Clustering Algorithms: K-means and DBSCAN

Yu-Chun Lo
howard.lo@nlplab.cc

Preface

The content of this slides is same as Ipython provided in GitHub. The only difference is that I've added some easy-read comments and explanations along with the code. Some tedious and unimportant code will not put on the slides, so I would suggest that while reading this tutorial, play with the code in Ipython.

Let's get started!

K-means

K-means

The Algorithm

1. Randomly choose k centroids $C = \{c_1, c_2, \ldots, c_k\}$ from the data points $X = \{x_1, x_2, \ldots, x_n\} \in \mathbb{R}^D$.
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).

First, import the necessary packages.

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
```

Generate three clusters with two-dimensional data.

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]])

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

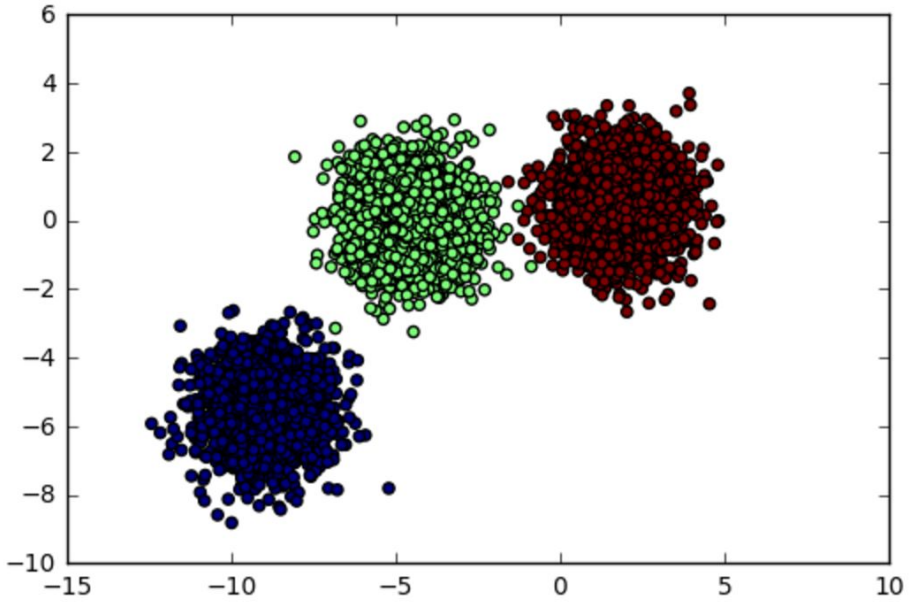
← The row and column of numpy array indicate t**he number of data points** and **the number of dimensions**. In this case, we generate **5000** data points and each data point is **two-dimensional** data, so the training data **X** is a **(5000 x 2)** matrix.

← Each data point corresponds to a cluster label **y** (ground truth) and the number of clusters is three (label = {0, 1, 2}), so the label **y** is a **5000-dimensional** vector.

Let’s visualize our training data and ground truth.

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
```

Use **KMeans** from **scikit-learn** to train our clustering model:

Parameters:

- **n_clusters**: How many clusters you want (choosing **"k"**).
- **n_inits**: How many times do you want *k-means* to perform. In this case, we set ***n_init = 3***, this means that we want *k-means* **run three times** and **choose the lowest "inertia"** (objective/cost function, sum squared distance). The Lowest inertia means the "position" in vector space of the centroids are the best for clustering, since the average distance of the data points to their belonging centroids are closest.
- **init**: How *k-means* initialize centroids. We've got **"random"** and "**k-means++"** to choose (we'll leave it behind to explain).
- **tol**: The distance threshold ratio which decides when does *k-means converge. The higher the ratio is, the earlier k-means converges.*
- **random_state**: Just for reproducibility. The same value produces the same result.
- **verbose**: Set to **True** to print the logs of training process of *k-means* and vice versa.

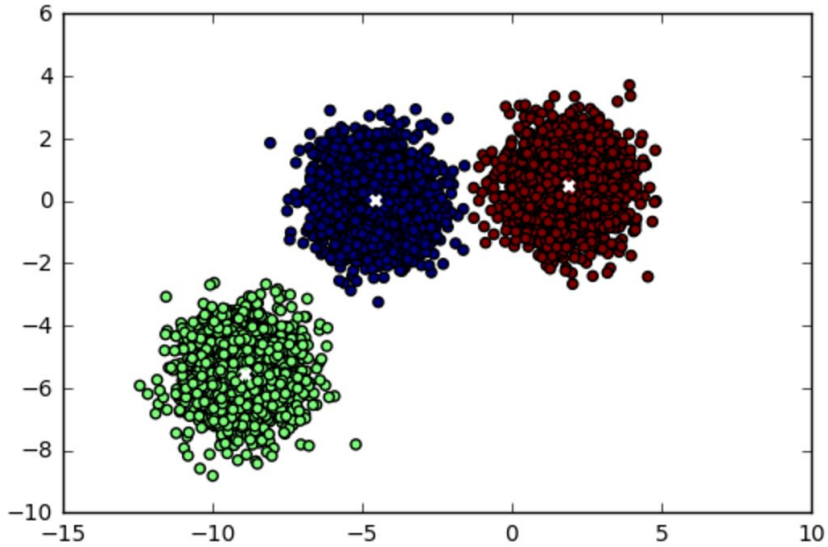
In [5]:
```
# Retrieve predictions and cluster centers (centroids).
display(kmeans.labels_)
display(kmeans.cluster_centers_)
```

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

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

← Use **“kmeans.labels_”** to retrieve our predictions (5000-D vector).
← Use **“kmeans.cluster_centers_”** to retrieve trained centroids.
There are **three clusters** and the data is **2-D**, thus it is **(3x2)** matrix.

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>
```



← Visualize predictions and centroids.

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)
```

← After training, use **"kmeans.predict()"** to predict new data **X_new** without training again.

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]])
```

← Use "**kmeans.transform()"** to retrieve the distances between every data point and every centroid. If you have **M** data points and **N** clusters, you'll have **(MxN)** matrix. Every row contains distances from a data point to **N** clusters.

K-means++

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.

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
```

If you want to use *k-means++*, just set *init='k-means++'*, simple enough huh? Besides, have you noticed that the speed of convergence is faster than just randomly picking centroids?

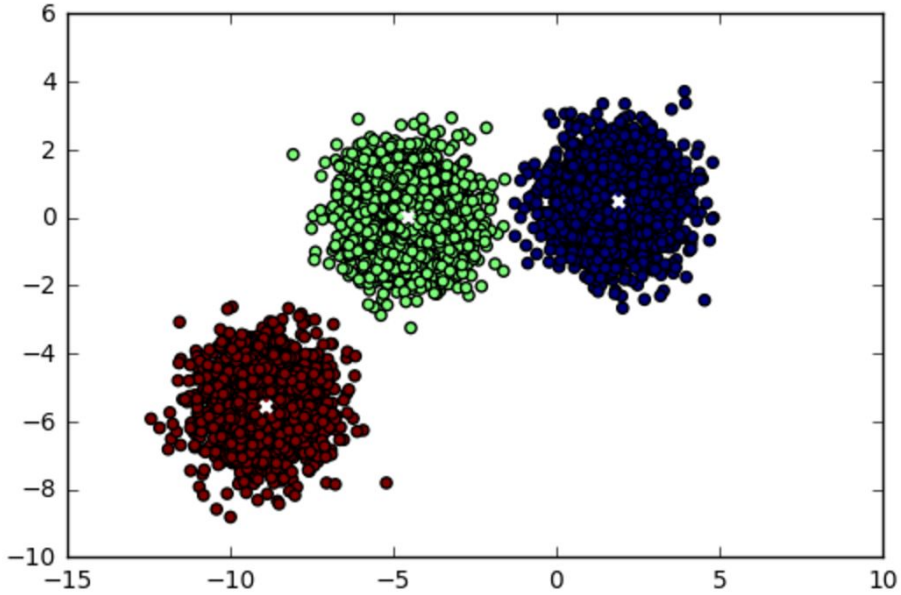
Actually in scikit-learn, it use *k-means++* as the default option, so you don't need to set it manually (the reason why we set it manually is just for educational purpose).

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

The predictions from *k-means++*.

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>



Please complete “clustering-lab1.ipynb”

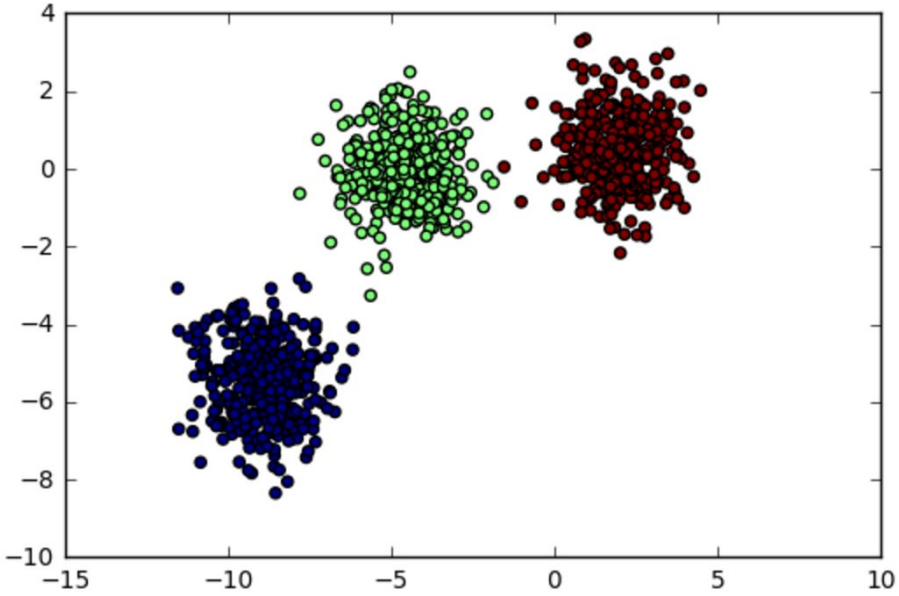
Drawbacks of K-means and How to Address Them

Drawback 1:
Need to choose the right “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>
```

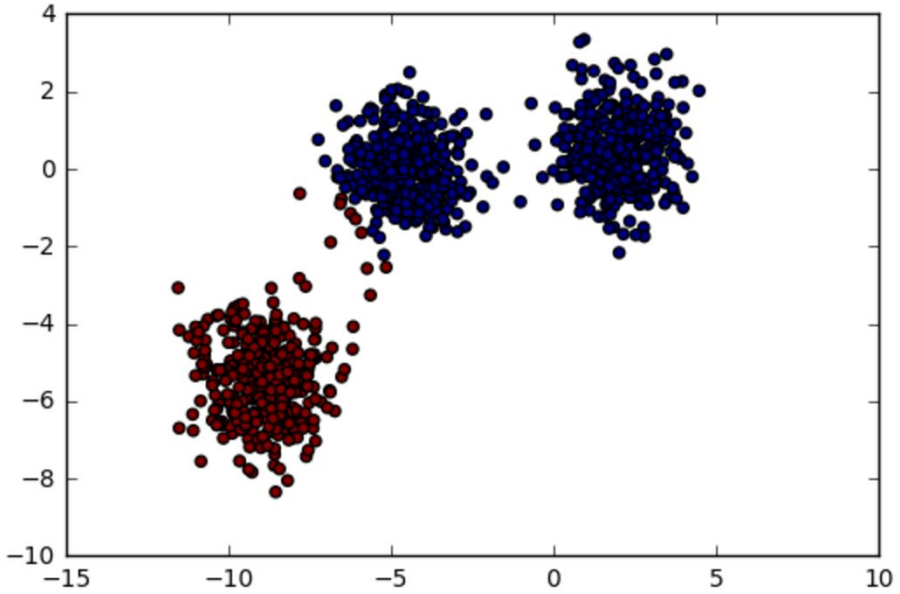


Let's begin by generating sythesized training data consisted of 3 clusters.

Here's a question: What if we choose a wrong number of clusters and directly apply *k-means* on the training data?

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>



The answer is, if we choose the wrong “k”, the *k-means* algorithm can still work without any error.

In the case of k=2, the cluster label of the center one among these clusters is viewed as the same as the right one (blue), which means that the overall distances between the center one is more closer to the right one (blue), instead of the left (red) one.

However, this result is not good enough. How do we choose “k” properly?

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.

Use supervised method only when you have ground truth with your training data.

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)

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

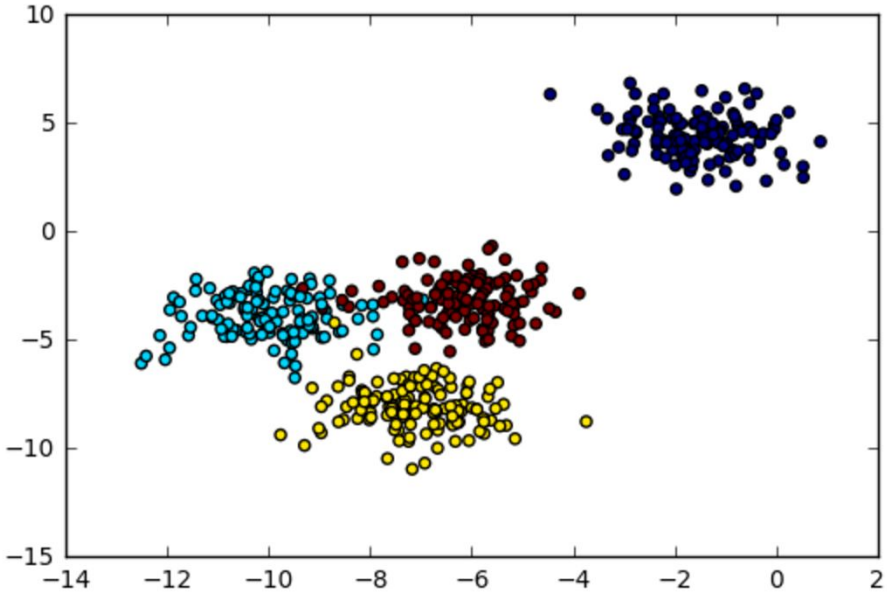
For the data point x_i and all the other clusters not containing x_i, caculate its average distance to all the other data points in the given clusters. Find the minimum distance value with respect to the given clusters.

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 $s(x_i) \to 1$.

Use unsupervised method when you don’t have ground truth with your training data.

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>



First, we generate 4-cluster training data and visualized it. Note that we mean to generate the data that there're 3 clusters (red, blue and yellow) are overlapped with each other on purpose.

Let's use the cluster quality measurements that just described above and run different values of "k" to observe which "k" is most preferable.

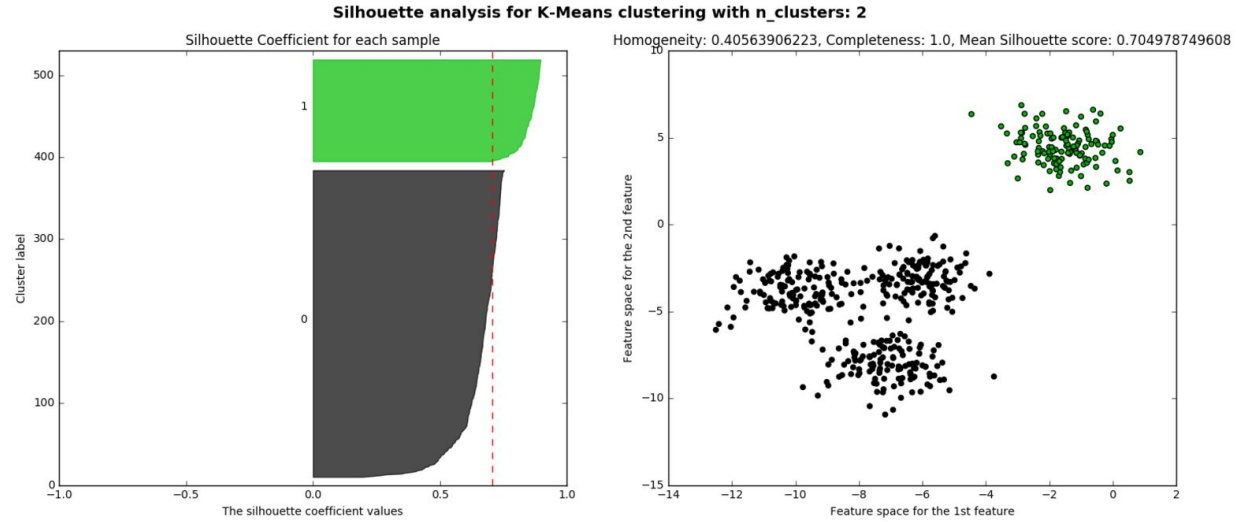
In [14]:
```
# List of number of clusters
range_n_clusters = [2, 3, 4, 5, 6]

# 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)
```

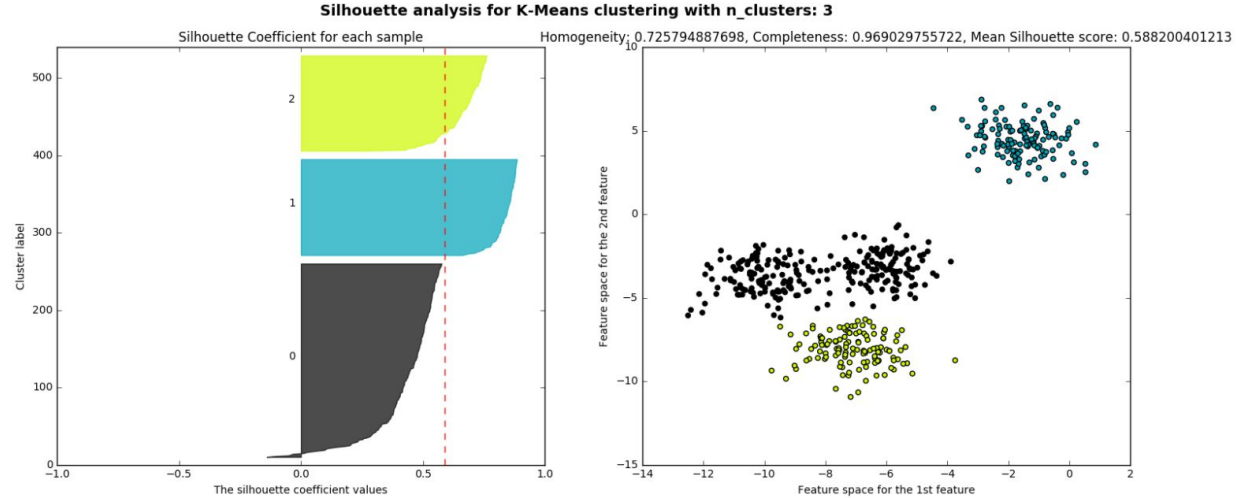
Code explanation:

- First, we set the range of “k”. In this case, we run 5 cases on **k=2, 3, 4, 5, 6** simply by using for loop. Each uses these 3 metrics: **homogeneity**, **completeness** and **silhouette coefficient** to evaluate the goodness of the clusters.

- Use **“homogeneity_score(y, y_pred)”** and **“completeness_score(y, y_pred)”** to compute the corresponding evaluations.

- Note that we have 2 options to compute silhouette coefficient in scikit-learn:
  - Use **“silhoette_samples(X, y_pred)”** to compute silhouette coefficient for each data point.
  - Use **“silhoette_score(X, y_pred)”** to compute the average silhouette coefficient of all data points.





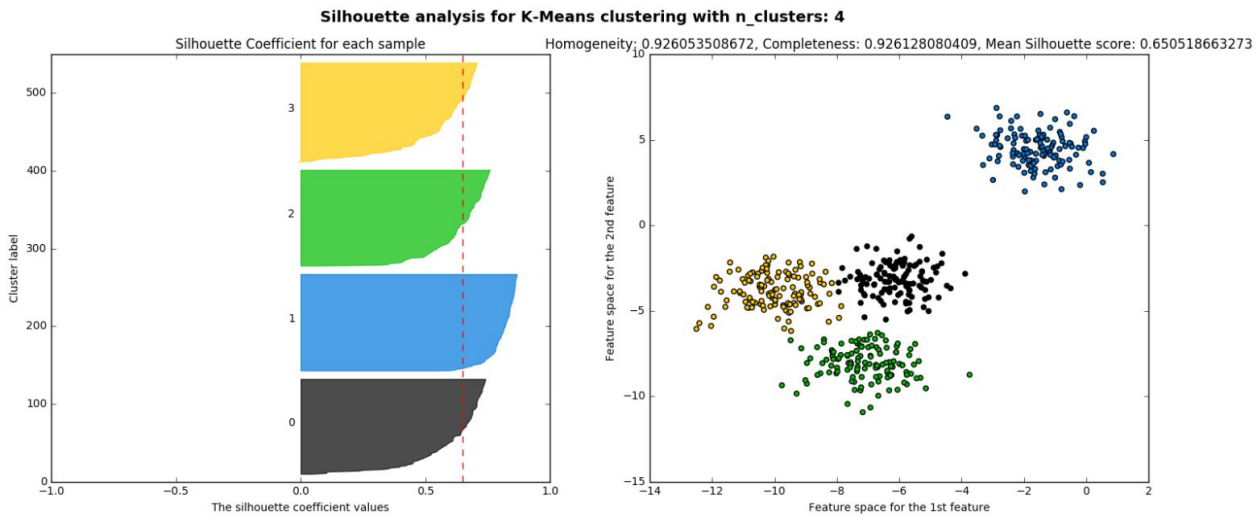
Since the code of visualization part is tedious, please refer it to Ipython.

Let's talk about how to interpret this diagram.

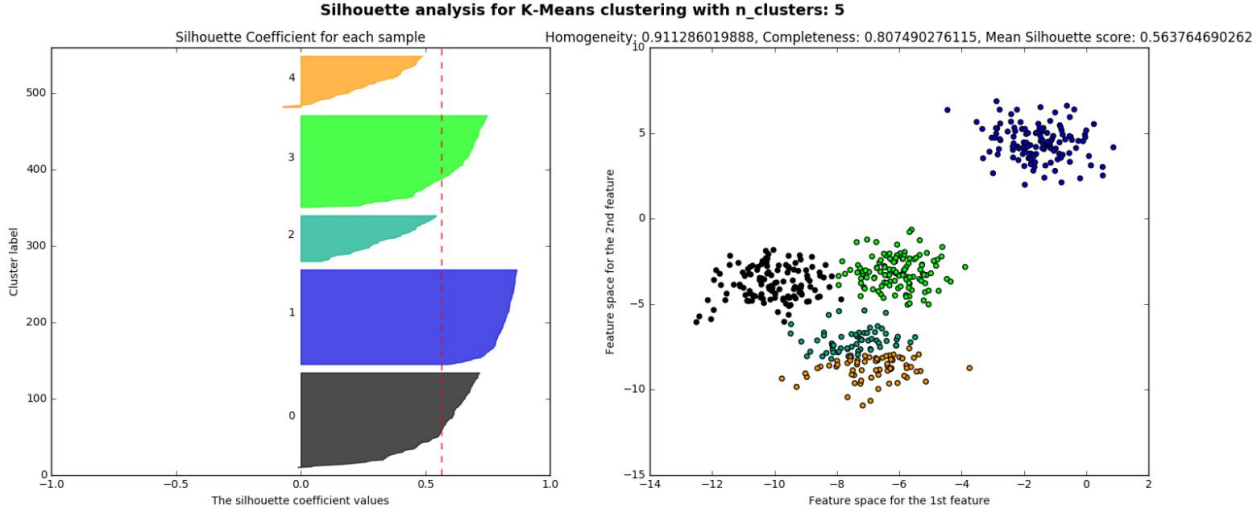
The left one is very simple, just the predictions of the clusters.

The right one:

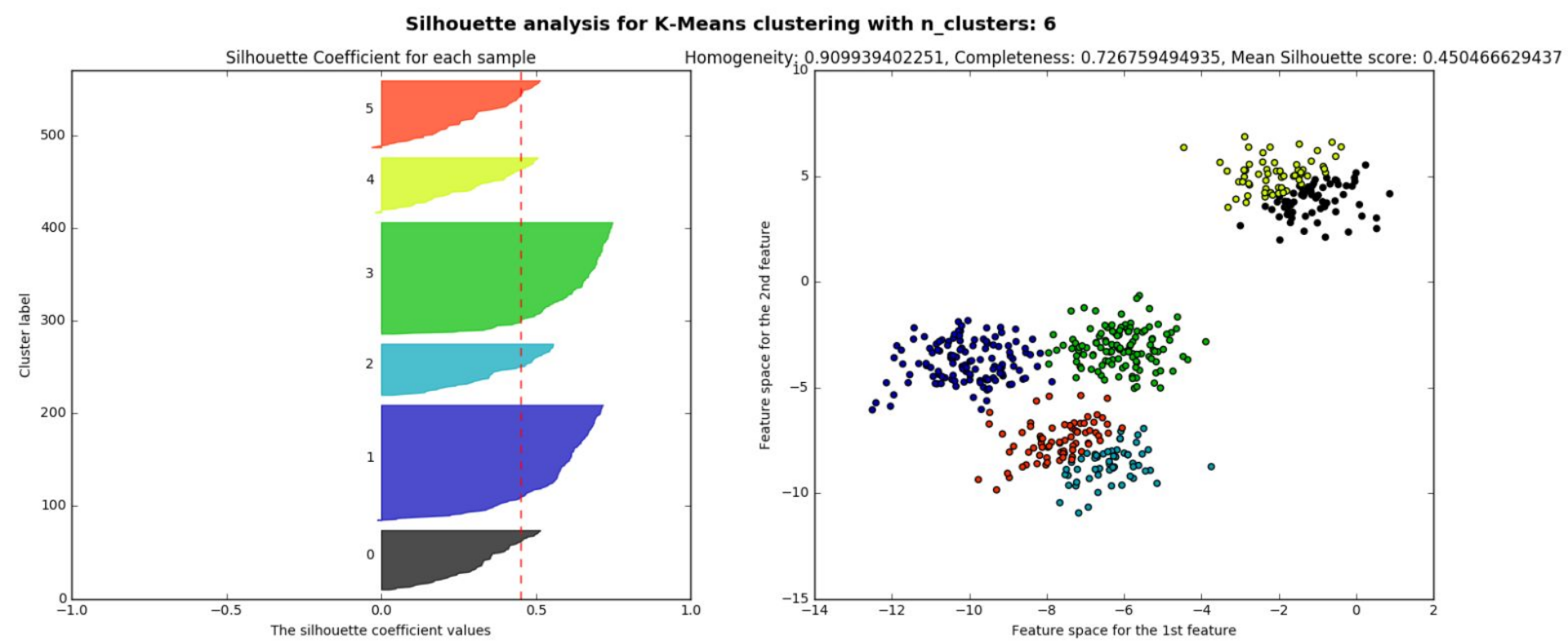
- X-axis: silhouette coefficient.
- Y-axis: data points.
- Data points that is in the same cluster will be plotted with the same color.
- The silhouette cofficient of each cluster is sorted in decending order.
- The red dashed line represents the mean silhouette coefficient computed by "**silhouette_score()**".



← When k=4, the silhouette coefficient of all cluster are generally beyond the red dashed line, meaning that these cluster qualities are good.



← When k=5, we can notice that there're 2 clusters (yellow and tiffany blue) are behind the red dashed line, meaning that their overall silhouette coeffiecients are low, since these 2 clusters are overlapped with each other. This results in low mean nearest cluster distance b(x), thus k=5 is not a good choice.

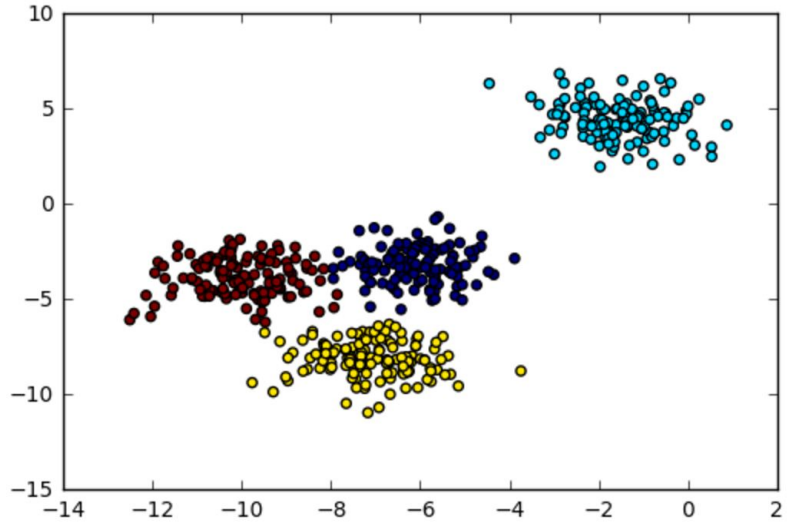


The silhouette plot shows that the `n_clusters` value of 3, 5 and 6 are a bad pick for the given data due to the presence of clusters with above average silhouette scores and also due to wide fluctuations in the size of the silhouette plots. Silhouette analysis is more ambivalent in deciding between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

Drawback 2:
Cannot Handle Noise Data and 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>
```



Extended from the same example above, we cluster the data with k=4 and the data seems to be well-clustered.

We can notice that some of the data points which are relatively far away from their belonging clusters, however, are still clustered by *k-means*, but they are obviously “outliers” .

So … how do we detect these outliers?

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.

You can simply compute the average distance for each cluster as the **distance threshold** for judging whether a data point is an outlier or not if the distance between the data point to its cluster center is less than the distance threshold (sometimes we control the scale of the distance threshold by multiplying a **distance threshold ratio**).

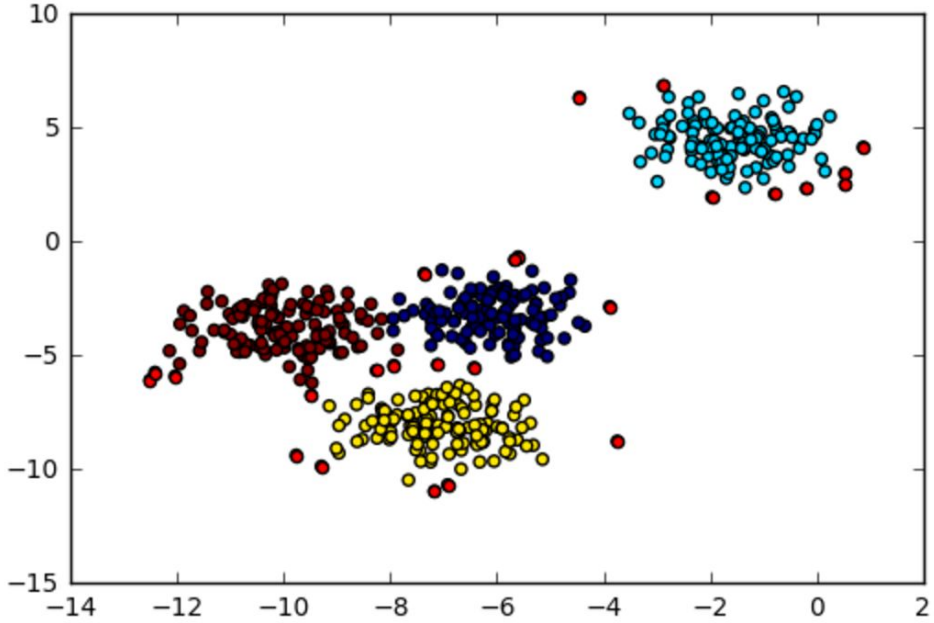
In [16]:

```
# 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. Retrieve the data points and centroid.

← 2. Compute all distances between every data points and their centroid. Average them to get our distance threshold.

← 3. Outlier detection.



← By doing the simple outlier detection we described above, we successfully detect the outliers (red).

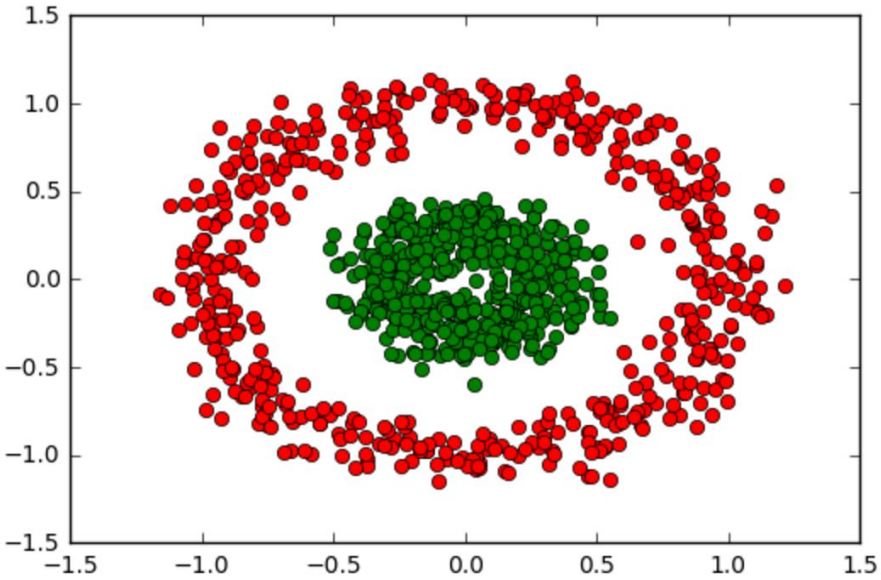
However, if there’re are too many outliers in your dataset, this method may not work appropriately, since *k-means* computes mean from all the data points, including these outliers.

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.

Drawback 3: Cannot Handle Non-spherical Data

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>]
```



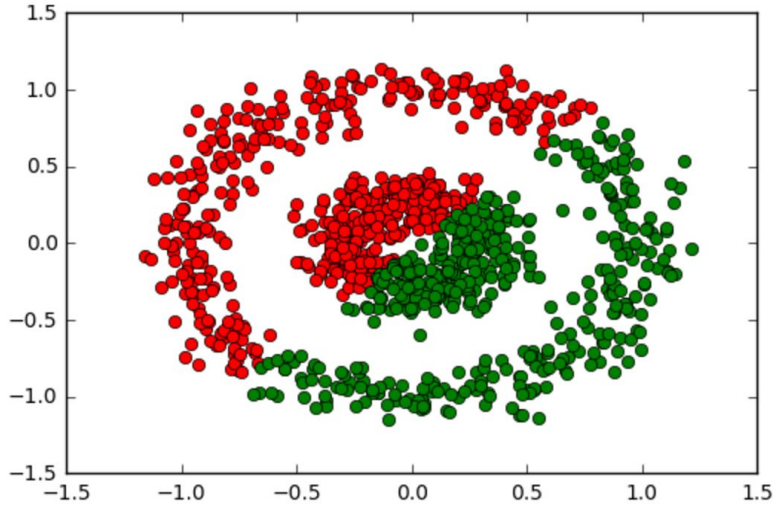
According to the definition of *k-means*: The algorithm aims to partition the data to the closest mean among the clusters. This is based on an assumption that your data has to be spherical, otherwise, the performance of *k-means* won't be desirable.

Let's see what will the clustering result be when we use *k-means* with k=2 to cluster the non-spherical data at the left hand side.

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
```



It turns out that the concentric-circle data is partitioned from the center into 2 clusters due to the mean values of the inner circle and the outer circle are very closed to each other.

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]:

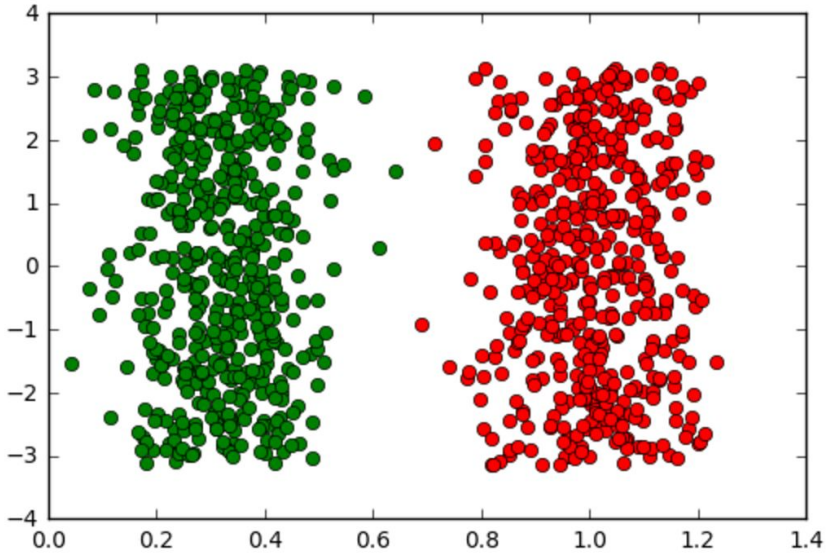
```
def cart2pol(x, y):
    radius = np.sqrt(x**2 + y**2)
    theta = np.arctan2(y, x)
    return radius, theta

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')
```

← Helper function for transforming Carteisan coordinates to polar coordinates.

← We only need the feature—radius, the second feature will be filled in zero, then finally apply *k-means*.

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')
```

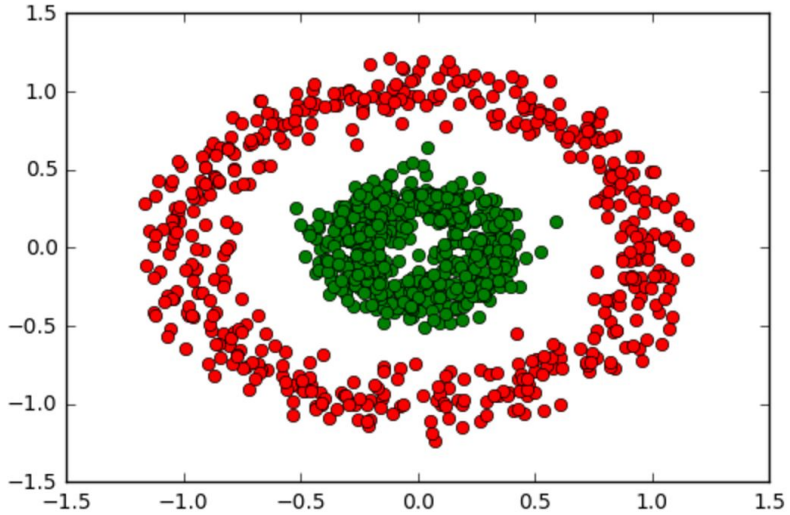
Out[38]:
```
[<matplotlib.lines.Line2D at 0x1163fe890>]
```



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!

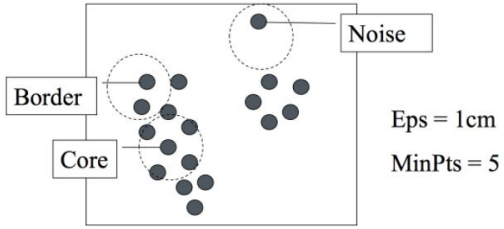
DBSCAN: Density-Based Spatial Clustering Algorithm with Noise

Parameters

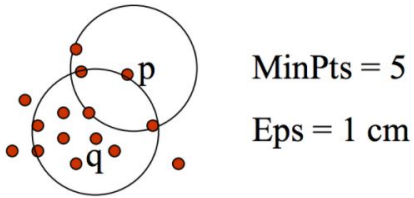
- Eps: Maximum radius of the neighborhood.
- $MinPts$: Minimum number of points in the Eps-neighborhood of a point.

Terms

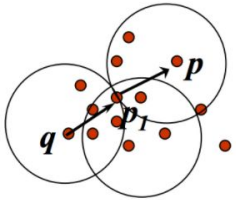
- The Eps-neighborhood of a point q – N_{Eps} : A point $p \in N_{Eps}(q)$ if $D(p, q) \leq Eps$. (Point inside the circle).
- Outlier: Not in a cluster.
- Core point: $|N_{Eps}(q)| \geq MinPts$ (dense neighborhood).
- Border point: In cluster but neighborhood is not dense.



- 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)))
```

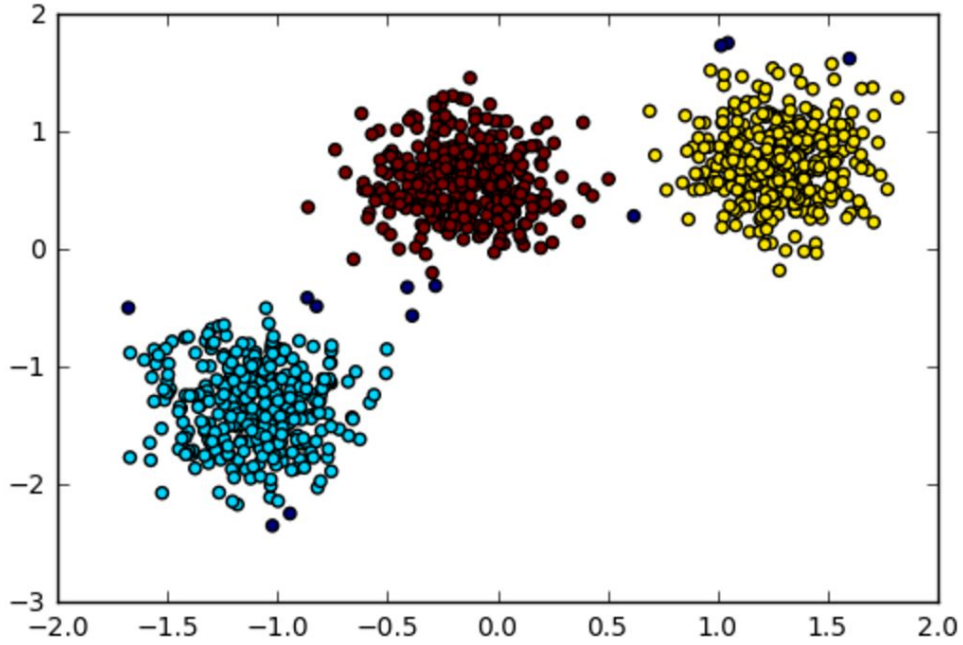
← 1. Generate training data X (1000x2) and corresponding ground truth y。

← 2. Standardizing features of training data.

← 3. Apply *DBSCAN*, the parameter ”min_samples” is same as “MinPts” described previously.

← 4. Print cluster qualities.

```
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.

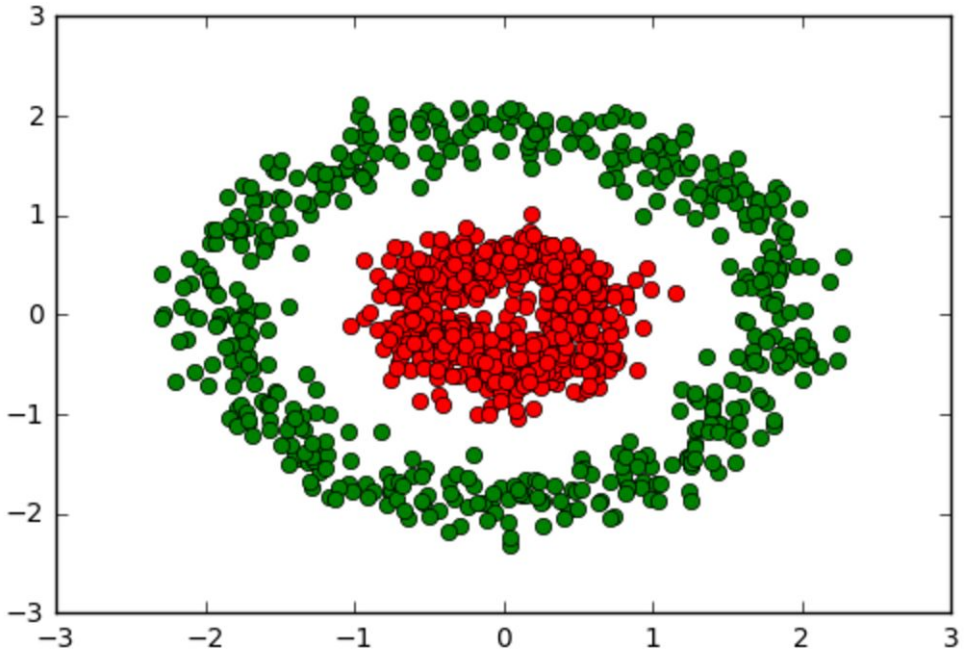
Note that we don't need to specify the number of clusters with *DBSCAN* algorithm. Besides, *DBSCAN* is good at finding out the outliers without requiring some hacks like we did above in *K-means* section.

Let’s try to apply *DBSCAN* on non-spherical data.

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.

Please complete “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