Clustering Algorithms

By **AchyutGanti** and **Nathaniel Bowerman**

Introduction

The idea was to dive a bit deeper into Clustering algorithms than what was discussed in the class.

We know that Clustering is an Unsupervised Machine learning technique that is used in grouping the data points together. It is called Unsupervised because unlike in Regression or Classification, we are not provided with labels using which we can train the algorithm to predict new data.

So, in theory the idea behind the clustering algorithms is that data points that fall in the same category should have similar properties and that are not in the same group should have dissimilar properties.

Types of Clustering Algorithms

There are many techniques that are available to us to cluster data but the popular ones discussed here are

- K-means Clustering
- K-medians Clustering
- Mean Shift Clustering Algorithm
- EM Clustering
- DBSCAN

K-Means Clustering Algorithm

Let us take a look at K-means algorithm first and try to implement it in Python.

K-means is probably the most well known and well used algorithm of all the clustering algorithms out there. It is really simple to use and understand. The k-means algorithm requires us to mention the number of groups or the value of 'k' before the algorithm even begins grouping the data poins.

It is based on this value that random centroids are initialized in the vector space.

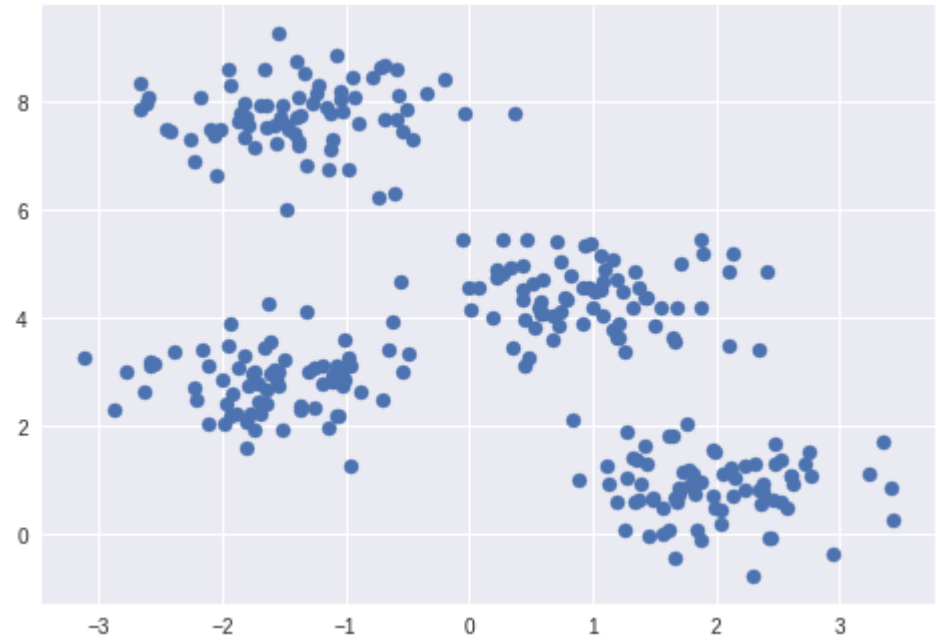
Now each data point is is classified based on the distance metric between the point and the centroid. The data point that is closest to a centroid is assigned to a particular group.

After the above step the group centers are reassigned by calculating the mean of the data points in a specific group.

the above steps are repeated again and again until we get reach a point where no more changes are seen in the group centers.

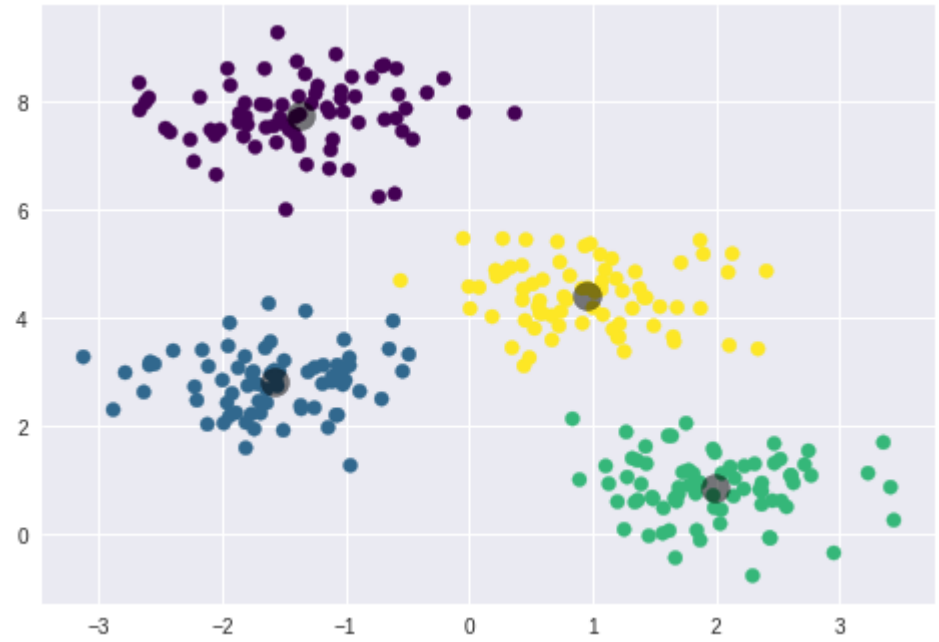
Now let us take a look at the Pythonic implementation of K-means algorithm using the scikit-learn package.

In [1]:
```
# Importing the required libraries

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()  # for plot styling
import numpy as np

# Creating a fake dataset to using make_blobs function in scikit lear
n.

from sklearn.datasets.samples_generator import make_blobs

X, y_true = make_blobs(n_samples=300, centers=4,
                       cluster_std=0.60, random_state=0)
plt.scatter(X[:, 0], X[:, 1], s=50);
```



In [3]:
```
from sklearn.cluster import KMeans

# Fit the k-means library on the data.
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)

y_kmeans = kmeans.predict(X)
```

In [4]:
```
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5
);
```



K-means algorithm grouped similar data together based on the k - value that we provided. Although the k-means algorithm has many advantages like being fast and easy, it has a downside. It requires us to choose the value of k beforehand and this can be not so efficient sometimes. Also, being dependent on means makes it sensitive to outliers.

K-medians

This is just a slight modification to the k-means algorithms. In k-means we compute the mean of the vector to adjust the group centers whereas in k-medians we compute the medians of the vector.

Therefore, it is less sensitive to outliers. But the only downside of using this algorithm as opposed to the k-means is that it gets slower as the data gets bigger becuase we have to compute median after every iteration and that requires us to sort the data. So, the bigger the data, the computationally expensive it becomes.

Mean-shift Clustering Algorithm

This is another clustering algorithm that finds its applications mainly in computer vision and image segmentation. It is a sliding window based algorithm that attempts to finds the denser regions of the data points.

Even this is centroid based but the one main difference between Mean shift and K-means is that the Mean shift algorithm doesnt calculate the distance between the data points and the centroid to assign groups.

The way this algorithm works is that it begins with a sliding window centered at point A with a radius 'r' as the kernel. And this kernel or the window attempts to move towards the denser locations as we iterate.

At every iteration the sliding window attempts to move towards the denser locations of the dataset by relocating the center point towards the region that has higher densities that the previous ones. It basically shifts the center point towards the mean of the points within that window.

This process keeps on repeating untill we reach a point where there is no more changes in the center point and we have reached the densest point. At this stage, the data points are clustered together according to the sliding window in which they fall.

For datasets with many possible clusters we assign randomly distributed sliding windows to start with so that the algorithm reaches the global maximum and does not gets saturated at the local minimas.

Now let us take a look at the pythonic implementation of the mean shift algortihm :

In [19]:
```
# Loading the required libraires

import numpy as np
from sklearn.cluster import MeanShift
from sklearn.datasets.samples_generator import make_blobs
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import style
style.use('ggplot')
```

In [20]:
```
## Create the data using random centers,
centers = [[1,1,1],[5,5,5],[3,10,10]]
X,_ = make_blobs(n_samples=500, centers = centers,cluster_std=1)
```

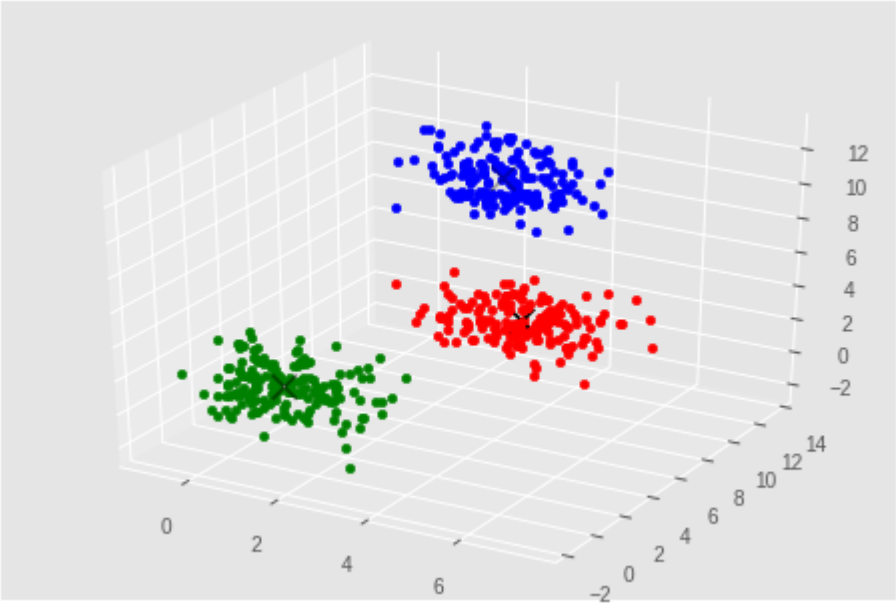
In [21]:
```
ms = MeanShift()
ms.fit(X)
labels = ms.labels_
cluster_centers = ms.cluster_centers_
print(cluster_centers)
```

```
[[  4.93452585   4.9231042    5.08278551]
 [  0.97486371   0.99661232   1.10636488]
 [  2.95947723   9.83694326  10.0236047 ]]
```

In [22]:
```
n_clusters_ = len(np.unique(labels))
print('number of estimated clulsters:',n_clusters_)
```

```
('number of estimated clulsters:', 3)
```

In [25]:
```
colors = 10*['r','g','b','c','k','y','m']
fig = plt.figure()
ax = fig.add_subplot(111,projection = '3d')

for i in range(len(X)):
    ax.scatter(X[i][0],X[i][1],X[i][2],c = colors[labels[i]],
               marker = 'o')
ax.scatter(cluster_centers[:,0],cluster_centers[:,1],
           cluster_centers[:,2],
          marker='x',color='k',s=150,linewidths=5,zorder=10)
plt.show()
```



As seen in the above example, the mean shift algorithm is effective in grouping similar data together. But the algorithm has its downsides. It requires us to set a value for the radius and the performance of the algorithm depends on this size of the sliding window. Lower the value of the radius, the algorithm takes a lot if time converging to the optimal location. If we choose a higher value for the sliding window, it might overshoot the global maximum and fall off in the local maximums.

EM-Clustering

EM Clustering is the most commonly used algorithm in the distribution-based class of clustering algorithms, which use probability distributions to cluster data. EM-Clustering makes use of the Expectation-Maximization (EM) Algorithm to cluster datapoints into groups. Briefly, the EM algorithm uses an iterative process to identify parameters that maximize a likelihood function based on the observed data. In each step, a guess of the parameters is obtained and the likelihood of this guess given the data is calculated. The guess is updated based on the likelihood function until the algorithm converges.

EM Clustering starts by defining the number of clusters, the believed underlying distribution of the data, and initial values for the parameters for these distributions. The EM algorithm is used to identify the most probable parameters for all clusters. For example, if the underlying distribution was multivariate normal, the mean vector and covariance matrix for each cluster would be the output. These parameters are used to calculate the probability that each point belongs to each cluster, which can be used to assign points to clusters.

EM-Clustering is similar to K-Means or K-Medians in that it has a pre-defined number of clusters and tries to assign points to them, but EM-Clustering uses probability distributions rather than the difference in mean distances to accomplish this. This adds flexibility and robustness to the algorithm.

EM-Clustering is performed in R using the assign.class() funtion in the EMCluster library. To demonstrate the potential advantages of EM-Clustering over K-means, a sample data set was created in R using three clusters created by random sampling inside two-dimensional ellipses (Figure 1). The clusters were assigned using K-means clustering (Figure 3), then using EM-Clustering (Figure 4). All 600 points in this dataset were clustered correctly using EM-clustering, but only 556 using K-means. Sample code is given in the attached R Markdown file.

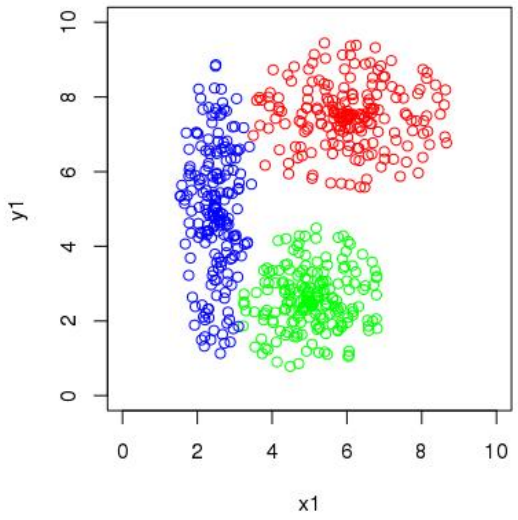


Figure 1: Sample data using actual groups as labels

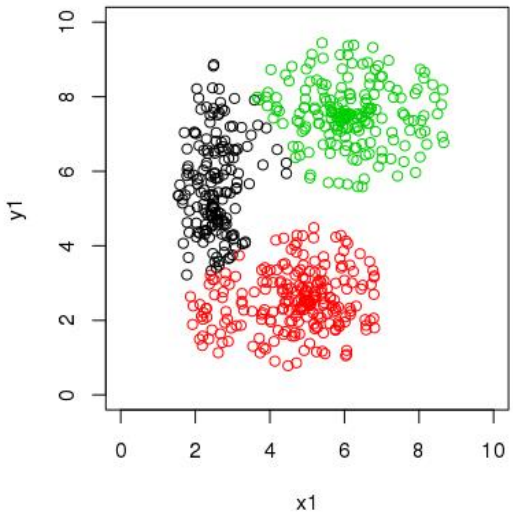


Figure 2: K-Means clustering of sample data

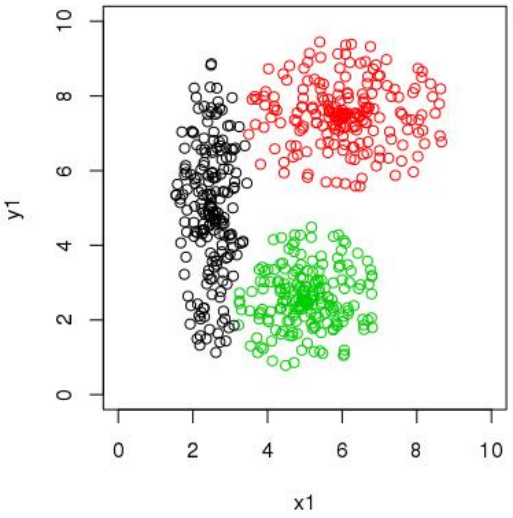


Figure 3: EM-Clustering of sample data

DBSCAN

DBSCAN is one of the most commonly used algorithms in the density-based class of clustering algorithms, which identify areas of high density in the data to assign clusters.

DBCAN is run in R using the dbscan() function in the dbscan package. An example of a dataset that clusters well using DBSCAN is given below in Figure 4. Despite having two clear clusters, this data would not cluster well using any of the previous cluster, which all use variation of distances from a center point to define clusters, due to the odd shape of the red cluster. Figure 5 shows the output of DBSCAN in R for this data. All points are assigned to the correct cluster, but note that there are some points in black not assigned to any clusters, which is one drawback of DBSCAN. An explanation of the algorithm is given on the next page.

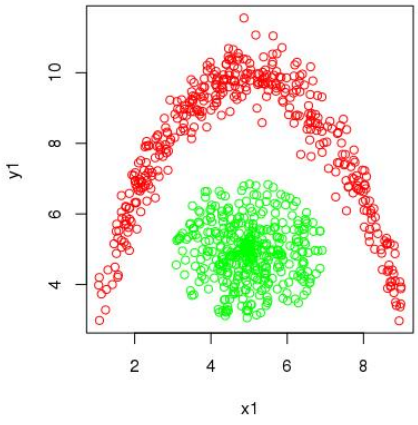


Figure 4: Sample data with actual clusters

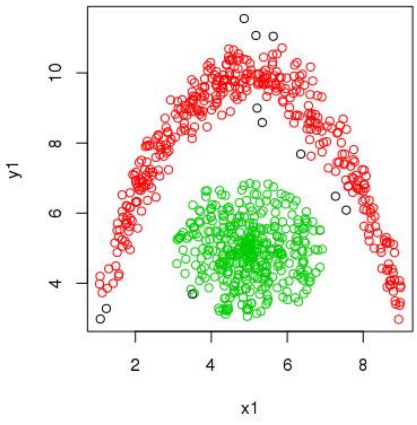
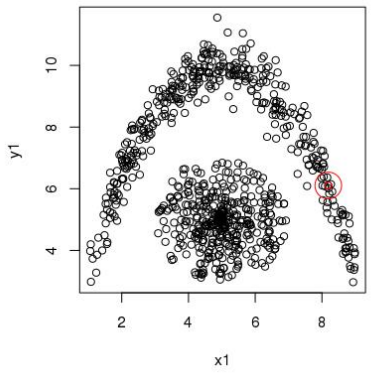
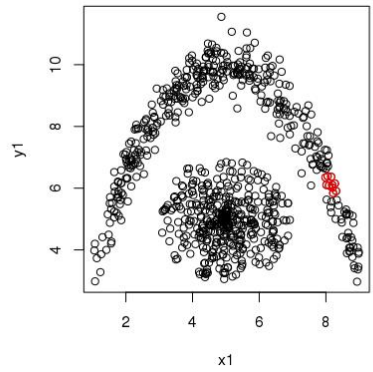


Figure 5: DBSCAN cluster results

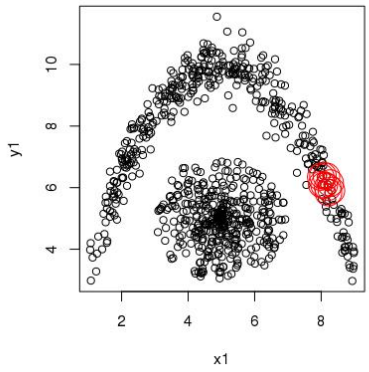
DBCAN works by finding points that are connected by neighborhoods of density of points to identify cluster membership. There are two parameters for this method: ε and MinPts. The algorithm starts by picking a random point, *p*. A neighborhood of distance ε around *p* is determined, which is highlighted in red in the figure to the right.



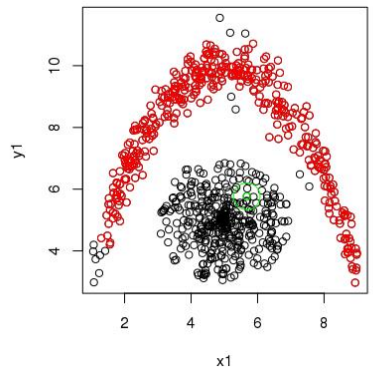
If the total number of points within the neighborhood of *p* is greater than or equal to MinPts, *p* and the points within ε from *p* are added to the cluster.



Neighborhoods of distance ε are drawn around all new points added to the cluster. For each of these points, any of these new neighborhoods that have greater than or equal to MinPts within them have those points added to the cluster.

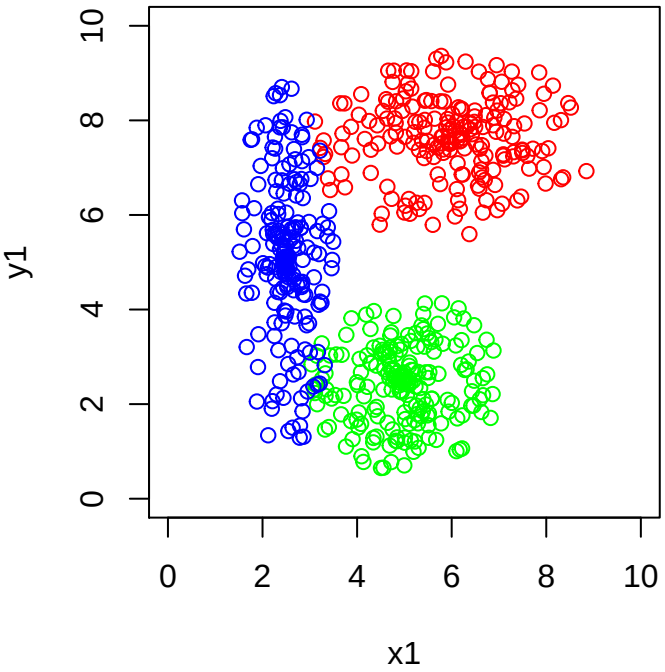


This continues until the cluster is unchanged (red dot on the figure to the right). Then a point not already in a cluster is chosen to start a new cluster (see in green on the figure to the right). The algorithm continues until all points have been examined.



DBSCAN and EM

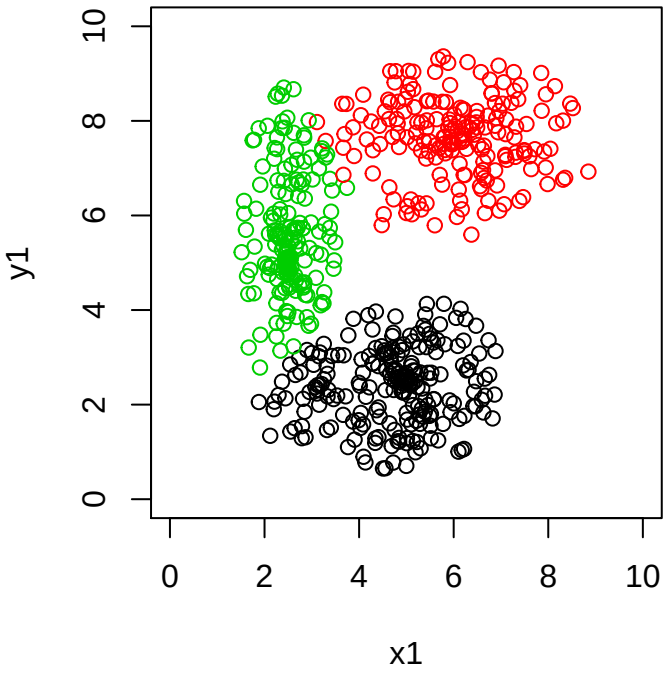
```r
#create sample EM dataset
set.seed(1)
n=200
r1=runif(n,0,2)
t1=runif(n,0,2*pi)
x1=r1*cos(t1)*1.5+6
y1=r1*sin(t1)+7.5

r2=runif(n,0,2)
t2=runif(n,0,2*pi)
x2=r2*cos(t2)+5
y2=r2*sin(t2)+2.5

r3=runif(n,0,2)
t3=runif(n,0,2*pi)
x3=r3*cos(t3)/2+2.5
y3=r3*sin(t3)*2+5

par(pty="s")
plot(x1,y1,col="red",xlim=c(0,10),ylim=c(0,10))
points(x2,y2,col="green")
points(x3,y3,col="blue")
```



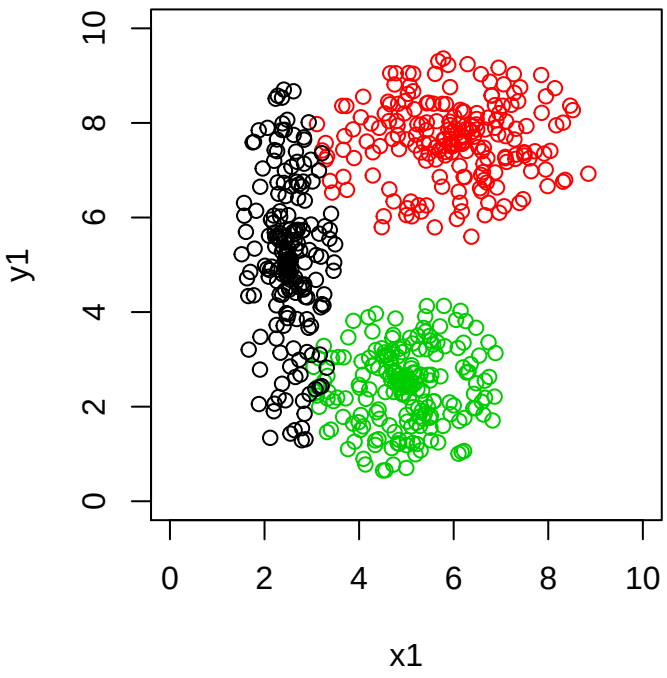
```r
EM_data=rbind(cbind(x1,y1,1),cbind(x2,y2,2),cbind(x3,y3,3))

#run K-means clustering on EM dataset
kmout = kmeans(EM_data,centers=3,iter.max=20)
par(pty="s")
plot(EM_data,col=kmout$cluster,xlim=c(0,10),ylim=c(0,10))

#run EM clustering on EM dataset
library(EMCluster)
```

```
## Loading required package: MASS
```

```
## Loading required package: Matrix
```




```r
emout=assign.class(EM_data,init.EM(EM_data,nclass=3))
plot(EM_data,col=emout$class,xlim=c(0,10),ylim=c(0,10))
```



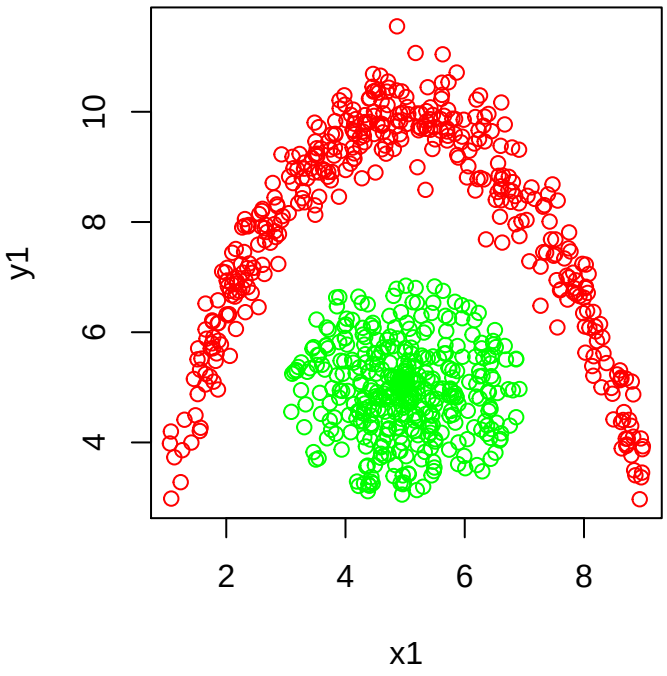

```
#determine how many observations clustered properly
c(sum(emout$class[1:200]==round(mean(emout$class[1:200]))),sum(emout$class[201:400]==round(mean(emout$cl
```

```
## [1] 200 200 200
```

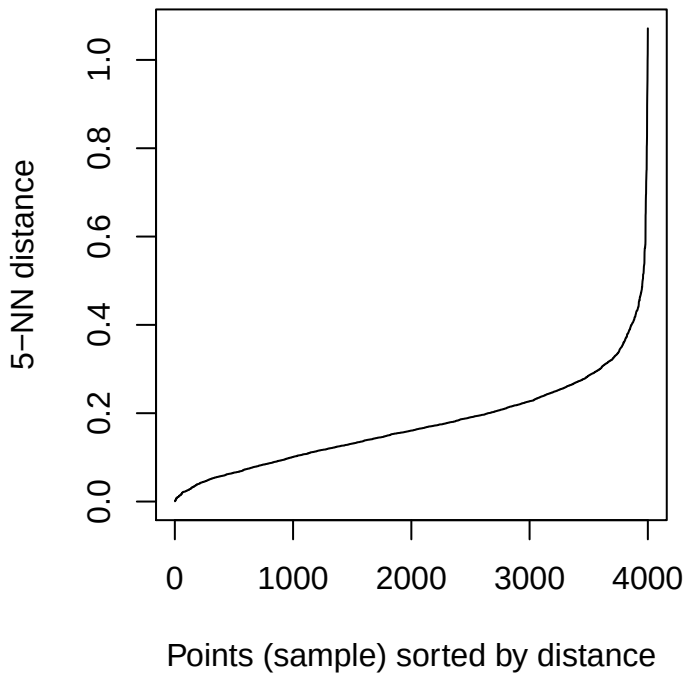
```
c(sum(kmout$cluster[1:200]==round(mean(kmout$cluster[1:200]))),sum(kmout$cluster[201:400]==round(mean(km
```

```
## [1] 194 200 174
```

```
set.seed(7213)
n=400
x1=runif(n,1,9)
y1=-(x1-5)^2/2.5+10+rnorm(n,sd=.5)
par(pty="s")
plot(x1,y1,col="red")

r2=runif(n,0,2)
t2=runif(n,0,2*pi)
x2=r2*cos(t2)+5
y2=r2*sin(t2)+5

points(x2,y2,col="green")
```



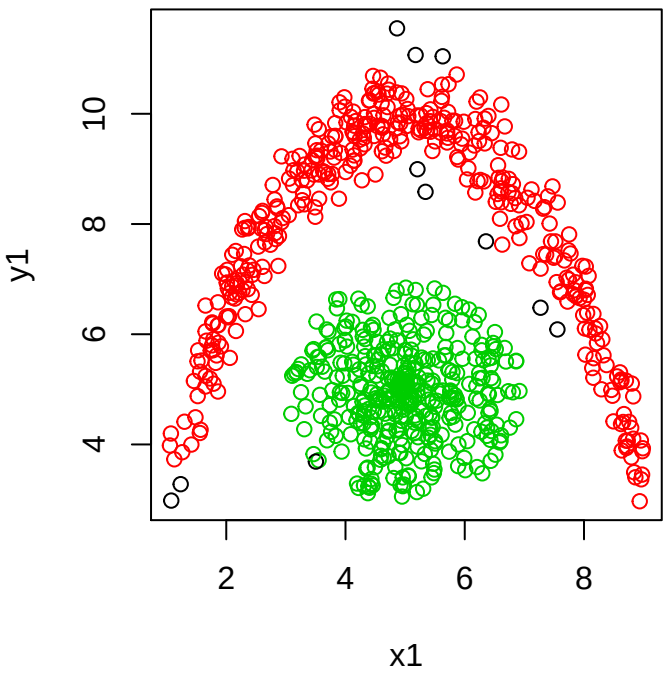

```
library(dbscan)
data=rbind(cbind(x1,y1),cbind(x2,y2))
kNNdistplot(data, k = 5)
```



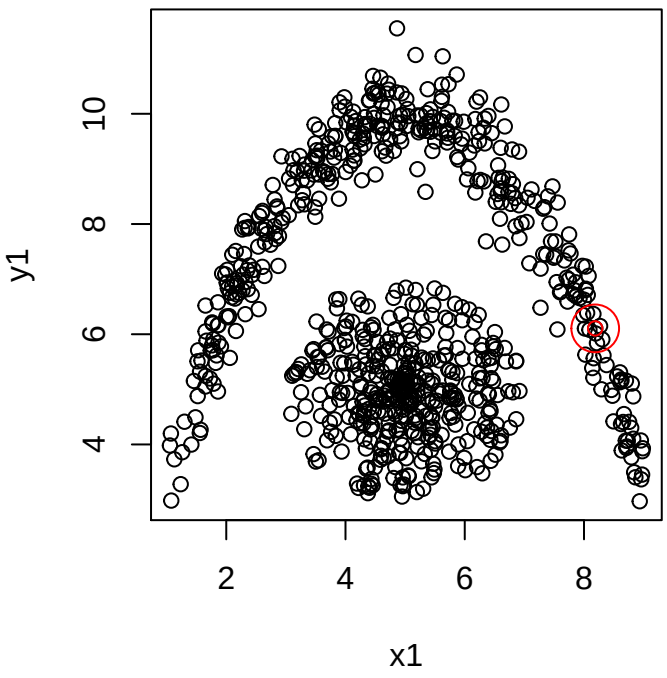

```
eps=.4
minPts=5
dbout=dbscan(data,eps=eps,minPts=minPts)
dbout
```

```
## DBSCAN clustering for 800 objects.
## Parameters: eps = 0.4, minPts = 5
## The clustering contains 2 cluster(s) and 11 noise points.
##
##   0   1   2
##  11 390 399
##
## Available fields: cluster, eps, minPts
```

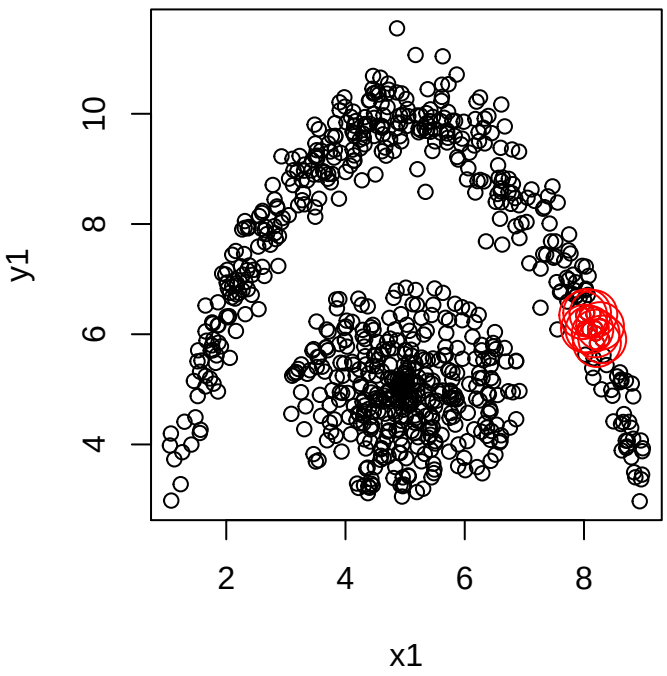
```
plot(data,col=(dbout$cluster+1))
```



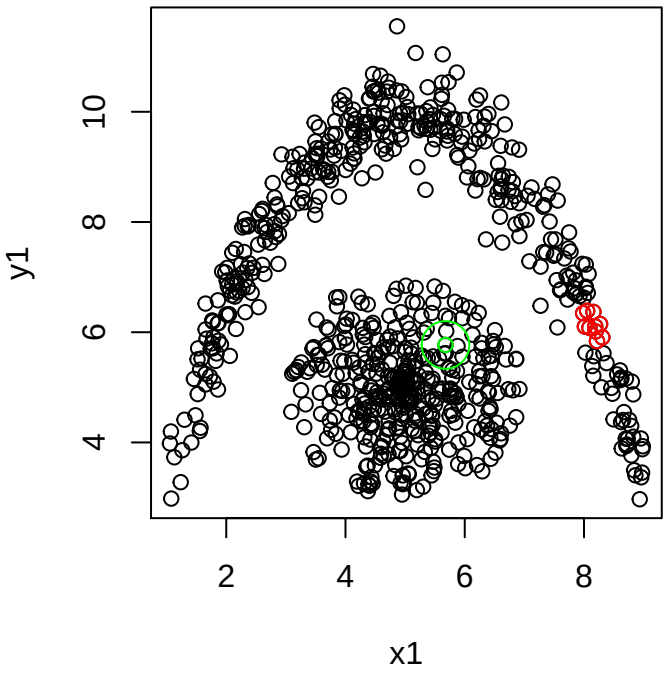

```
#pre-calculate some distance stuff
distances=as.matrix(dist(data,method="euclidean"))
eps_count=rep(0,dim(data)[1])
for (i in 1:length(eps_count)) {
  eps_count[i]=sum(distances[i,]<eps)-1
}
ptsUnder=which(eps_count>=minPts)

#pick the first point
c1=c(75)
par(pty="s")
plot(data)
points(x=data[c1,1],y=data[c1,2],col="red")
symbols(x=data[c1,1], y=data[c1,2], fg="red",circles=rep(eps,length(c1)), add=TRUE, inches=FALSE)
```




```
#repeat this chunk
temp=c1
for (i in 1:length(temp)) {
  c1=unique(c(c1,as.vector(which(distances[c1[i],] < eps))))
}
toPlot1=intersect(setdiff(c1,temp),ptsUnder)
par(pty="s")
plot(data)
points(x=data[c1,1],y=data[c1,2],col="red")
symbols(x=data[toPlot1,1], y=data[toPlot1,2], fg="red",circles=rep(eps,length(toPlot1)), add=TRUE, inch
```




```
#pick a new cluster
c2=c(600)
par(pty="s")
plot(data)
points(x=data[c1,1],y=data[c1,2],col="red")
points(x=data[c2,1],y=data[c2,2],col="green")
symbols(x=data[c2,1], y=data[c2,2], fg="green",circles=rep(eps,length(c2)), add=TRUE, inches=FALSE)
```


y1
10
8
6
4
2
4
6
8
x1