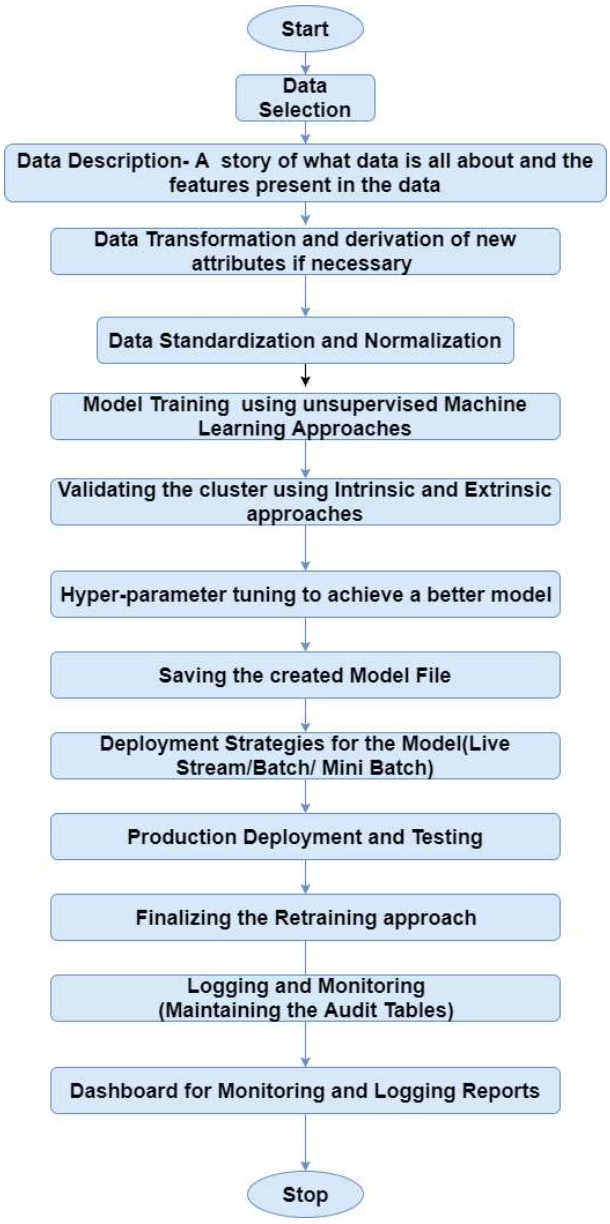
Application Flow

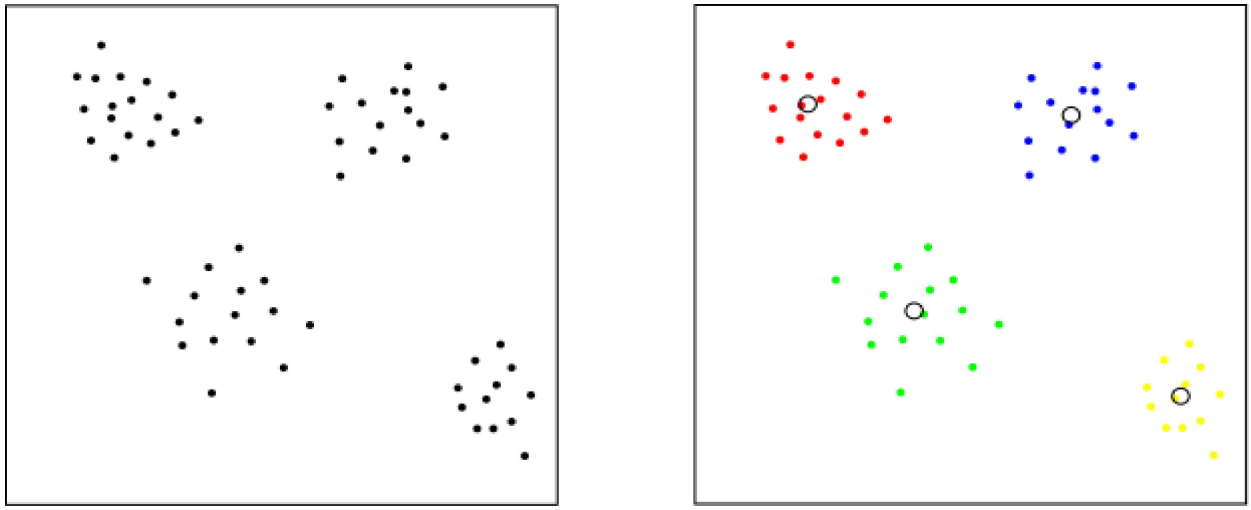
Before proceeding with the algorithm, let's first discuss the lifecycle of an unsupervises machine learning model. This diagram explains the creation of an unsupervised Machine Learning model from scratch and then taking the same model further with hyperparameter tuning to increase its accuracy, deciding the deployment strategies for that model and once deployed setting up the logging and monitoring frameworks to generate reports and dashboards based on the client requirements. A typical lifecycle diagram for an unsupervised machine learning model looks like:



Clustering

Let's suppose we give a child different objects to group. How does a child make a group? The child may group over the colour, over the shape, over the hardness or softness of the objects etc. The basic idea here is that the child tries to find out similarities and dissimilarities between different objects and then tries to make a group of similar objects. This is called **clustering**, the method of identifying similar instances and keeping them together. In Other words, clustering identifies homogeneous subgroups among the observations.

Clustering is an unsupervised approach which finds a structure/pattern in a collection of unlabeled data. A cluster is a collection of objects which are “similar” amongst themselves and are “dissimilar” to the objects belonging to a different cluster. For example:



In the figure above, we can easily identify 4 different clusters. The clustering criteria here is distance. Whichever points are near to each other are kept in the same cluster and the faraway points belong to a different cluster

The Goal of Clustering

The goal of clustering is to determine the intrinsic groups in unlabeled data. The question is: what constitutes a good cluster? It can be shown that there is no absolute “best” criterion for cluster validation. Consequently, it is the user who must supply the criterion for validating the cluster. For example, we might be interested in finding representatives of homogeneous instances for finding the “natural clusters” and identifying their unknown properties (like “natural” data types), for finding appropriate groupings or in finding unusual(which are different from all other data) data objects (outlier detection).

Applications

The scikit-learn book describes the various applications of clustering as follows:

- **For customer segmentation:** You can cluster your customers based on their purchases,their activity on your website, and so on. This is useful to understand who your customers are and what they need, so you can adapt your products and marketing campaigns to each segment. For example, this can be useful in recommender systems to suggest content that other users in the same cluster enjoyed.
- **For data analysis:** When analyzing a new dataset, it is often useful to first discover clusters of similar instances, as it is often easier to analyze clusters separately.
- **As a dimensionality reduction technique:** Once a dataset has been clustered, it is usually possible to measure each instance’s affinity with each cluster (affinity is any measure of how well an instance fits into a cluster). Each instance’s feature vector x can then be replaced with the vector of its cluster affinities. If there are k clusters, then this vector is k dimensional. This is typically much lower dimensional than the original feature vector, but it can preserve enough information for further processing.
- **For anomaly detection (also called outlier detection):** Any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if you have clustered the users of your website based on their behavior, you can detect users with unusual behavior, such as an unusual number of requests per second, and so on. Anomaly detection is particularly useful in detecting defects in manufacturing, or for fraud detection.

- **For semi-supervised learning:** If you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster. This can greatly increase the amount of labels available for a subsequent supervised learning algorithm, and thus improve its performance. .
- **For search engines:** For example, some search engines let you search for images that are similar to a reference image. To build such a system, you would first apply a clustering algorithm to all the images in your database: similar images would end up in the same cluster. Then when a user provides a reference image, all you need to do is to find this image's cluster using the trained clustering model, and you can then simply return all the images from this cluster.
- **To segment an image:** By clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to reduce the number of different colors in the image considerably. This technique is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object.

Main Requirements:

The primary requirements that should be met by a clustering algorithm are:

- It should be scalable
- It should be able to deal with attributes of different types;
- It should be able to discover arbitrary shape clusters;
- It should have an inbuilt ability to deal with noise and outliers;
- The clusters should not vary with the order of input records;
- It should be able to handle data of high dimensions.
- It should be easy to interpret and use.

Approaches for Clustering:

The clustering approaches can be broadly divided into two categories: *Agglomerative* and *Divisive*.

Agglomerative: This approach first considers all the points as individual clusters and then finds out the similarity between two points, puts them into a cluster. Then it goes on finding similar points and clusters until there is only one cluster left i.e., all points belong to a big cluster. This is also called the bottom-up approach.

Divisive: It is opposite of the agglomerative approach. It first considers all the points to be part of one big cluster and in the subsequent steps tries to find out the points/ clusters which are least similar to each other and then breaks the bigger cluster into smaller ones. This continues until there are as many clusters as there are datapoints. This is also called the top-down approach.

K-Means Clustering

It was proposed by Stuart Lloyd at the Bell Labs in 1957 as a technique for pulse-code modulation, but it was only published outside of the company in 1982, in a paper titled _"Least square quantization in PCM"._By then, in 1965, Edward W. Forgy had published virtually the same algorithm, so K-Means is sometimes referred to as Lloyd-Forgy.

K-Means is a clustering approach in which the data is grouped into K distinct non-overlapping clusters based on their distances from the K centres. The value of **K** needs to be specified first and then the algorithm assigns the points to exactly one cluster.

Theory

The theory discussed above can be mathematically expressed as:

- Let C1, C2, Ck be the K clusters
- Then we can write: $C_1 U C_2 U C_3 U \ldots U C_k$ ={1,2,3,…,n} i.e., each datapoint has been assigned to a cluster.
- Also,

$$C_k \cap C_{k'} = \emptyset \text{ for all } k \neq k'.$$

  This means that the clusters are non-overlapping.
- The idea behind the K-Means clustering approach is that the within-cluster variation amongst the point should be minimum. The within-cluster variance is denoted by: W(Ck).Hence, according to the statement above, we need to minimize this variance for all the clusters. Mathematically it can be written as:

$$\underset{C_1,\ldots,C_K}{\text{minimize}} \left\{ \sum_{k=1}^{K} W(C_k) \right\}.$$

- The next step is to define the criterion for measuring the within-cluster variance. Generally, the criterion is the Euclidean distance between two data points.

$$W(C_k) = \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2,$$

- The above formula says that we are calculating the distances between all the point in a cluster, then we are repeating it for all the K clusters(That's why two summation signs) and then we are dividing it by the number of observation in the clusters (Ck is the number of observations in the Kth cluster) to calculate the average.

So, ultimately our goal is to minimize:

$$\underset{C_1,\ldots,C_K}{\text{minimize}} \left\{ \sum_{k=1}^{K} \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2 \right\}.$$

The following algorithm steps are used to solve this problem

Algorithm:

1. Randomly assign K centres.
2. Calculate the distance of all the points from all the K centres and allocate the points to cluster based on the shortest distance. The model's *inertia* is the mean squared distance between each instance and its closest centroid. The goal is to have a model with the lowes intertia.
3. Once all the points are assigned to clusters, recompute the centroids.
4. Repeat the steps 2 and 3 until the locations of the centroids stop changing and the cluster allocation of the points becomes constant.

Python Explanation:

In [2]:

```
# Doing the necessary imports
%pylab inline
from pylab import *
import random as pyrandom
from scipy.spatial.distance import cdist
matplotlib.rc("image",cmap="gray")
from collections import Counter
```

```
Populating the interactive namespace from numpy and matplotlib
```

Consider a collection of points that are sampled from three different densities, in this case normal densities with the same covariances but different means.

In [3]:

```
data = r_[10*randn(1000,2)+array([70,30]),
          10*randn(1000,2)+array([10,10]),
          10*randn(1000,2)+array([50,80])]
data = data[pyrandom.sample(range(len(data)),len(data))]
```

Here is a scatterplot of this data.

We clearly see three *clusters_, corresponding to the three _mixture components*. How can we recover these clusters?

This is the job of *clustering algorithms*.

In [4]:

```
figsize(10,10)
plot(data[:,0],data[:,1],'b+')
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x2984fd69b70>]
```

Mixture densities arise in both unsupervised learning and in supervised learning. In both cases, they commonly represent a problem structure in which data is generated from a number of ideal prototypes (the cluster centres) but then corrupted by noise.

- When each cluster has a distinct class label, we have a regular classification problem with normal densities.
- When no cluster has any labels, we can hope to recover the underlying clusters with a clustering algorithm and then assign labels to these clusters. This is a form of *semi-supervised learning*.
- When there is training data with labels available, often each class is a mixture of multiple clusters. That is, each class is generated by multiple prototypes (think characters in different fonts).

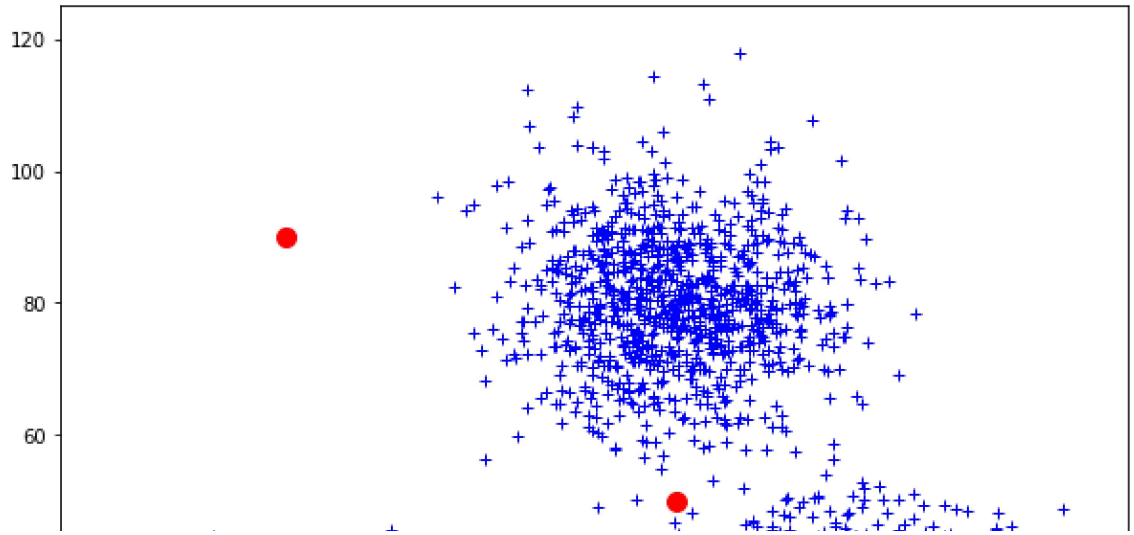
You can perform clustering either at the class level or across all samples and then label each cluster with its corresponding class label.

In [5]:

```
protos = array([[30,30],[40,20],[0,90],[50,50]]) # Initialize centroids
start = protos.copy()

figsize(10,10)
plot(data[:,0],data[:,1],'b+')
plot(protos[:,0],protos[:,1],'ro',markersize=10)
```

Out[5]:

```
[<matplotlib.lines.Line2D at 0x2984fdd4cc0>]
```



Obviously, those centers are wrong, but let's keep going.

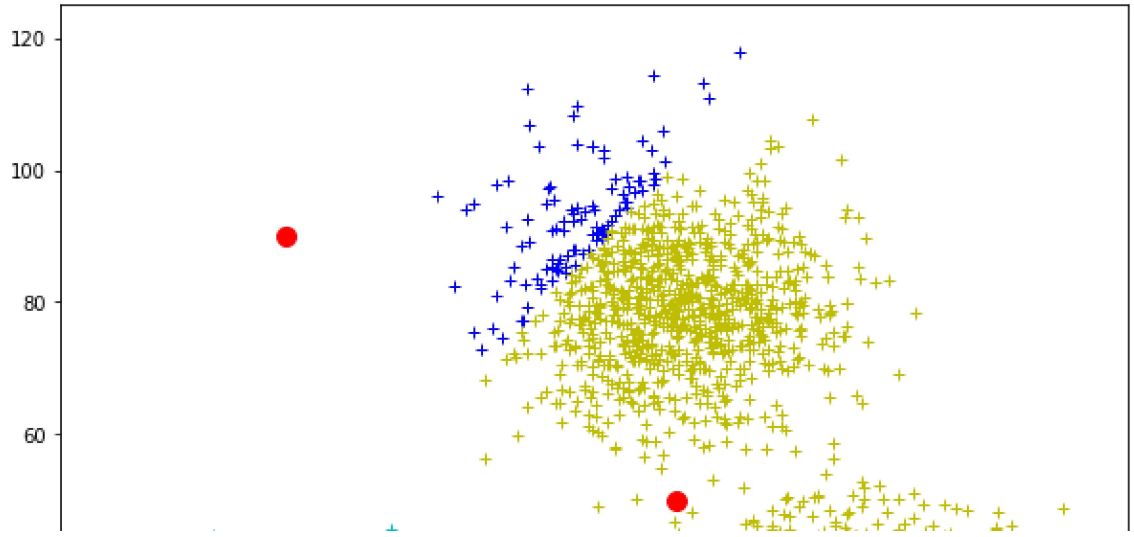
Now, we compute the assignment of the data points to the prototypes (array `closest`). This is also wrong, but we're going to be using it anyay.

In [6]:

```
figsize(10,10)
dists = cdist(protos,data)
closest = argmin(dists,axis=0)
for i in range(len(protos)):
    plot(data[closest==i,0],data[closest==i,1],['c+','g+','b+','y+'][i])
plot(protos[:,0],protos[:,1],'ro',markersize=10)
```

Out[6]:

```
[<matplotlib.lines.Line2D at 0x2984fe169b0>]
```



In [7]:

```
history = [protos.copy()]

for i in range(len(protos)):
    protos[i,:] = average(data[closest==i],axis=0)

history.append(protos.copy())
```

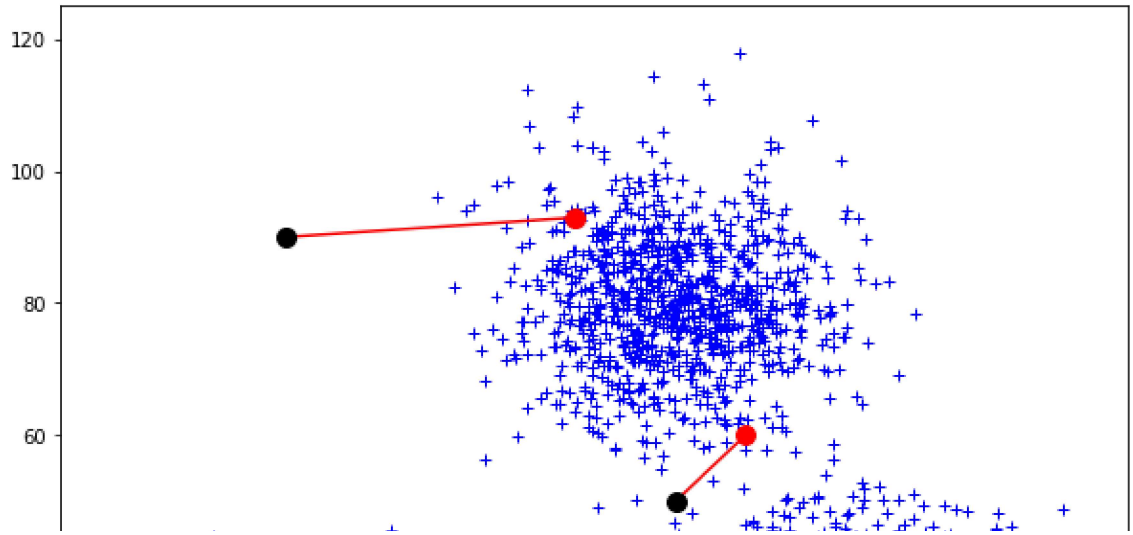
Now we pretend that the cluster assignments are correct and recompute the location of the centers.

In [8]:

```
# Plotting the centroid movements
figsize(10,10)
plot(data[:,0],data[:,1],'b+')
harray = array(history)
for i in range(len(protos)):
    plot(harray[:,i,0],harray[:,i,1],'r')
plot(harray[0,:,0],harray[0,:,1],'ko',markersize=10)
plot(protos[:,0],protos[:,1],'ro',markersize=10)
```

Out[8]:

```
[<matplotlib.lines.Line2D at 0x298503eafd0>]
```



As you can see, the centers have moved, and it looks like they have generally moved in the right direction.

Now let's just repeat this process multiple times

In [9]:

```
for round in range(1000):
    if round%100==0: sys.stderr.write("%d "%round)
    dists = cdist(protos,data)
    closest = argmin(dists,axis=0)
    for i in range(len(protos)):
            protos[i,:] = average(data[closest==i],axis=0)
    history.append(protos.copy())
```

```
0 100 200 300 400 500 600 700 800 900
```

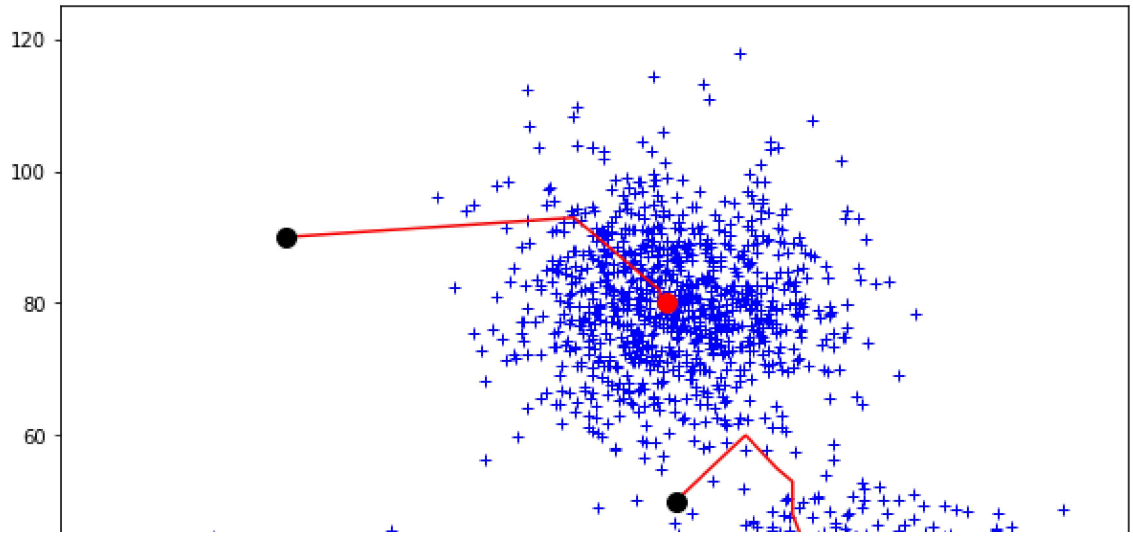
We can now plot the path that the prototype guesses have taken.

In [10]:

```
figsize(10,10)
plot(data[:,0],data[:,1],'b+')
history = array(history)
for i in range(len(protos)):
    plot(history[:,i,0],history[:,i,1],'r')
plot(history[0,:,0],history[0,:,1],'ko',markersize=10)
plot(protos[:,0],protos[:,1],'ro',markersize=10)
```

Out[10]:

```
[<matplotlib.lines.Line2D at 0x298501fc860>]
```



As you can see, the final location of the prototype centers (red) are nicely in the center of the classes. The algorithm doesn't give us exactly the cluster centers because there are three clusters but we postulated four cluster centers.

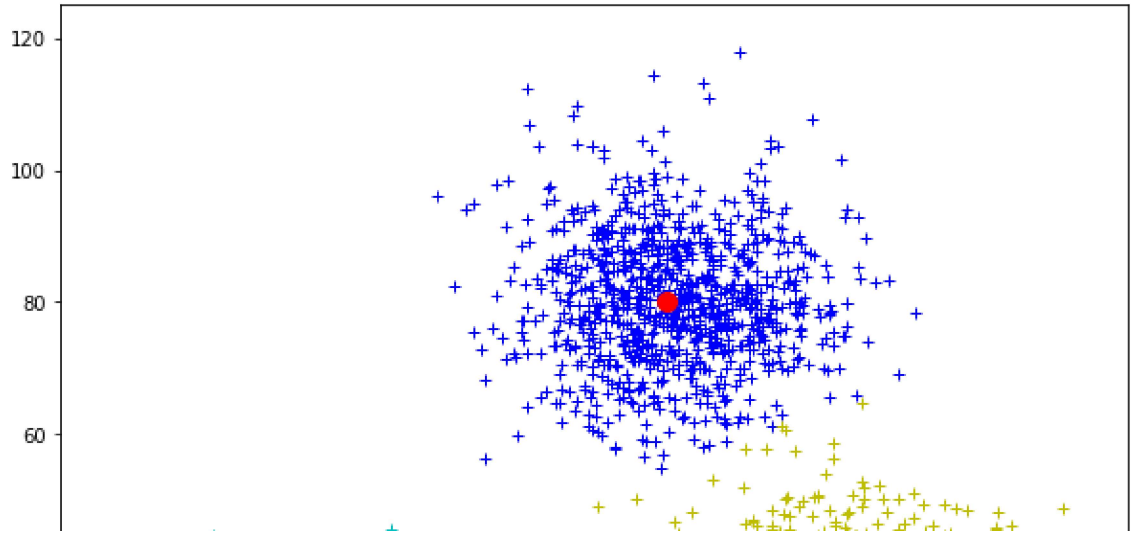
We can also look at the partition of the data induced by these cluster centers.

In [11]:

```
figsize(10,10)
dists = cdist(protos,data)
closest = argmin(dists,axis=0)
for i in range(len(protos)):
  plot(data[closest==i,0],data[closest==i,1],['c+','g+','b+','y+'][i])
plot(protos[:,0],protos[:,1],'ro',markersize=10)
```

Out[11]:

```
[<matplotlib.lines.Line2D at 0x298501e8e48>]
```

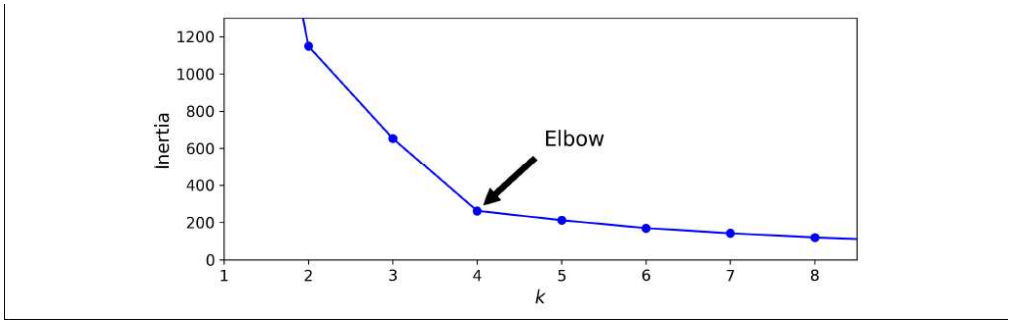


The diagram above shows the step by step implementation of the K-Means algorithm.

As we saw earlier, we need to provide the value of K beforehand. But the question is how to get a good value of K. An optimum value of K is obtained using the Elbow Method.

The Elbow-Method

This method is based on the relationship between the within-cluster sum of squared distances(WCSS Or Inertia) and the number of clusters. It is observed that first with an increase in the number of clusters WCSS decreases steeply and then after a certain number of clusters the drop in WCSS is not that prominent. The point after which the graph between WCSS and the number of clusters becomes comparatively smother is termed as the elbow and the number of cluster at that point are the optimum number of clusters as even after increasing the clusters after that point the variation is not decreasing by much i.e., we have accounted for almost all the dissimilarity in the data. An elbow-curve looks like:



An example with actual data

In [79]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
% matplotlib inline
```

```
UsageError: Line magic function `%` not found.
```

In [80]:

```
dataset=pd.read_csv('Mall_Customers.csv')
```

In [81]:

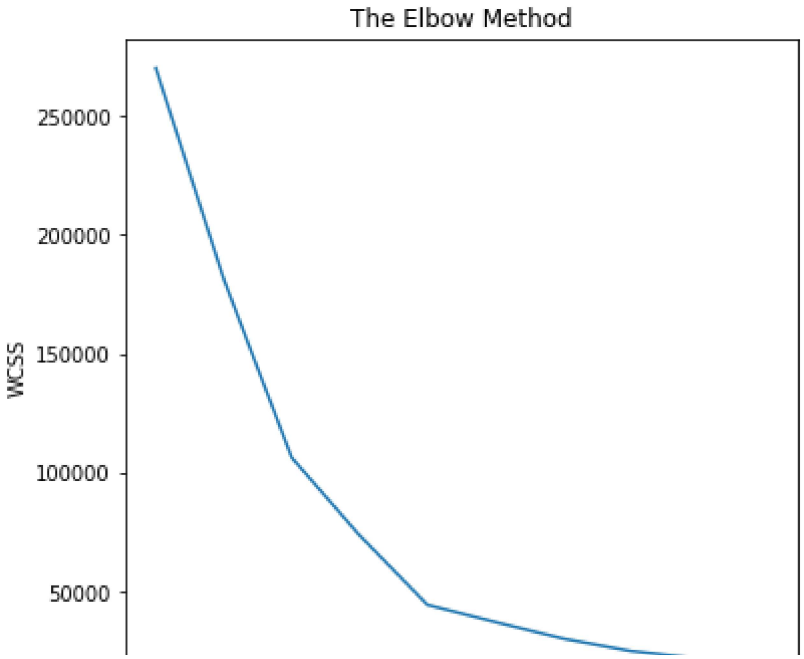
```
#dataset
X=dataset.iloc[:,3:]
X
```

Out[81]:

	Annual Income (k$)	Spending Score (1-100)
0	15	39
1	15	81
2	16	6
3	16	77
4	17	40
5	17	76
6	18	6
7	18	94
8	19	3
9	19	72
10	19	14

In [82]:

```
#elbow method
from sklearn.cluster import KMeans
wcss=[]
for i in range (1,11):
    kmeans=KMeans(n_clusters=i,init='k-means++',random_state=42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1,11),wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



In [83]:

```
# Fitting K-Means to the dataset
kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)
y_kmeans = kmeans.fit_predict(X)
print(y_kmeans)
```

```
[2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2
 3 2 3 2 3 2 0 2 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 4 1 4 0 4 1 4 1 4 0 4 1 4 1 4 1 4 1 4 0 4 1 4 1 4
 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1
 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4]
```

In [85]:

```
# It predicts the cluster number to which the datapoint belongs to
test=kmeans.predict(np.asarray([[3,3]]))
test[0]
```

Out[85]:

```
2
```

In [77]:

```
# saving the model to the local file system
import pickle
filename = 'finalized_model.pickle'
pickle.dump(kmeans, open(filename, 'wb'))
```

In [ ]:

```
# Looking at the points which belong to Cluster0
X[y_kmeans==0]
```

In [ ]:

```
# Visualising the clusters
plt.scatter(X[y_kmeans == 0]['Annual Income (k$)'], X[y_kmeans == 0]['Spending Score (1
plt.scatter(X[y_kmeans == 1]['Annual Income (k$)'], X[y_kmeans == 1]['Spending Score (1
plt.scatter(X[y_kmeans == 2]['Annual Income (k$)'], X[y_kmeans == 2]['Spending Score (1
plt.scatter(X[y_kmeans == 3]['Annual Income (k$)'], X[y_kmeans == 3]['Spending Score (1
plt.scatter(X[y_kmeans == 4]['Annual Income (k$)'], X[y_kmeans == 4]['Spending Score (1
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s = 300, c =
plt.title('Clusters of customers')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.legend()
plt.show()
```

Custom Centroid Initialization

If you already know approximately where the centroids should be (e.g., if you ran another clustering algorithm earlier), then you can set the init hyperparameter to a NumPy array containing the list of centroids, and set n_init to 1:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

Challenges and improvements in K-Means:

1. We need to specify the number of clusters beforehand.
2. It is required to run the algorithm multiple times to avoid a sub-optimal solution
3. K-Means does not behave very well when the clusters have varying sizes, different densities, or non-spherical shapes.
4. The clusters sometimes vary based on the initial choice of the centroids. An important improvement to the K-Means algorithm, called **K-Means++**, was proposed in a *2006 paper by David Arthur and Sergei Vassilvitskii*. They introduced a smarter initialization step that tends to select centroids that are distant from one another, and this makes the K-Means algorithm much less likely to converge to a suboptimal solution.
5. Another important improvement to the K-Means algorithm was proposed in a *2003 paper by Charles Elkan*.It considerably accelerates the algorithm by avoiding many unnecessary distance calculations: this is achieved by exploiting the *triangle inequality* (i.e., the straight line is always the shortest; in a triangle with sides a,b and c=> **a+b>c**) and by keeping track of lower and upper bounds for distances between instances and centroids.
6. Yet another important variant of the K-Means algorithm was proposed in a *2010 paper by David Sculley*. 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 by a factor of 3 or 4 and makes it possible to cluster huge datasets that do not fit in memory. Scikit-Learn implements this algorithm in the **MiniBatchKMeans** class. You can just use this class like the KMeans class:

```
from sklearn.cluster import MiniBatchKMeans

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

Hierarchical clustering

One main disadvantage of K-Means is that it needs us to pre-enter the number of clusters (K). Hierarchical clustering is an alternative approach which does not need us to give the value of K beforehand and also, it creates a beautiful tree-based structure for visualization.

Here, we are going to discuss the bottom-up (or Agglomerative) approach of cluster building. We start by defining any sort of similarity between the datapoints. Generally, we consider the Euclidean distance. The points which are closer to each are more similar than the points which re farther away. The Algorithms starts with considering all points as separate clusters and then grouping pints together to form clusters.

The Algorithm:

1. Begin with n observations and a measure (such as Euclidean distance) of all the n(n−1)/2 pairwise dissimilarities(or the Euclidean distances generally). Treat each observation as its own cluster. Initially, we have n clusters.
2. Compare all the distances and put the two closest points/clusters in the same cluster. The dissimilarity(or the Euclidean distances) between these two clusters indicates the height in the dendrogram at which the fusion line should be placed.
3. Compute the new pairwise inter-cluster dissimilarities(or the Euclidean distances) among the remaining clusters.
4. Repeat steps 2 and 3 till we have only one cluster left.

Code Example

In [13]:

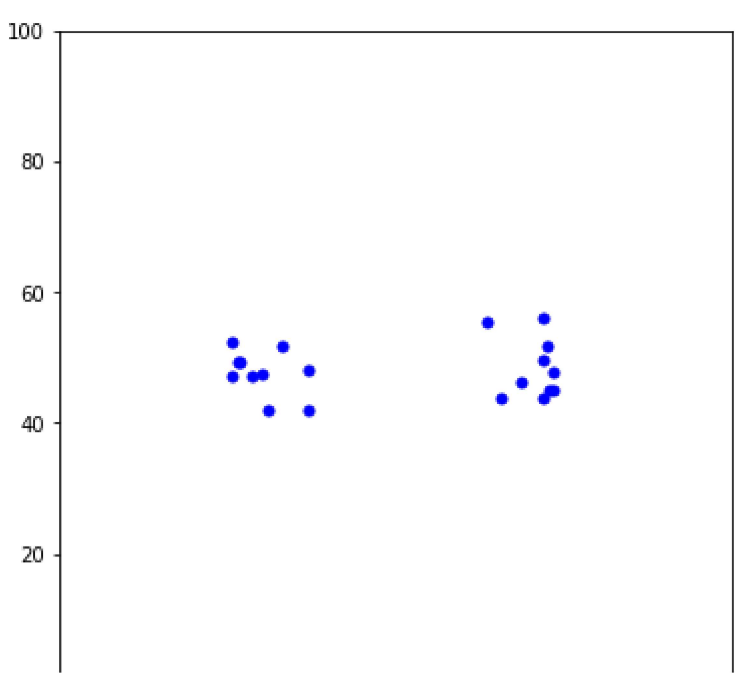
```
# Importing the necessary libraries
import random as pyrandom
from scipy.spatial.distance import cdist
figsize(6,6)
```

In [14]:

```
#Defining a method to plot the clusters
ccolors = ['go', 'ro','bo','mo','co','yo']
def plotclusters(data,centers=None):
    xlim([0,100]); ylim([0,100])
    if centers is None:
        plot(data[:,0],data[:,1],'bo',markersize=5)
    else:
        for i in range(amax(centers)+1):
            plot(data[centers==i,0],data[centers==i,1],ccolors[i%len(ccolors)],markersi
```

In [15]:

```
# A Simple Clustering Problem
data = r_[4*randn(10,2)+array([70,50]),
          4*randn(10,2)+array([30,50])]
# shuffle(data)
plotclusters(data)
```



The idea behind linkage clustering, or hierarchical clustering, is to put things that are close together into the same cluster. Linkage clustering is usually based on distances only.

In [16]:

```
# Compute distance between each pair of the two collections of inputs i.e., calculate d
from scipy.spatial.distance import cdist
ds = cdist(data,data)
```

Based on pairwise distances, we can now compute a linkage matrix. We successively merge the closest points/clusters into the same cluster. The linkage "matrix" is simply a table listing which pairs of points are merged at what step and what distance.

In [17]:

```
from scipy.cluster.hierarchy import *
lm = linkage(ds,"single")
lm[:5]
```

```
C:\Users\virat\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: ClusterW
arning: scipy.cluster: The symmetric non-negative hollow observation matrix
looks suspiciously like an uncondensed distance matrix
```

Out[17]:

```
array([[11.        , 16.        ,  0.50111644,  2.        ],
       [ 8.        ,  9.        ,  1.73971512,  2.        ],
       [ 3.        , 21.        ,  4.48217897,  3.        ],
       [15.        , 20.        ,  5.34399685,  3.        ],
       [14.        , 19.        ,  6.17529356,  2.        ]])
```

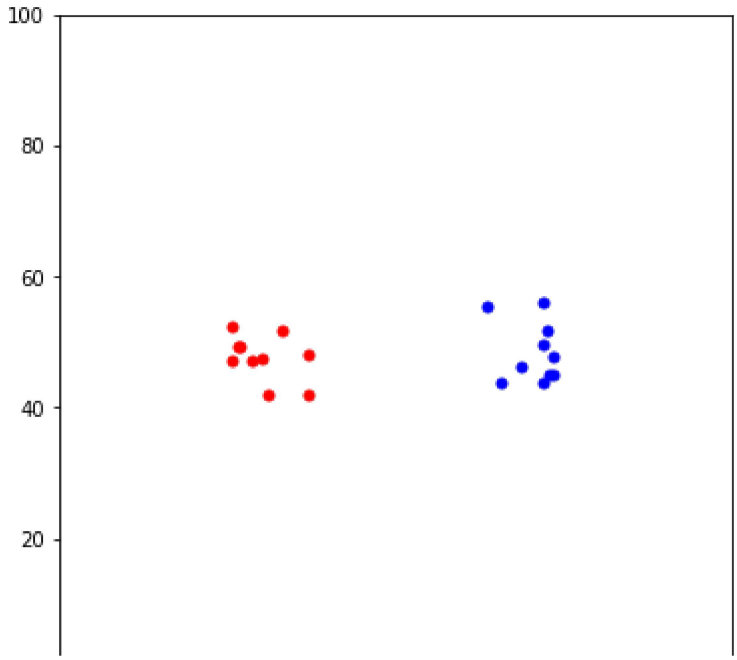
In [18]:

```
_=dendrogram(lm)
```

We can "cut" the dendrogram to form flat clusters. If we cut the above diagram into two clusters, we get a good answer.

In [19]:

```
## Plotting the individual clusters
plotclusters(data,fcluster(lm,2,criterion='maxclust'))
```



As you can see that now the clusters are shown in two different colours. It means that the algorithm has created two separate groupings based on some similarity criteria.

In the lines above, have talked about linkage matrices and we have written in our code *lm = linkage(ds,"**single**")*. So Let's discuss about the different types of linkages that we generally use.

Linkage Methods

In [21]:

```
# Taking a data for example
data = r_[dot(randn(100,2),diag([4,10]))+array([60,60]),
          dot(randn(100,2),diag([4,10]))+array([40,40])]
shuffle(data)
```

Initially, hierarchical clustering starts out with clusters consisting of individual points.

Later, it compares clusters with each other and merges the two "closest" clusters.

Since clusters are sets of points, there are many different kinds of linkage methods:

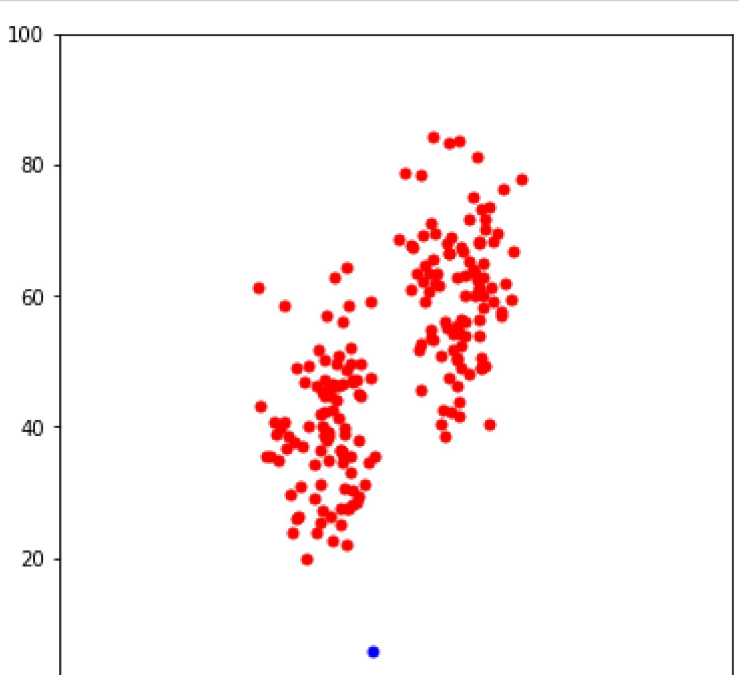
- Single Linkage: cluster distance = smallest pairwise distance
- Complete Linkage: cluster distance = largest pairwise distance
- Average Linkage: cluster distance = average pairwise distance
- Centroid Linkage: cluster distance= distance between the centroids of the clusters
- Ward's Linkage: cluster criteria= Minimize the variance in the cluster

Single Linkage: Minimal intercluster dissimilarity. Compute all pairwise dissimilarities between the observations in cluster A and the observations in cluster B, and record the smallest of these dissimilarities. Single linkage can result in extended, trailing clusters in which single observations are fused one-at-a-time.

- cluster distance is the smallest distance between any point in cluster 1 and any point in cluster 2
- highly sensitive to outliers when forming flat clusters
- works well for low-noise data with an unusual structure

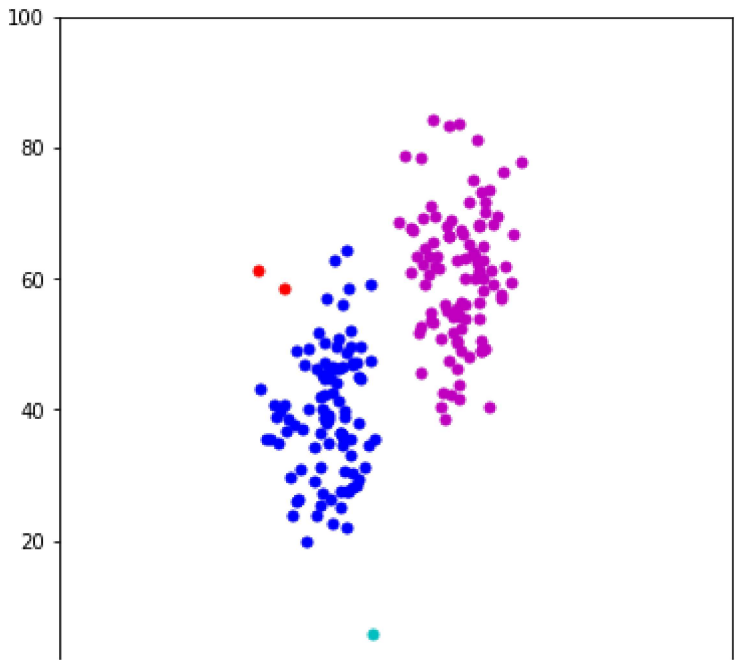
In [23]:

```
assignment = fclusterdata(data,2,metric='euclidean',method='single',criterion='maxclust
plotclusters(data,assignment)
```



In [24]:

```
assignment = fclusterdata(data,4,metric='euclidean',method='single',criterion='maxclust
plotclusters(data,assignment)
```

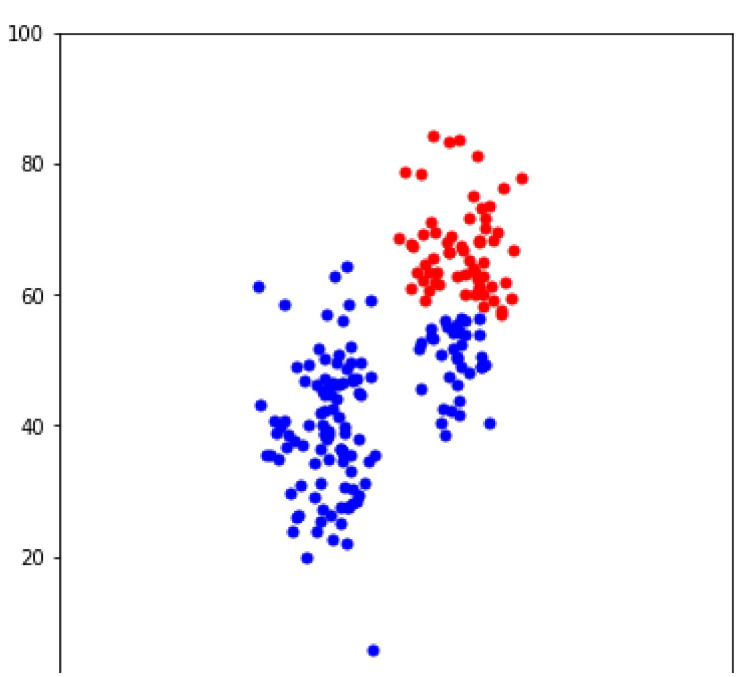


Complete Linkage: Maximal intercluster dissimilarity. Compute all pairwise dissimilarities between the observations in cluster A and the observations in cluster B, and record the largest of these dissimilarities.

- cluster distance is the largest distance between any point in cluster 1 and any point in cluster 2
- less sensitive to outliers than single linkage

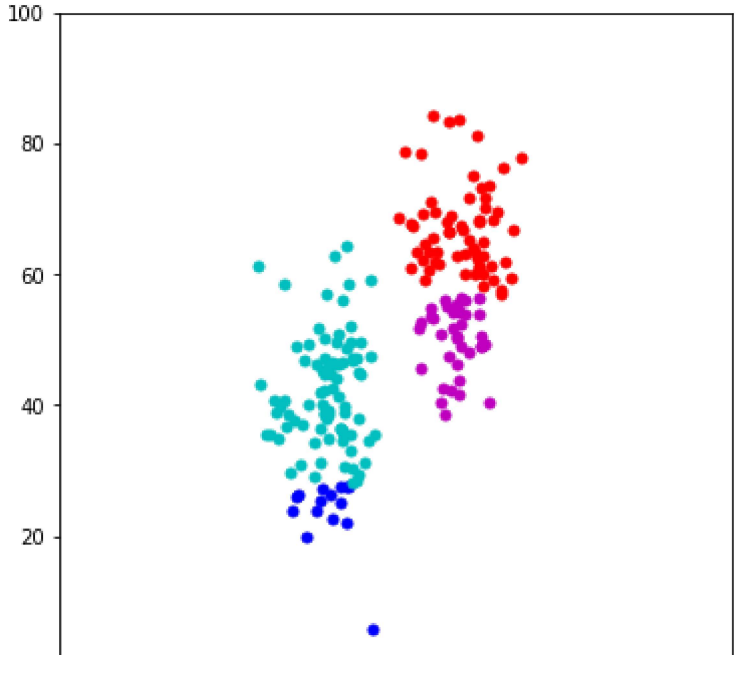
In [25]:

```
assignment = fclusterdata(data,2,metric='euclidean',method='complete',criterion='maxclu
plotclusters(data,assignment)
```



In [26]:

```
assignment = fclusterdata(data,4,metric='euclidean',method='complete',criterion='maxclu
plotclusters(data,assignment)
```

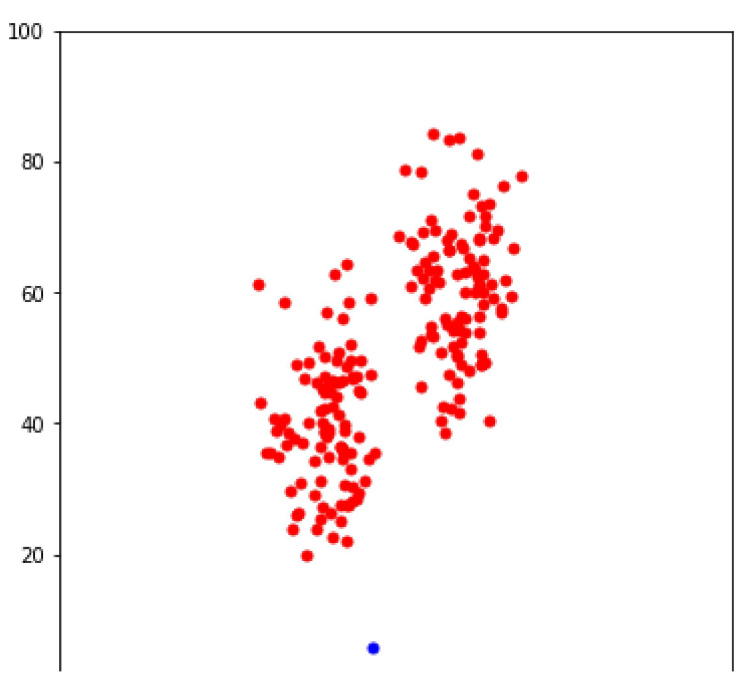


Average Linkage: Mean intercluster dissimilarity. Compute all pairwise dissimilarities between the observations in cluster A and the observations in cluster B, and record the average of these dissimilarities.

- cluster distance is the average distance of all pairs of points in clusters 1 and 2

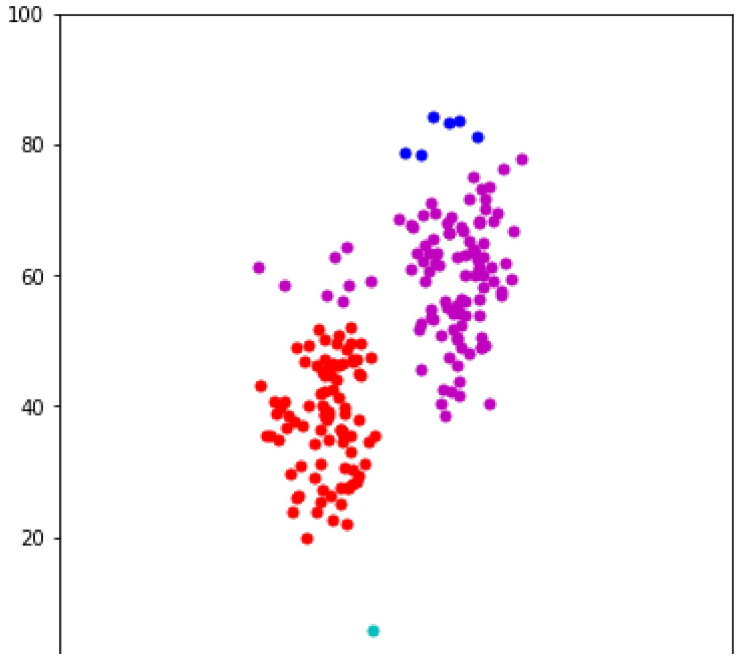
In [27]:

```
assignment = fclusterdata(data,2,metric='euclidean',method='average',criterion='maxclus
plotclusters(data,assignment)
```



In [28]:

```
assignment = fclusterdata(data,4,metric='euclidean',method='average',criterion='maxclus
plotclusters(data,assignment)
```

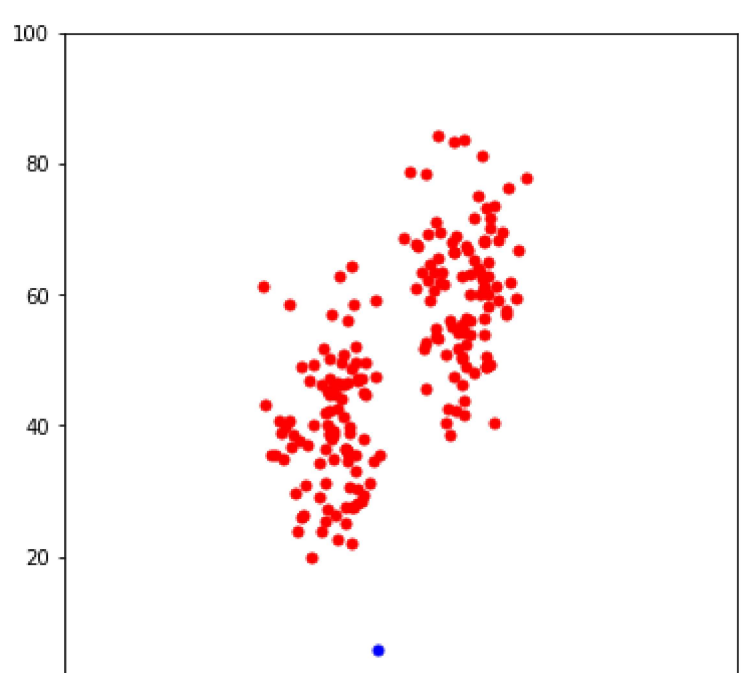


Centroid Linkage: The dissimilarity between the centroid for cluster A (a mean vector of length p) and the centroid for cluster B. Centroid linkage can result in undesirable inversions.

- cluster distance is the distance of the centroids of both clusters

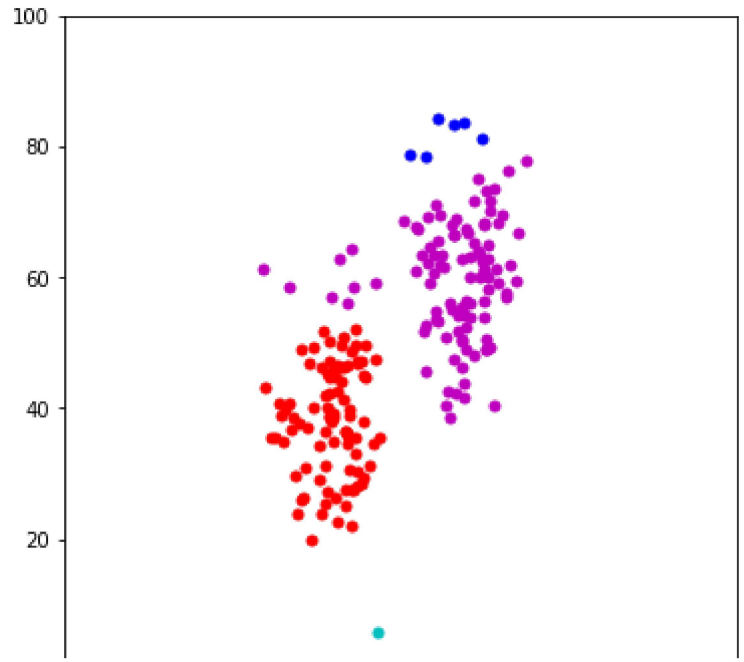
In [29]:

```
lm = linkage(data,metric='euclidean',method='centroid')
assignment = fcluster(lm,2,criterion='maxclust')
plotclusters(data,assignment)
```



In [30]:

```
lm = linkage(data,metric='euclidean',method='centroid')
assignment = fcluster(lm,4,criterion='maxclust')
plotclusters(data,assignment)
```

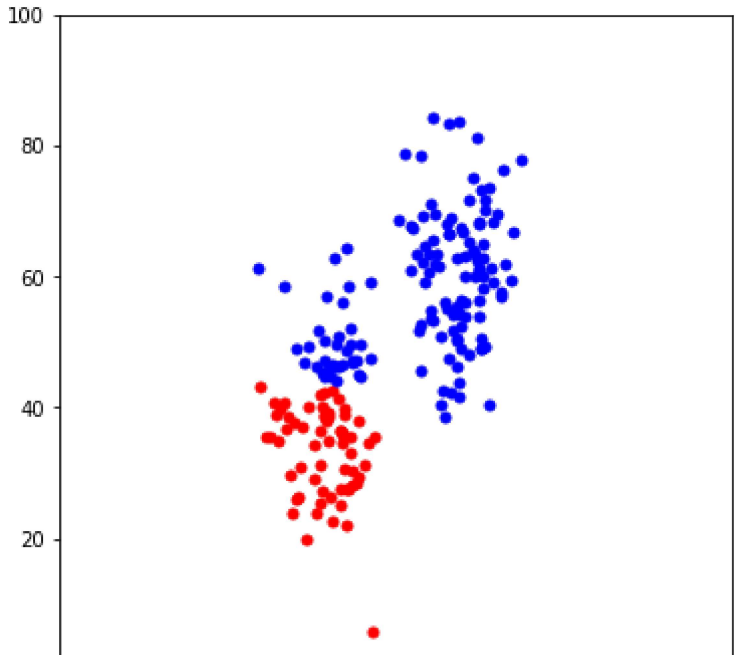


Ward linkage: Wikipidea says *Ward's minimum variance criterion minimizes the total within-cluster variance. To implement this method, at each step find the pair of clusters that leads to minimum increase in total within-cluster variance after merging.*

- based on minimizing a variance criterion before and after merging

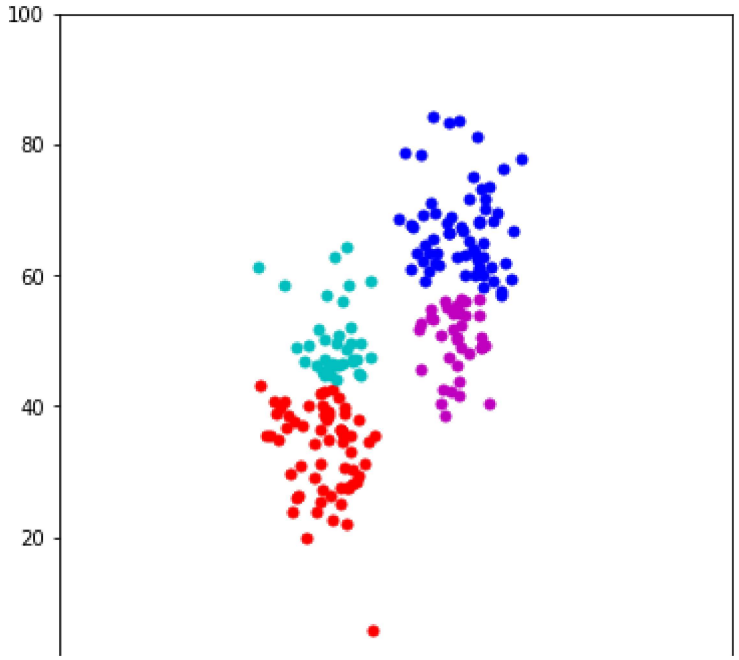
In [31]:

```
lm = linkage(data,metric='euclidean',method='ward')
assignment = fcluster(lm,2,criterion='maxclust')
plotclusters(data,assignment)
```



In [32]:

```
lm = linkage(data,metric='euclidean',method='ward')
assignment = fcluster(lm,4,criterion='maxclust')
plotclusters(data,assignment)
```



DBSCAN(Density Based Spatial Clustering of Apllications with Noise)

It is an unsupervised machine learning algorithm.This algorithm defines clusters as continuous regions of high density.

Some definitions first:

Epsilon: This is also called eps. This is the distance till which we look for the neighbouring points.

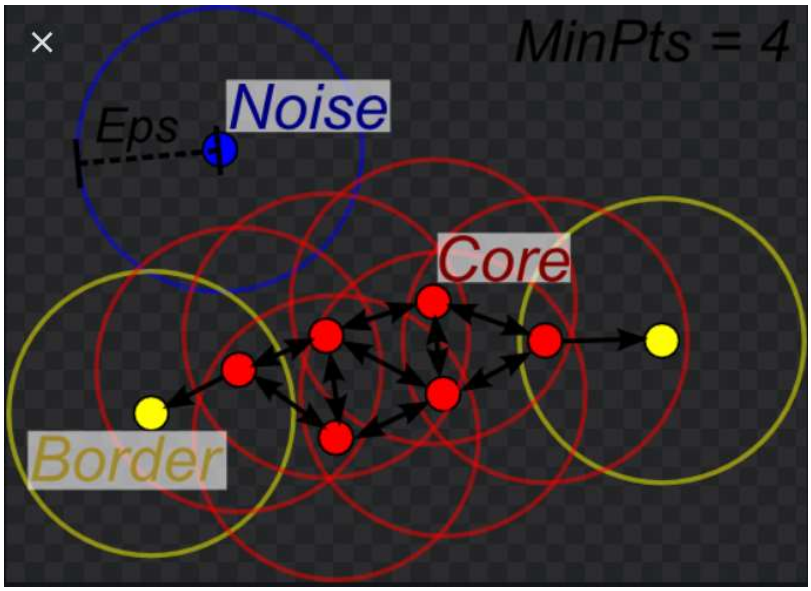
Min_points: The minimum number of points specified by the user.

Core Points: If the number of points inside the *eps radius* of a point is greater than or equal to the *min_points* then it's called a core point.

Border Points: If the number of points inside the *eps radius* of a point is less than the *min_points* and it lies within the *eps radius* region of a core point, it's called a border point.

Noise: A point which is neither a core nor a border point is a noise point.

Let's say if the eps=1 and min_points =4



Algorithm Steps:

1. The algorithm starts with a random point in the dataset which has not been visited yet and its neighbouring points are identified based on the eps value.
2. If the point contains greater than or equal points than the min_pts, then the cluster formation starts and this point becomes a _core point_, else it's considered as noise. The thing to note here is that a point initially classified as noise can later become a border point if it's in the eps radius of a core point.
3. If the point is a core point, then all its neighbours become a part of the cluster. If the points in the neighbourhood turn out to be core points then their neighbours are also part of the cluster.
4. Repeat the steps above until all points are classified into different clusters or noises.

This algorithm works well if all the clusters are dense enough, and they are well separated by low-density regions.

Python Example

In [39]:

```
# Necessary Imports

import numpy as np

from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
```

In [40]:

```
# Data creation
centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=750, centers=centers, cluster_std=0.4,
                            random_state=0)  # generate sample blobs

X = StandardScaler().fit_transform(X)
```

In [48]:

```
# DBSCAN
db = DBSCAN(eps=0.3, min_samples=10).fit(X)

# we are calculating these for showcasing in diagram
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)  # creating an array of true
core_samples_mask[db.core_sample_indices_] = True  # setting the indices of the core re
labels = db.labels_  # similar to the model.fit() method, it gives the labels of the cl
```

In [42]:

```
# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0) # the label -1 is considere
n_noise_ = list(labels).count(-1)  # calculating the number of clusters
```

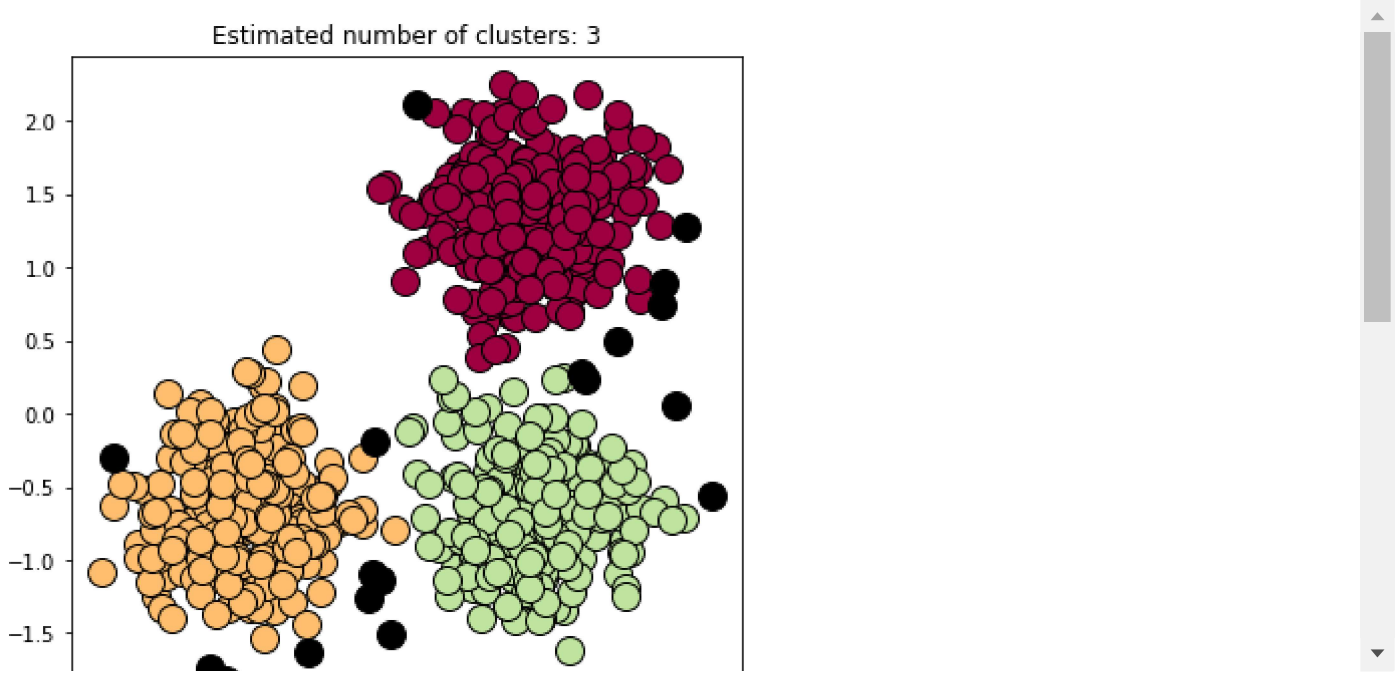
In [44]:

```
print('Estimated number of clusters: %d' % n_clusters_)
print('Estimated number of noise points: %d' % n_noise_)

"""Homogeneity metric of a cluster labeling given a ground truth.

A clustering result satisfies homogeneity if all of its clusters
contain only data points which are members of a single class."""

print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels_true, labels))
```

```
Estimated number of clusters: 3
Estimated number of noise points: 18
Homogeneity: 0.953
```

In [58]:

```
# Plot result
import matplotlib.pyplot as plt

# Black  is used for noise.
unique_labels = set(labels)  # identifying all the unique labels/clusters
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]  # creating the list of co



for k, col in zip(unique_labels, colors):


    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)   # assigning class members for each class


    xy = X[class_member_mask & core_samples_mask] # creating the list of points for eac

    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)


    xy = X[class_member_mask & ~core_samples_mask] # creating the list of noise points

    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)



plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```



In short, DBSCAN is a very simple yet powerful algorithm, capable of identifying any number of clusters, of any shape, it is robust to outliers, and it has just two hyper parameters(eps and min_samples). However, if the

density varies significantly across the clusters, it can be impossible for it to capture all the clusters properly. Moreover, its computational complexity is roughly O(m log m), making it pretty close to linear with regards to the number of instances.

Evaluation of Clustering

Cluster Validity

The validation of clusters created is a troublesome task. The problem here is: ”clusters are in the eyes of the beholder”

A good cluster will have:

- High inter-class similarity, and
- Low intraclass similarity

Aspects of cluster validation

- **External:** Compare your cluster to the ground truth.
- **Internal:** Evaluating the cluster without reference to external data.
- **Reliability:** The clusters are not formed by chance(randomly)- some statistical framework can be used.

External Measures:

N: Number of objects in the data P: ${P_1,P_2,…,P_m}$ the set of ground truth clusters C: ${C_1,C_2,…C_n}$ the set of clusters formed by the algorithm

The Incidence Matrix: N* N matrix

$P_ij =1$ if the two points O_i and O_j belong to the same cluster in the ground truth else $P_ij=0$

$C_ij =1$ if the two points O_i and O_j belong to the same cluster in the ground truth else $C_ij=0$

Now there can be the following scenarios:

1. $C_ij=P_ij=1$ --> both the points belong to the same cluster for both our algorithm and ground truth(Agree)--- **SS**
2. $C_ij=P_ij=0$ --> both the points don’t belong to the same cluster for both our algorithm and ground truth(Agree)--- **DD**
3. $_Cij=1 but P_ij=0$ --> The points belong in the same cluster for our algorithm but in different clusters for the ground truth (Disagree)---- **SD**
4. $C_ij=0 but P_ij=1$ --> The points don’t belong in the same cluster for our algorithm but in same clusters for the ground truth (Disagree)----**DS**

Rand Index= $\frac{Total Agree}{Total Disagree}=\frac{(SS+DD)}{(SS+DD+DS+SD)}$

The disadvantage of this is that it could be dominated by DD.

Jaccard Coeeficient=$\frac{ SS}{(SS+SD+DS)}$

In [64]:

```
print("Rand Index: %0.3f"
      % metrics.adjusted_rand_score(labels_true, labels))
```

```
Rand Index: 0.952
```

In [65]:

```
metrics.jaccard_similarity_score(labels_true, labels)
```

Out[65]:

```
0.972
```

A higher value of Rand Index and Jaccard's coefficient mean that the clusters generated by our algorithm mostly agree to the ground truth.

Confusion matrix:

n= number of points

m_i=points in *cluster i*

c_j=points in *class j*

n_{ij}= points in cluster i coming from cluster j

$p_{ij}=\frac{n_{ij}}{m_i}$= probability of element from cluster i to be assigned to class j

	Class 1	Class 2	Class 3	
Cluster 1	n_{11}	n_{12}	n_{13}	m_1
Cluster 2	n_{21}	n_{22}	n_{23}	m_2
Cluster 3	n_{31}	n_{32}	n_{33}	m_3
	c_1	c_2	c_3	n

	Class 1	Class 2	Class 3	
Cluster 1	p_{11}	p_{12}	p_{13}	m_1
Cluster 2	p_{21}	p_{22}	p_{23}	m_2
Cluster 3	p_{31}	p_{32}	p_{33}	m_3
	c_1	c_2	c_3	n

Entropy Entropy of Cluster i, given by $e_i= - \sum p_{ij} log (p_{ij})$

For the entire clustering algorithm, the entropy can be given as: $e= \sum \frac{m_i}{n}e_i$

Purity The purity is the total percentage of data points clustered correctly.

The purity of cluster i, given by $p_i=max (p_{ij})$

And for the entire cluster it is: $p(C)=\sum \frac{m_i}{n}p_i$

The Scikit-Learn Package hasen't yet implemented the Purity metrics. Hence, we'll write our custom code to implememt that.

In [67]:

```
import numpy as np
from sklearn import metrics

def purity_score(y_true, y_pred):
    # compute contingency matrix (also called confusion matrix)
    contingency_matrix = metrics.cluster.contingency_matrix(y_true, y_pred)
    # return purity
    return np.sum(np.amax(contingency_matrix, axis=0)) / np.sum(contingency_matrix)
```

In [68]:

```
purity_score(labels_true, labels)
```

Out[68]:

```
0.9813333333333333
```

A high value of purity score means that our clustering algorithm performs well against the ground truth.

So far we have discussed the External Measures. But as it is an unsupervised learning problem, most of the times there is no ground truth to compare to. Let's discuss the internal measures.

Internal Measures

These are the methods use to measure the quality of clusters without external references. There are two aspects to it.

- **Cohesion:** How closely the objects in the same cluster are related to each other. It is the within-cluster sum of squared distances. It is the same metric that we used to calculate for the K-Means algorithm. $WCSS= \sum \sum (x-m_i)^2$
- **Separation:** How different the objects in different clusters are and how distinct a well-separated cluster is from other clusters. It is the between cluster sum of squared distances. $BSS=\sum C_i(m-m_i)^2$

Where C is the size of the individual cluster and m is the centroid of all the data points.

Note: BSS+WSS is always a constant.

The silhouette can be calculated as:

$$s(x) = \frac{b(x) - a(x)}{\max\{a(x), b(x)\}}$$

Where a(x) is the avarage distance of x from all the other points in the same cluster and b(x) is the avarage distance of x from all the other points in the other clusters.

And the Silhoeutte coefficient is given by:

$SC=\frac{1}{N} \sum S(x)$

In [62]:

```
1 metrics.silhouette_score
```

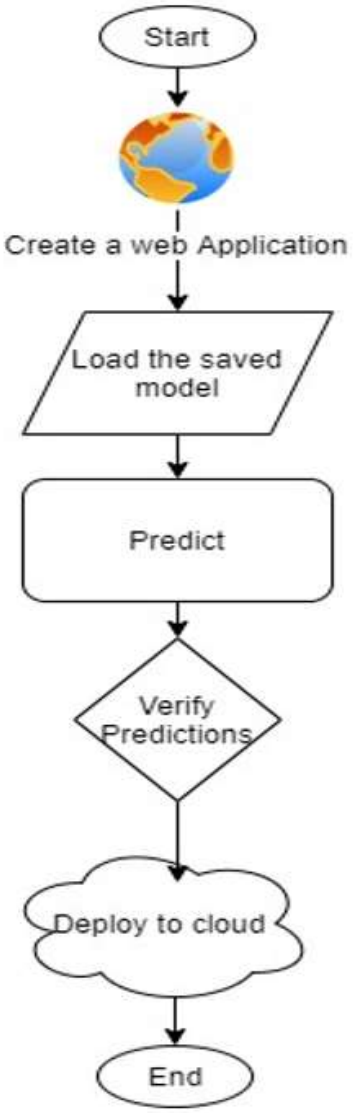
Out[62]:

```
0.972
```

A higher silhouette score means that the inter cluster similarity is less and the intra cluster dissimilarity is more which is the measure of good clustering.

Cloud Deployment (AWS)

Once the training is completed, we need to expose the trained model as an API for the user to consume it. For prediction, the saved model is loaded first and then the predictions are made using it. If the web app works fine, the same app is deployed to the cloud platform. The application flow for cloud deployment looks like:



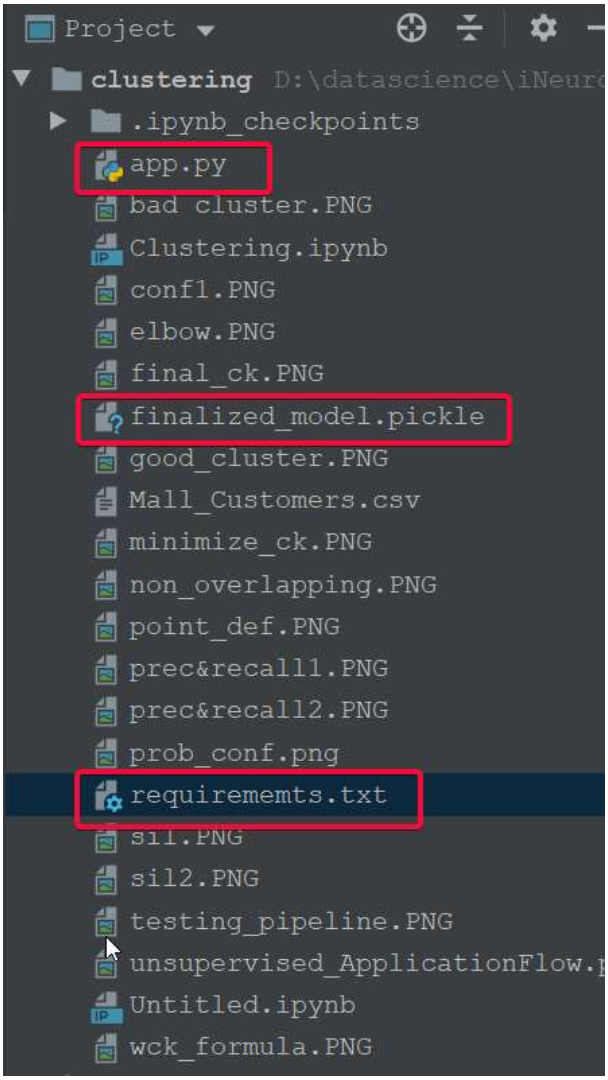
Pre-requisites:

- Basic knowledge of flask framework.

- Any Python IDE installed(we are using PyCharm).
- A Microsoft Azure account.
- Basic understanding of HTML.

Deployment Steps:

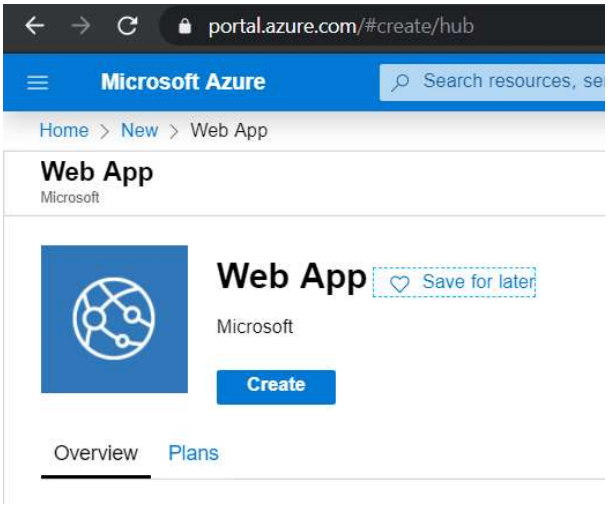
1. Create the project structure, as shown below:



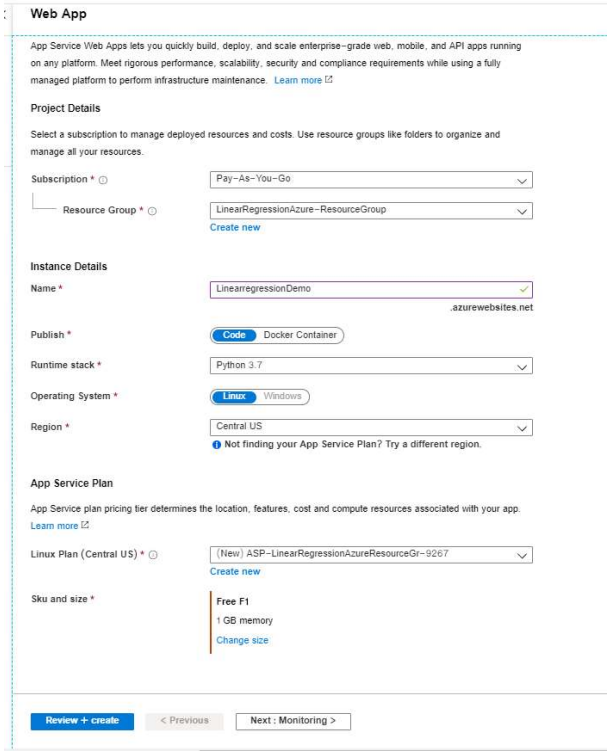

```
2. Create the 'app.py' file as:

# importing the necessary dependencies
from flask import Flask, render_template, request,jsonify
from flask_cors import CORS,cross_origin
import pickle
from flask import Response
import numpy as np

app = Flask(__name__) # initializing a flask app


@app.route('/predict',methods=['POST','GET']) # route to show the predictions
@cross_origin()
def index():
    if request.method == 'POST':
        try:
            #  reading the inputs given by the user
```

```
            annual_income=float(request.json['annual_income'])
            spending_score = float(request.json['spending_score'])
            filename = 'finalized_model.pickle'
            loaded_model = pickle.load(open(filename, 'rb')) # loading the model file from the storage
            # predictions using the loaded model file

prediction=loaded_model.predict(np.asarray([[annual_income,spending_score]]))
            print('prediction is', (list(prediction)[0]))
            # showing the prediction results in a UI
            return Response(str((list(prediction)[0])))
        except Exception as e:
            print('The Exception message is: ',e)
            return Response('something is wrong.')



if __name__ == "__main__":
    #app.run(host='127.0.0.1', port=8001, debug=True)
   app.run(debug=True) # running the app


```

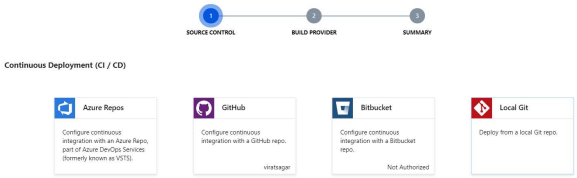
3. Go to https://portal.azure.com/ (https://portal.azure.com/) and create an account if already haven't created one.
4. Go to the Azure account and create a web app.



5. Provide the app name, resource group(create new if necessary), runtime stack(Python 3.7), region, select the 1 GB size, which is free to use. Click Review+create to create the web app



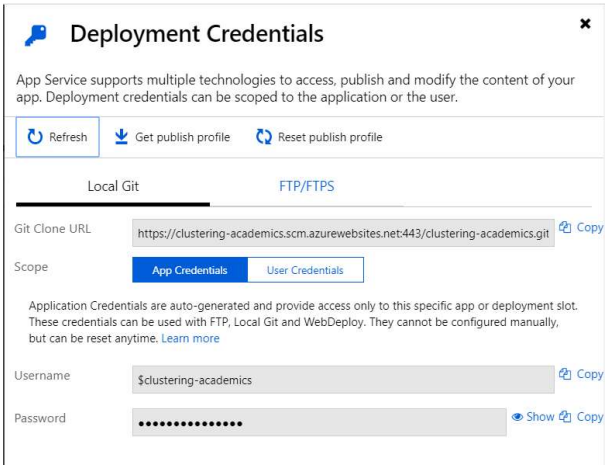
6. Once the deployment is completed, open the app and go to the 'Deployment Center' option. Select 'local git' for source control and click continue



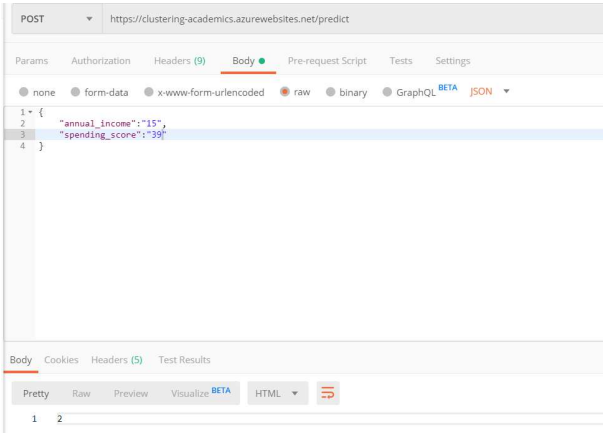
7. Select the kudo 'App service build provider' as the build provider and click continue.

8. Click 'Finish' to complete the setup.
9. Go to the overview section of the app, and the Git link now will be visible.

10. Go to 'Deployment Credentials' in deployment center and copy the username and password. These will be required when doing the final push to the remote git repository.



11. Open a command prompt and navigate to your project folder.
12. Run **git init** to initialise an empty git repository
13. Create a new remote git alias using the command: git remote add
14. Use **git add .** to add all the files to the local git repository.
15. Use **git commit -m "first commit"** to commit the code to the git repo.
16. Push the code to the remote repo using: git push master –f
17. This prompts for a username and password. Provide the same credentials as copied in the step above.
18. After deployment, from the ‘overview’ section, copy the URL and paste into any web API test to see the application running.



References:

Introduction to Statistical Learning

Hands-on Machine Learning with Scikit-Learn

Scikit-Learn Definitions