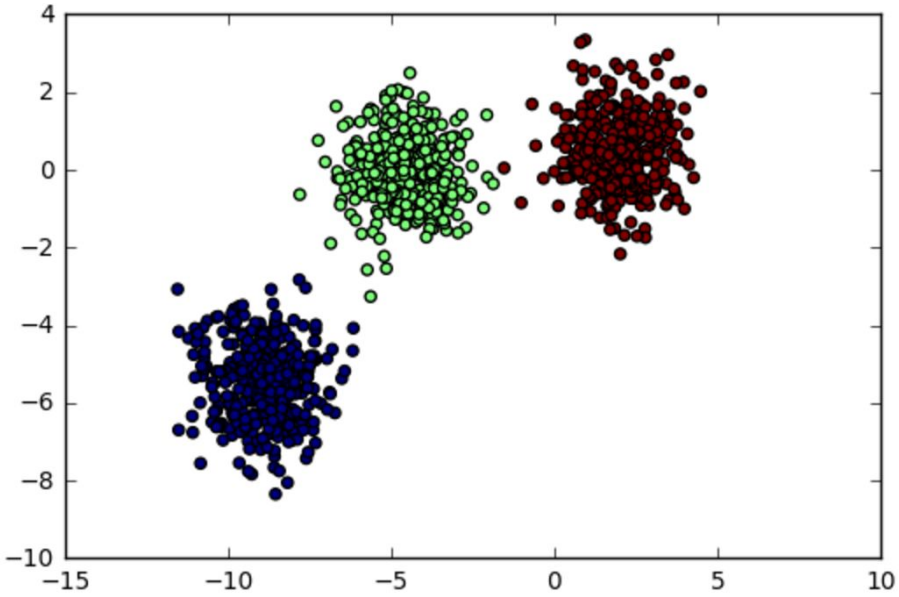
再來，我們要來分析 K-means 的種種弱點以及如何解決它們

弱點一：需要選擇正確的 “k”

Drawback 1: Need to choose a right number of clusters

In [11]:
```
# Generate data.
X, y = make_blobs(n_samples=1000,
                  n_features=2,
                  centers=3,
                  random_state=170)

# Plot the data distribution.
plt.scatter(X[:,0], X[:,1], c=y)
```

Out[11]: <matplotlib.collections.PathCollection at 0x1129adc90>

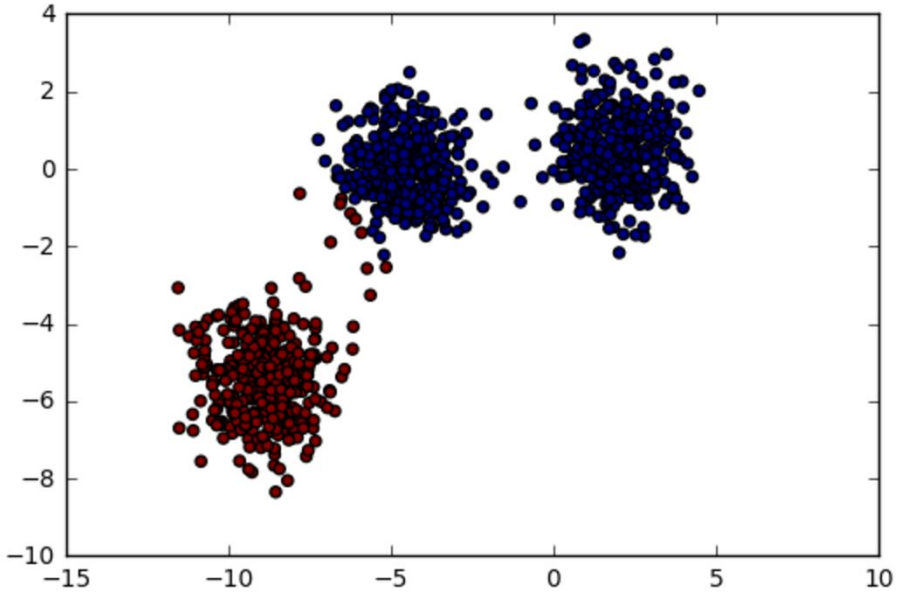


首先，我們一樣先產生三群資料，並且視覺化 ground truth 來看看。

如果我們選擇錯誤的 “k”，會變得怎麼樣呢？

In [12]:
```
# Run k-means on non-spherical data.
y_pred = KMeans(n_clusters=2, random_state=170).fit_predict(X)

# Plot the predictions.
plt.scatter(X[:,0], X[:,1], c=y_pred)
```

Out[12]: <matplotlib.collections.PathCollection at 0x112b5bc90>



答案是：k 選錯，還是可以 work！

如果我們選擇 k=2，結果還是可以分群，你可以看到中間的 cluster 被歸類成藍色，是因為與右邊的 cluster 比較近的關係。

雖然能 work，但與 ground truth 明顯不符，因此未來如果要預測新資料，也許會表現得不好。

那 … 我們要怎麼選擇一個好的 k 呢？

Solution: Measuring Cluster Quality to Determine the Number of Clusters

Supervised method

Homogeneity: Each cluster contains only members of a single class.

Completeness: All members of a given class are assigned to the same cluster.

如果你的訓練資料中已經含有 ground truth，也就是知道哪一筆資料點是屬於哪一個 cluster，那你可以使用 “supervised” 的方式來評估 clusters 的好壞！主要有兩種指標：

- Homogeneity：每個 cluster 中的資料點最好都要是同一個 label。
- Completeness：每個資料點所被預測出來的 cluster label 最好跟 ground truth 一樣（猜對）。

Unsupervised method

Sihouette Coefficient: Evaluate how well the **compactness** and the **separation** of the clusters are. (Note that the notation below is consistent with the above content.) Using *Sihouette Coefficient*, we can choose an optimal value for number of clusters.

$a(x_i)$ denotes the **mean intra-cluster distance**. Evaluate the compactness of the cluster to which x_i belongs. (The smaller the more compact)

$$a(x_i) = \frac{\sum_{x_k \in C_j,\, k \neq i} D(x_i, x_k)}{|C_j| - 1}$$

For the data point x_i, caculate its average distance to all the other data points in its cluster. (Minusing one in denominator part is to leave out the current data point x_i)

$b(x_i)$ denotes the **mean nearest-cluster distance**. Evaluate how x_i is separated from other clusters. (The larger the more separated)

$$b(x_i) = \min_{C_j:\, 1 \leq j \leq k,\, x_i \notin C_j} \left\{ \frac{\sum_{x_k \in C_j} D(x_i, x_k)}{|C_j|} \right\}$$

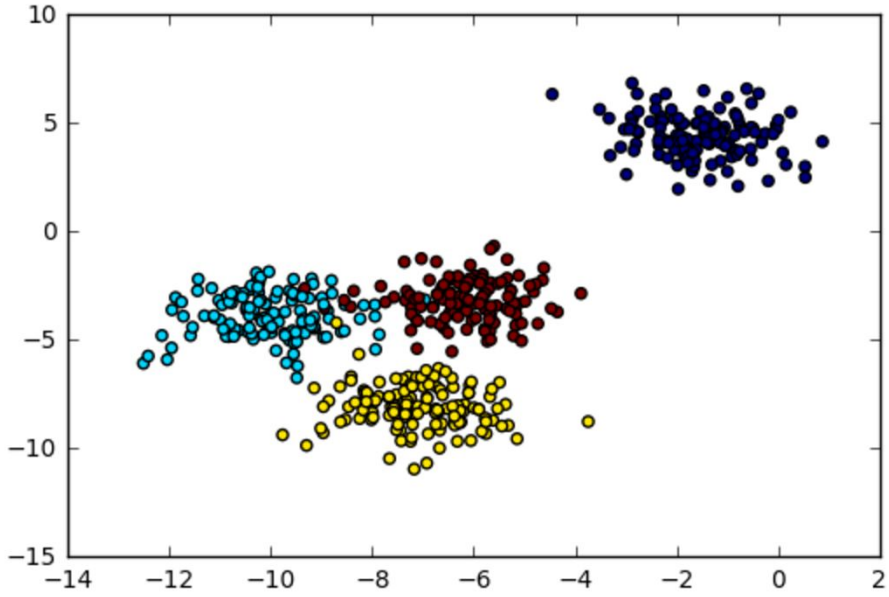
For the data point x_i and all the other clusters not containing x_i, caculate its average distance to all the other data points in the given clusters. Find the minimum distance value with respect to the given clusters.

Finally, *Silhouette Coefficient*: $s(x_i) = \frac{b(x_i) - a(x_i)}{\max\{a(x_i), b(x_i)\}}$, $-1 \leq s(x_i) \leq 1$. Want $a(x_i) < b(x_i)$ and $a(x_i) \to 0$ so as to $s(x_i) \to 1$.

如果很不幸地，你的訓練資料中沒有 ground truth，沒關係！我們還有 “unsupervised” 的方式可以評估 clusters 的好壞！

In [13]:
```
# Generate data.
# This particular setting has one distinct cluster and 3 clusters placed close together.
X, y = make_blobs(n_samples=500,
                  n_features=2,
                  centers=4,
                  cluster_std=1,
                  center_box=(-10.0, 10.0),
                  shuffle=True,
                  random_state=1)

# Plot the data distribution.
plt.scatter(X[:,0], X[:,1], c=y)
```

Out[13]: <matplotlib.collections.PathCollection at 0x112ccd950>

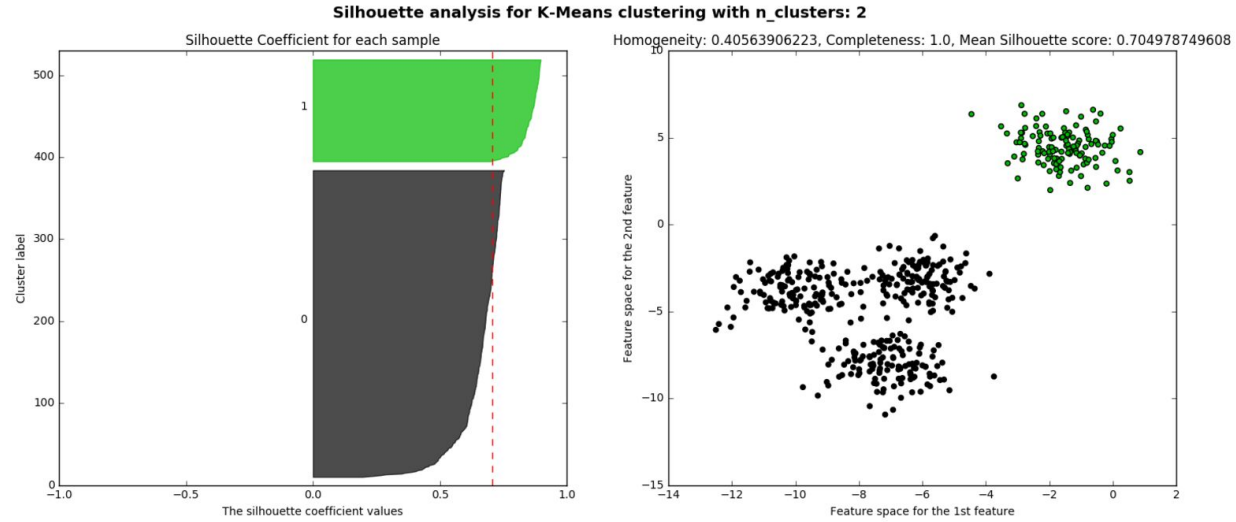


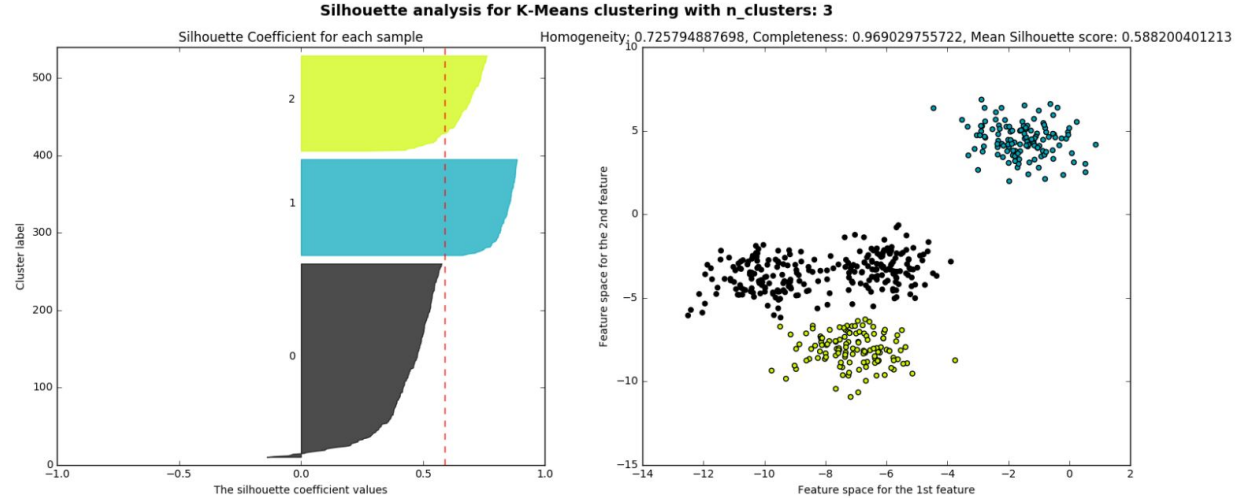
首先，我們先產生四群 clusters 的資料，並且視覺化出來，注意到一點是我們故意 產生出三群 clusters 是稍微有重疊到的，方便我們用上述所提到的 cluster quality measurements 來跑看看如果選擇不同的 “k”，系統跑出來的 cluster quality 指數是好還是壞。

In [14]:
```
# List of number of clusters
range_n_clusters = [2, 3, 4, 5, 6]

# For each number of clusters, perform Silhouette analysis and visualize the results.
for n_clusters in range_n_clusters:

    # Perform k-means.
    kmeans = KMeans(n_clusters=n_clusters, random_state=10)
    y_pred = kmeans.fit_predict(X)

    # Compute the cluster homogeneity and completeness.
    homogeneity = metrics.homogeneity_score(y, y_pred)
    completeness = metrics.completeness_score(y, y_pred)

    # Compute the Silhouette Coefficient for each sample.
    s = metrics.silhouette_samples(X, y_pred)

    # Compute the mean Silhouette Coefficient of all data points.
    s_mean = metrics.silhouette_score(X, y_pred)
```

上述程式碼解釋:

- 首先我們先設定要跑的 “k” 的範圍，此例我們嘗試跑 k=2, 3, 4, 5, 6 各個不同的 case，並且使用 for 迴圈，分別計算不同 “k” 之分群結果的 homogeneity, completeness 跟 silhouette coefficient。
- 使用 “homogeneity_score(y, y_pred)” 與 “completeness_score(y, y_pred)” 即可計算對應的兩種指標
- 注意到 scikit-learn 在計算 silhouette coefficient 時有提供兩種選擇:
  - 使用 “silhoette_samples(X, y_pred)” 來計算每個資料點的 silhouette coefficient。
  - 使用 “silhoette_score(X, y_pred)” 來計算每個資料點的 silhouette coefficient 加總之後的平均值。



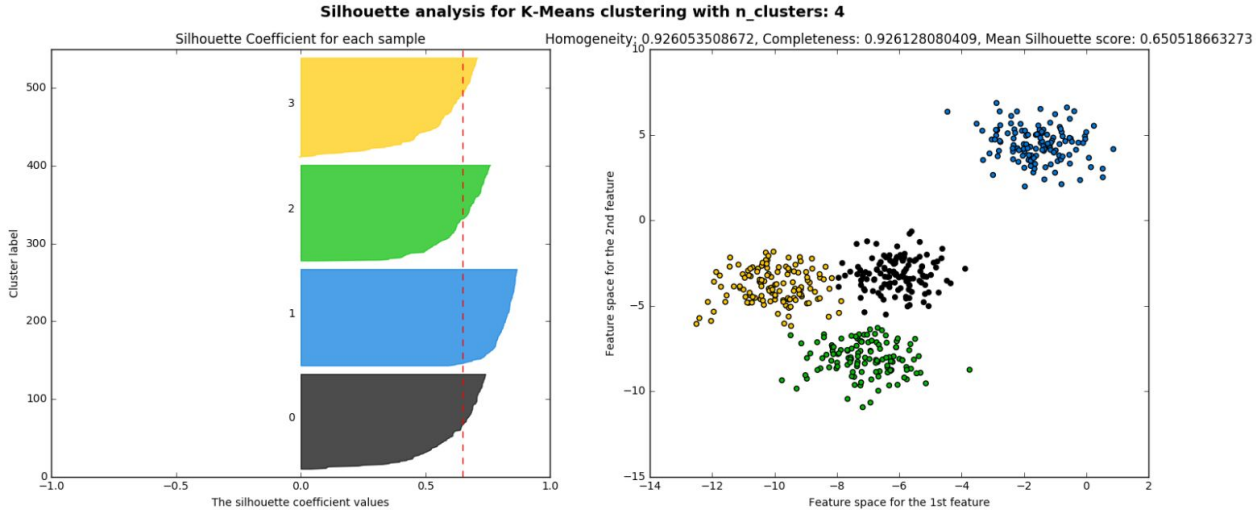


由於視覺化的程式碼稍微複雜，有興趣請查看 Ipython。

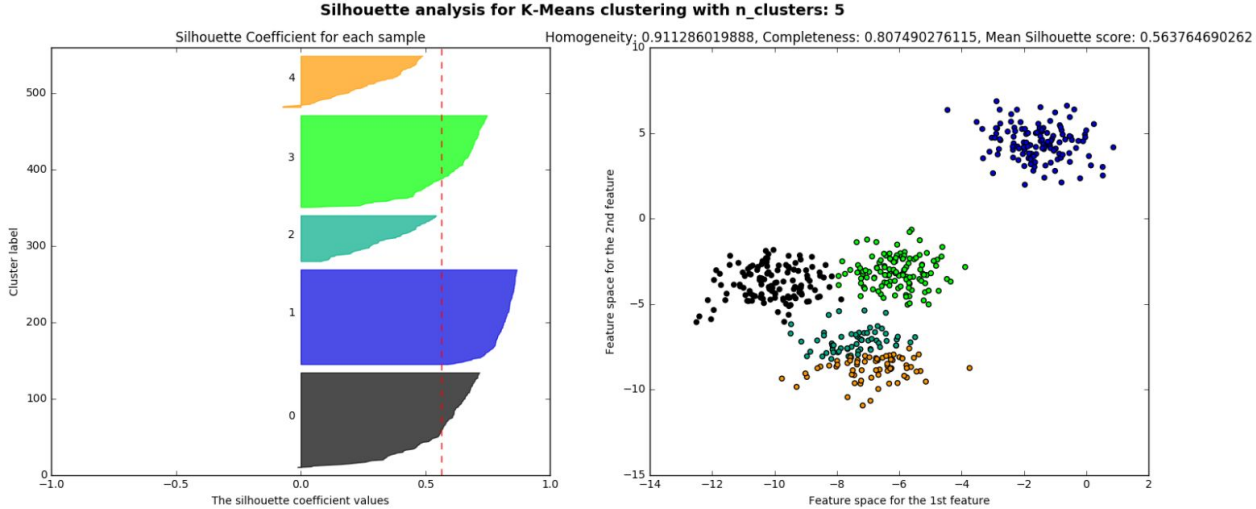
我們先來講講怎麼看這張圖吧！

右邊很簡單，就是分群後的結果，
主要是左邊：

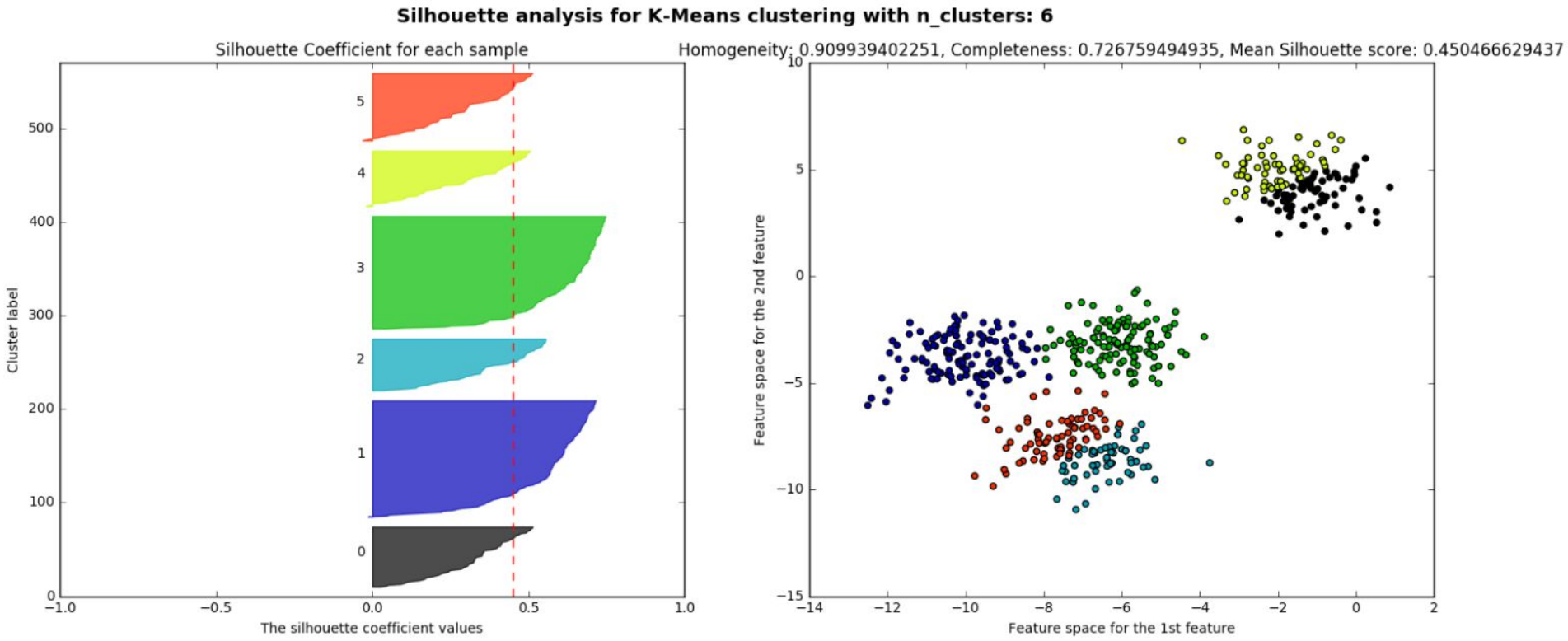
- 橫軸代表 silhouette coefficient。
- 縱軸代表 data points。
- 相同 cluster 的資料點畫成相同顏色，且該 cluster 內的資料點之 silhouette coefficient 皆排序過。
- 注意：紅色的垂直虛線是 “silhouette_score()” 算出來的全部 silhouette coefficient 加總之平均。



← k=4 時，每個 cluster 內的資料點之 silhouette coefficient 普遍高於平均值(超出紅色垂直線)，因此以 k=4 來講，分群效果還算不錯。



← k=5 時，可以注意到有兩群(土黃色以及蒂芬尼藍)的平均 silhouette coeffiecient 很低，因為兩群在右圖中重疊很多(影響到 silhouette coeffiecient 中計算 mean nearest cluster distance b(x)的指標)，因此 k=5 不是一個好選擇。



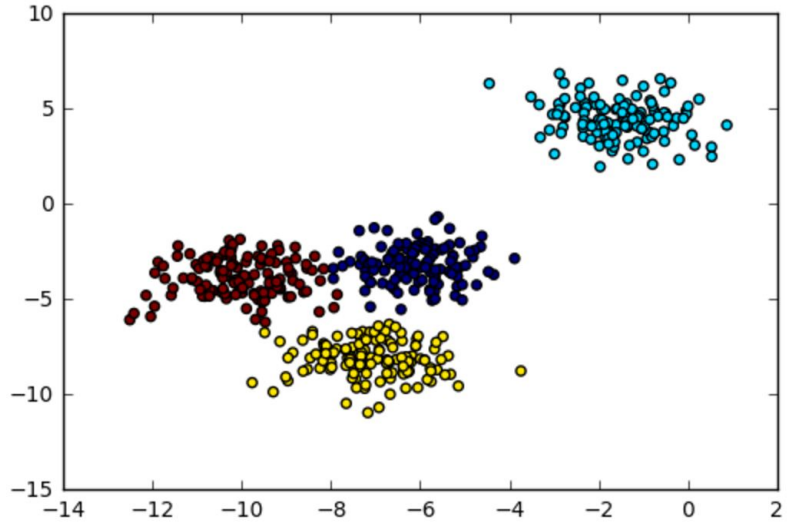
The silhouette plot shows that the `n_clusters` value of 3, 5 and 6 are a bad pick for the given data due to the presence of clusters with above average silhouette scores and also due to wide fluctuations in the size of the silhouette plots. Silhouette analysis is more ambivalent in deciding between 2 and 4.

總體來說，k=3, 5, 6 分群效果不佳，而 k=2, 4 還不錯，至於要選 2 還是 4，就看應用在什麼問題上了。

弱點二：不能處理 Noise data 與 Outliers

In [15]:
```
# Generate data.
# This particular setting has one distinct cluster and 3 clusters placed close together.
# (Same as the above example)
X, y = make_blobs(n_samples=500,
                  n_features=2,
                  centers=4,
                  cluster_std=1,
                  center_box=(-10.0, 10.0),
                  shuffle=True,
                  random_state=1)

# Perform k-means with n_clusters=4
kmeans = KMeans(n_clusters=4, random_state=10)
y_pred = kmeans.fit_predict(X)

# Plot the prediction
plt.scatter(X[:,0], X[:,1], c=y_pred)
```

Out[15]:
```
<matplotlib.collections.PathCollection at 0x1066c8e50>
```



一樣使用上面同樣的資料，並且使用 k=4 去分群，可以發現其實還是有些點離它們各自的 cluster 很遠，但是 k-means 依然將他們分群至 cluster 內。

那 … 我們要怎麼把這些離得比較遠的點，也就是所謂的 outliers 給偵測出來呢？

Solution: Use Distance Threshold to Detect Noise data and Outliers

However, we can detect the noises/outliers conditioning on whether the distance between the data point x_i and the centroid c_j of x_i 's corresponding cluster is larger than the average distance in the cluster. That is to say:

$$x_i = \begin{cases} \text{Outlier}, & \text{if } D(x_i, c_j) > \frac{1}{|Cluster_j|} \sum_{k=0,\, k \neq i}^{|Cluster_j|} D(x_k, c_j) \\ \text{Non-outlier}, & \text{otherwise} \end{cases} \quad \text{where } c_j \in Cluster_j$$

Let's begin to find out the outliers of each cluster.

你可以藉由計算每個 cluster 中的 average distance 當作判斷一個資料點是否是 outlier 的 distance threshold (有時候 distance threshold 還會再乘上一個 ratio 來控制 threshold 大小)。

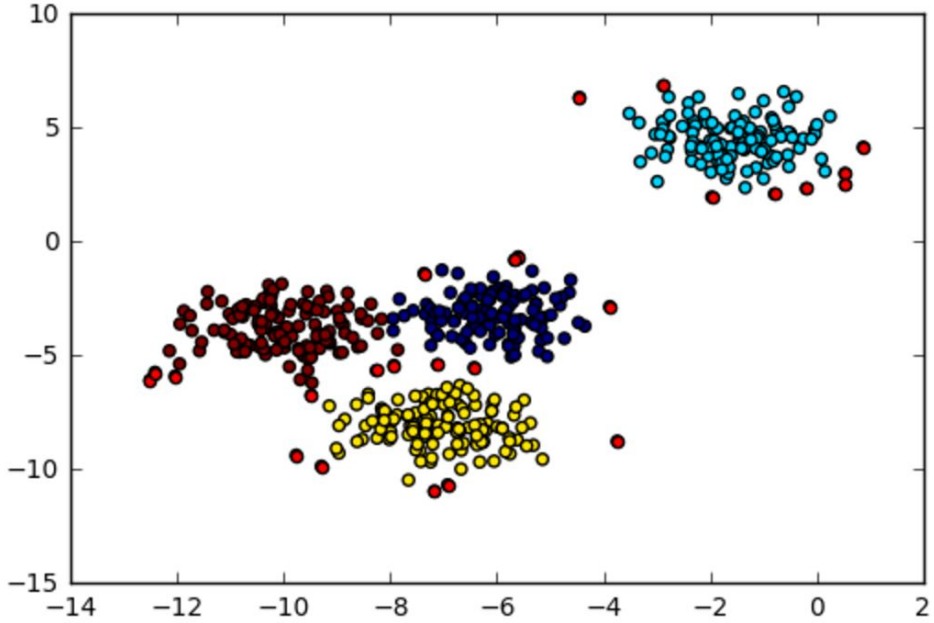
In [16]:

```
# Ratio for our distance threshold, controlling how many outliers we want to detect.
distance_threshold_ratio = 2.0

# Plot the prediction same as the above.
plt.scatter(X[:,0], X[:,1], c=y_pred)

# For each ith cluster, i=0~3 (we have 4 clusters in this example).
for i in [0, 1, 2, 3]:

    # Retrieve the indexs of data points belong to the ith cluster.
    # Note: `np.where()` wraps indexs in a tuple, thus we retrieve indexs using `tuple[0]`
    indexs_of_X_in_ith_cluster = np.where(y_pred == i)[0]

    # Retrieve the data points by the indexs
    X_in_ith_cluster = X[indexs_of_X_in_ith_cluster]

    # Retrieve the centroid.
    centroid = kmeans.cluster_centers_[i]

    # Compute distances between data points and the centroid.
    # Same as: np.sqrt(np.sum(np.square(X_in_ith_cluster - centroid), axis=1))
    # Note: distances.shape = (X_in_ith_cluster.shape[0], 1). A 2-D matrix.
    distances = metrics.pairwise.euclidean_distances(X_in_ith_cluster, centroid)

    # Compute the mean distance for ith cluster as our distance threshold.
    distance_threshold = np.mean(distances)

    # Retrieve the indexs of outliers in ith cluster
    # Note: distances.flatten() flattens 2-D matrix to vector, in order to compare with scalar `distance_threshold`.
    indexs_of_outlier = np.where(distances.flatten() > distance_threshold * distance_threshold_ratio)[0]

    # Retrieve outliers in ith cluster by the indexs
    outliers = X_in_ith_cluster[indexs_of_outlier]

    # Plot the outliers in ith cluster.
    plt.scatter(outliers[:,0], outliers[:,1], c='r')
```

← 1. 取得 cluster 內所有資料點，以及該 cluster 的 centroid。

← 2. 計算 cluster 內所有點到 centroid 的距離，並且平均起來，得出 distance threshold。

← 3. 找出資料點到 centroid 的距離如果大於 (distance threshold * ratio)，則視為 outliers。



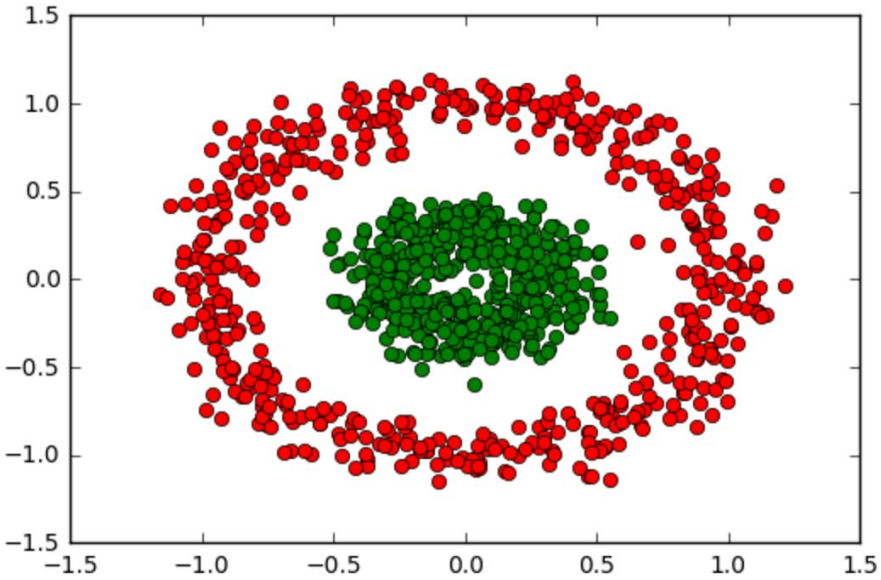
← 藉由上述方法，我們成功將 outliers（紅色）給偵測出來了！不過其實 值得提醒的一點是：當你的 cluster 之中佔有太多的 outliers，可想而知，k-means 分群出來得結果還是會不理想喔！因為在計算 distance threshold 時，還是會將這些 outliers 列入計算，因此大大影響了 distance threshold。

As we've mentioned about measuring cluster quality analysis, you can run different settings of `distance_threshold_ratio` to find out the best cluster quality.

弱點三：不能處理非球型分佈的資料集

In [17]:
```
# Generate non-spherical data.
X, y = make_circles(n_samples=1000, factor=0.3, noise=0.1)

# Plot the data distribution. (Here's another way to plot scatter graph)
plt.plot(X[y == 0, 0], X[y == 0, 1], 'ro')
plt.plot(X[y == 1, 0], X[y == 1, 1], 'go')
```

Out[17]:
```
[<matplotlib.lines.Line2D at 0x10f8e1fd0>]
```



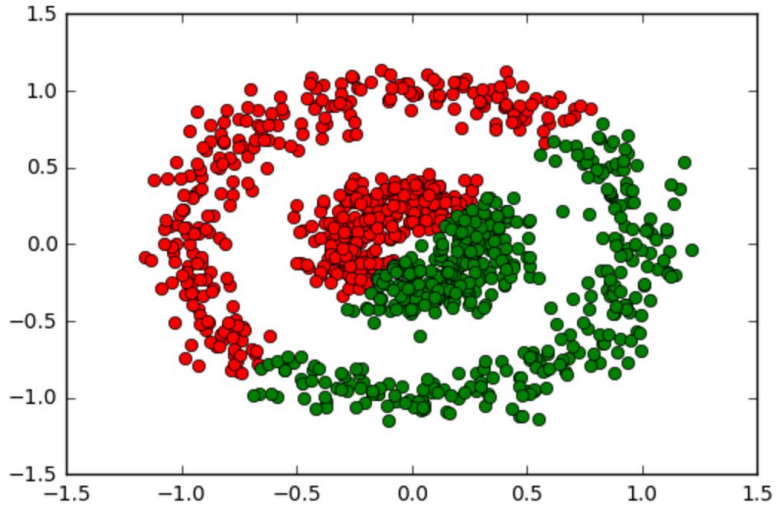
k-means 演算法根據其定義：此演算法目的是將資料點分成離它們最接近的 mean 值 的 cluster 之中。這個假設建立於你的資料分布必須得是球型分佈，因此如果我們的資料長得像左圖這種同心圓資料分佈，就會輕易的使 k-means 分群的效果不佳。

讓我們來看看當 k=2 時，k-means 會將資料分成什麼樣子。

After performing *K-means* on non-spherical data, the following result shows that it fails to cluster non-spherical data, since *K-means* has an assumption that the data distribution is spherical.

In [18]:
```
# Run k-means on non-spherical data.
y_pred = KMeans(n_clusters=2, random_state=170).fit_predict(X)

# Plot the predictions.
plt.plot(X[y_pred == 0, 0], X[y_pred == 0, 1], 'ro')
plt.plot(X[y_pred == 1, 0], X[y_pred == 1, 1], 'go')

# Print the evaluations
print('Homogeneity: {}'.format(metrics.homogeneity_score(y, y_pred)))
print('Completeness: {}'.format(metrics.completeness_score(y, y_pred)))
print('Mean Silhouette score: {}'.format(metrics.silhouette_score(X, y_pred)))
```

```
Homogeneity: 0.000184968858484
Completeness: 0.000185182645182
Mean Silhouette score: 0.295848480827
```



使用 k-means 分群後的效果並不佳，因為外圈與內圈的 mean 是差不多的，而 k-means 是基於 euclidean distance 來做分群，因此當 k=2 時，此同心圓有一半會跟其中一個 centroid 比較近，另一半會跟另一個 centroid 比較近，因此就被硬生生的砍成一半了。

Solution: Using Feature Transformation or Extraction Techiques Makes Data Clusterable

If you know that your clusters will always be concentric circles, you can simply convert your cartesian (x-y) coordinates to polar coordinates, and use only the radius for clustering - as you know that the angle theta doesn't matter.

Or more generally: use a suitable kernel for k-means clustering, e.g. use *Kernel PCA* to find a projection of the data that makes data linearly separable, or use another clustering algorithm, such as *DBSCAN*.

In [19]:

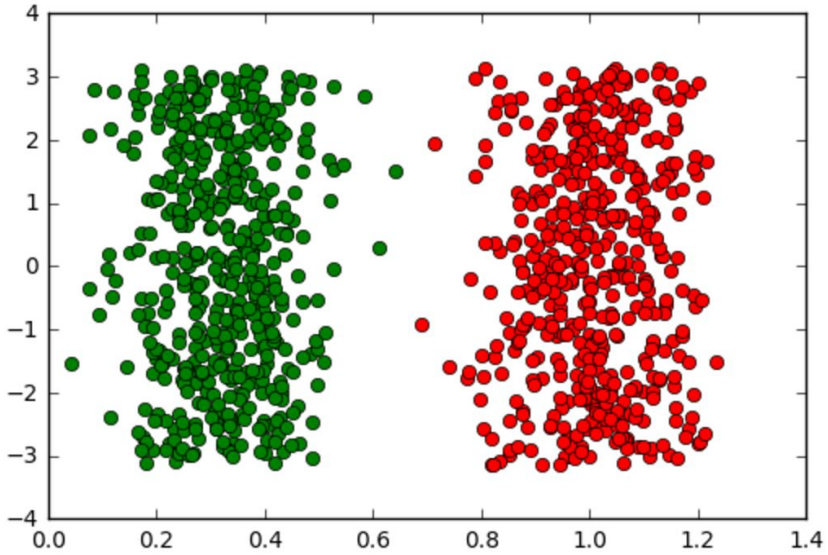
```python
def cart2pol(x, y):
    radius = np.sqrt(x**2 + y**2)
    theta = np.arctan2(y, x)
    return radius, theta

X_transformed = np.zeros_like(X)
# Convert cartesian (x-y) to polar coordinates.
X_transformed[:,0], _ = cart2pol(X[:,0], X[:,1])

# Only use `radius` feature to cluster.
y_pred = KMeans(n_clusters=2).fit_predict(X_transformed)

plt.plot(X[y_pred == 0, 0], X[y_pred == 0, 1], 'ro')
plt.plot(X[y_pred == 1, 0], X[y_pred == 1, 1], 'go')
```

← 將 Carteisan 座標轉換成 polar 座標的 function。

← 我們只看第一個 feature－radius，第二個 feature 就全部填 0，最後直接丟到 k-means 做分群。

In [38]:
```
def cart2pol(x, y):
    radius = np.sqrt(x**2 + y**2)
    theta = np.arctan2(y, x)
    return radius, theta

X_transformed = np.zeros_like(X)
X_transformed[:,0], X_transformed[:,1] = cart2pol(X[:,0], X[:,1])

plt.plot(X_transformed[y == 0, 0], X_transformed[y == 0, 1], 'ro')
plt.plot(X_transformed[y == 1, 0], X_transformed[y == 1, 1], 'go')
```

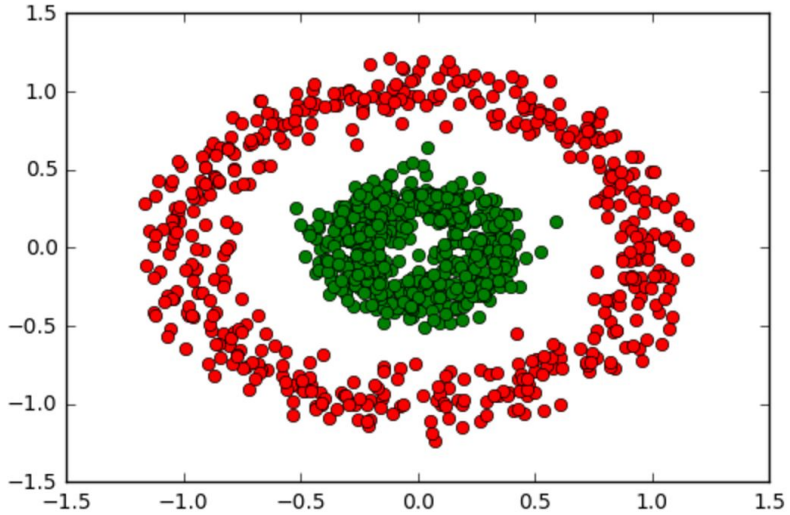
Out[38]:
```
[<matplotlib.lines.Line2D at 0x1163fe890>]
```



We just successfully make data linearly separable by converting features (x-y) to (radius-theta) !

In [39]:
```
def cart2pol(x, y):
    radius = np.sqrt(x**2 + y**2)
    theta = np.arctan2(y, x)
    return radius, theta

X_transformed = np.zeros_like(X)
# Convert cartesian (x-y) to polar coordinates.
X_transformed[:,0], _ = cart2pol(X[:,0], X[:,1])

# Only use `radius` feature to cluster.
y_pred = KMeans(n_clusters=2).fit_predict(X_transformed)

plt.plot(X[y_pred == 0, 0], X[y_pred == 0, 1], 'ro')
plt.plot(X[y_pred == 1, 0], X[y_pred == 1, 1], 'go')
```

Out[39]:
```
[<matplotlib.lines.Line2D at 0x1181a2490>]
```



Now the data is successfully clustered!

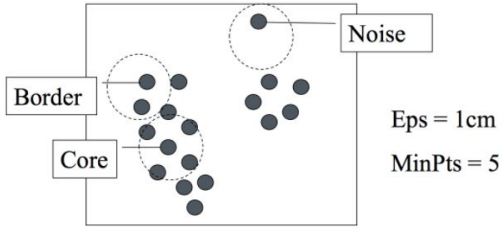
DBSCAN: Density-Based Spatial Clustering Algorithm with Noise

Parameters

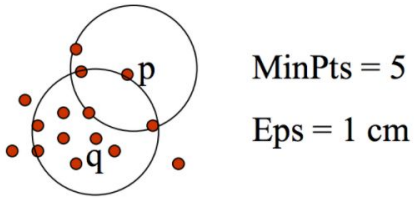
- Eps: Maximum radius of the neighborhood.
- $MinPts$: Minimum number of points in the Eps-neighborhood of a point.

Terms

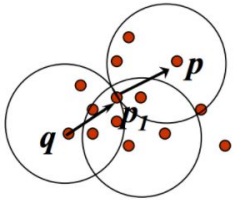
- The Eps-neighborhood of a point q – N_{Eps} : A point $p \in N_{Eps}(q)$ if $D(p, q) \leq Eps$. (Point inside the circle).
- Outlier: Not in a cluster.
- Core point: $|N_{Eps}(q)| \geq MinPts$ (dense neighborhood).
- Border point: In cluster but neighborhood is not dense.



- Directly density-reachable: A point p is **directly density-reachable** from a point q w.r.t Eps and $MinPts$ if:
  - $p \in N_{Eps}(q)$, and q is a **core point**.
  - p **doesn't** need to be a core point.



- Density-reachable: A point p is **density-reachable** from a point q w.r.t. Eps and $MinPts$ if there is a chain of points $p_1, \ldots, p_n$, $p_1 = q$, $p_n = p$ such that p_{i+1} is directly density-reachable from p_i.



The Algorithm

1. Randomly choose a point p.
2. Retrieve all points density-reachable from p w.r.t. Eps and $MinPts$.
3. If p is a core point, a cluster is formed.
4. If p is a border point, no points are density-reachable from p, then visit the next point.
5. Repeat the process until all the data points have been processed.

Let's begin to perform *DBSCAN* on spherical data

In [20]:

```
# Generate data with 3 centers.
X, y = make_blobs(n_samples=1000,
                  n_features=2,
                  centers=3,
                  random_state=170)

# Standardize features to zero mean and unit variance.
X = StandardScaler().fit_transform(X)

# Perform DBSCAN on the data
y_pred = DBSCAN(eps=0.3, min_samples=30).fit_predict(X)

# Plot the predictions
plt.scatter(X[:,0], X[:,1], c=y_pred)

# Print the evaluations
print('Number of clusters: {}'.format(len(set(y_pred[np.where(y_pred != -1)]))))
print('Homogeneity: {}'.format(metrics.homogeneity_score(y, y_pred)))
print('Completeness: {}'.format(metrics.completeness_score(y, y_pred)))
print('Mean Silhouette score: {}'.format(metrics.silhouette_score(X, y_pred)))
```

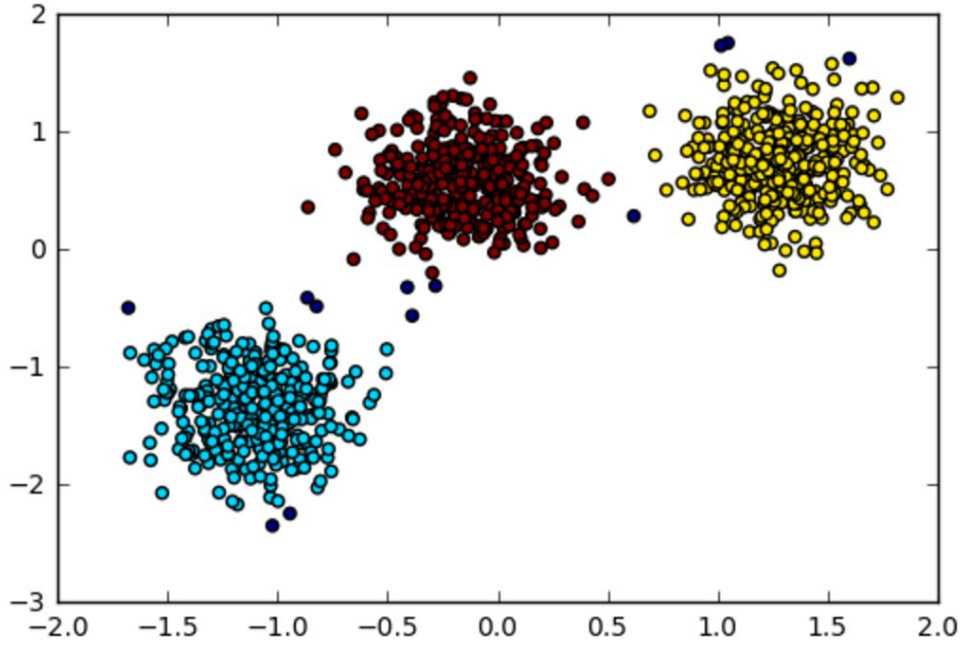
← 1. 產生 (1000x2) 的訓練資料 X 及對應 ground truth y。

← 2. 將資料 standardize 有助於避免某些維度因為 scaling 很大而影響 classification 或 clustering 的結果。

← 3. 使用 DBSCAN 來分群，其參數 ”min_samples” 等同前面所介紹的 “MinPts”。

← 4. 印出 cluster quality 指數

```
Number of clusters: 3
Homogeneity: 0.982039796605
Completeness: 0.93781096194
Mean Silhouette score: 0.679854880541
```



The **black** data points denote the **outliers** in the above result.

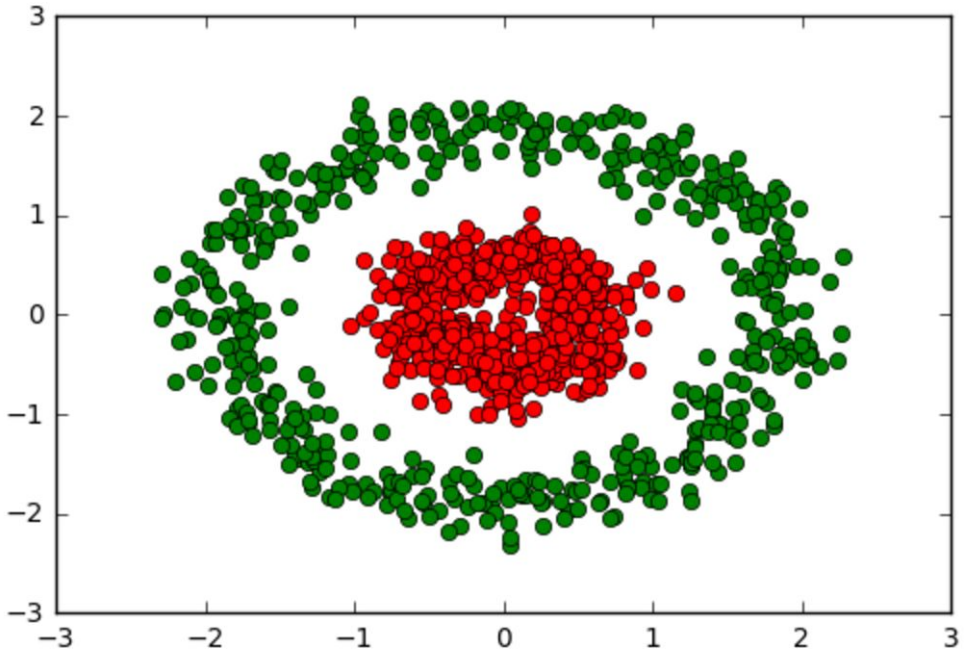
Note that we don't need to specify the number of clusters with *DBSCAN* algorithm. Besides, *DBSCAN* is good at finding out the outliers without requiring some hacks like we did above in *K-means* section.

讓我們來試試 DBSCAN 跑在非球狀資料分布上

Now, let's try *DBSCAN* on non-spherical data.

In [21]:
```
# Generate non-spherical data.
X, y = make_circles(n_samples=1000, factor=0.3, noise=0.1)

# Standardize features to zero mean and unit variance.
X = StandardScaler().fit_transform(X)

# Perform DBSCAN on the data
y_pred = DBSCAN(eps=0.3, min_samples=10).fit_predict(X)

# Plot the data distribution. (Here's another way to plot scatter graph)
plt.plot(X[y_pred == 0, 0], X[y_pred == 0, 1], 'ro')
plt.plot(X[y_pred == 1, 0], X[y_pred == 1, 1], 'go')

# Print the evaluations
print('Number of clusters: {}'.format(len(set(y_pred[np.where(y_pred != -1)]))))
print('Homogeneity: {}'.format(metrics.homogeneity_score(y, y_pred)))
print('Completeness: {}'.format(metrics.completeness_score(y, y_pred)))
print('Mean Silhouette score: {}'.format(metrics.silhouette_score(X, y_pred)))
```

```
Number of clusters: 2
Homogeneity: 0.996390359526
Completeness: 0.957691789224
Mean Silhouette score: 0.174843819318
```



Comparing to *K-means*, we can directly apply *DBSCAN* on this form of data distribution due to the density-based clustering criterion.

Note: It's worth mention that the *Silhouette score* is generally higher for **convex** clusters than other concepts of clusters, such as density based clusters.

請完成 “clustering-lab2.ipynb”

Thanks for Listening.
Any Questions ?

Reference

- scikit-learn
- How to understand the drawbacks of K-means
- The Sihouette Coefficient
- Cluster Analysis in Data Mining tutorial video series
- DBSCAN