DBSCAN(Density-Based Spatial Clustering of Applications with Noise)

DBSCAN is most widely used density based algorithm. It uses the concept of  **density reachability** and **density connectivity**.   
**Density Reachability**- A point "p" is said to be density reachable from a point "q" if point "p" is within ε distance from point "q" and "q" has sufficient number of points in its neighbors which are within distance ε.  
**Density Connectivity** - A point "p" and "q" are said to be density connected if there exist a point "r" which has sufficient number of points in its neighbors and both the points "p" and "q" are within the ε distance. This is chaining process. So, if "q" is neighbor of "r", "r" is neighbor of "s", "s" is neighbor of "t" which in turn is neighbor of "p" implies that "q" is neighbor of "p".

**Algorithmic steps for DBSCAN clustering**

Let  X = {x1, x2, x3, ..., xn} be the set of data points. DBSCAN requires two parameters: ε (eps) and the minimum number of points required to form a cluster (minPts).

1) Start with an arbitrary starting point that has not been visited.

2) Extract the neighborhood of this point using ε (All points which are within the ε distance are neighborhood).

3) If there are sufficient neighborhood around this point then clustering process starts and point is marked as visited else this point is labeled as noise (Later this point can become the part of the cluster).

4) If a point is found to be a part of the cluster then its ε neighborhood is also the part of the cluster and the above procedure from step 2 is repeated for all ε neighborhood points. This is repeated until all points in the cluster is determined.

5) A new unvisited point is retrieved and processed, leading to the discovery of a further cluster or noise.

6) This process continues until all points are marked as visited.

**Algorithm**

dbscanAlgo<-function(x,eps,minPts = 4){

dist\_mat<-dist(x,method="euclidean",diag = TRUE, upper = TRUE)

dist\_mat<-as.matrix(dist\_mat)

noiseptscount=nrow(dist\_mat)

corepts<-c()

for(i in 1:nrow(dist\_mat)){

count=0;

#print(i)

for(j in 1:ncol(dist\_mat)){

#(new)here consider i=j also

if(dist\_mat[i,j]<=eps)

count=count+1

#sprintf(count)

}

if(count>=minPts){

corepts<-union(corepts,c(i))

}

}

#print(dist\_mat)

#print(corepts)

unvisited<-corepts

stack<-c()

clustercount=0

#print(length(unvisited))

clusterno<-matrix(0, 1,nrow(dist\_mat)) #(new)

while(length(unvisited)!=0){

stack<-union(stack,unvisited[1])

unvisited<-unvisited[-1]

cluster<-c();

clustercount<-clustercount+1

while(length(stack)!=0){

point<-stack[1]

clusterno[1,point]=clustercount #(new)added clusterno to corepoint

stack<-stack[-1]

cluster<-union(cluster,point)

#add neighbours of point

for(j in 1:ncol(dist\_mat)){

if(dist\_mat[point,j]<=eps && point!=j){

#(new)don't add if already in cluster

if(is.element(point,cluster))

cluster<-union(cluster,j)

#(new)giving class to j

clusterno[1,j]=clustercount

#neighbours of point added

if(is.element(j,unvisited)){

stack<-union(stack,j)

#(new)removing j from unvisited as now it will be visited example is given below

remove <- c(j)

unvisited<-unvisited [! unvisited %in% remove]

}

}

}

}

#(new)removing duplicates

cluster<-unique(cluster)

noiseptscount=noiseptscount-length(cluster)

print(paste("Cluster:", clustercount,"No of point:",length(cluster)))

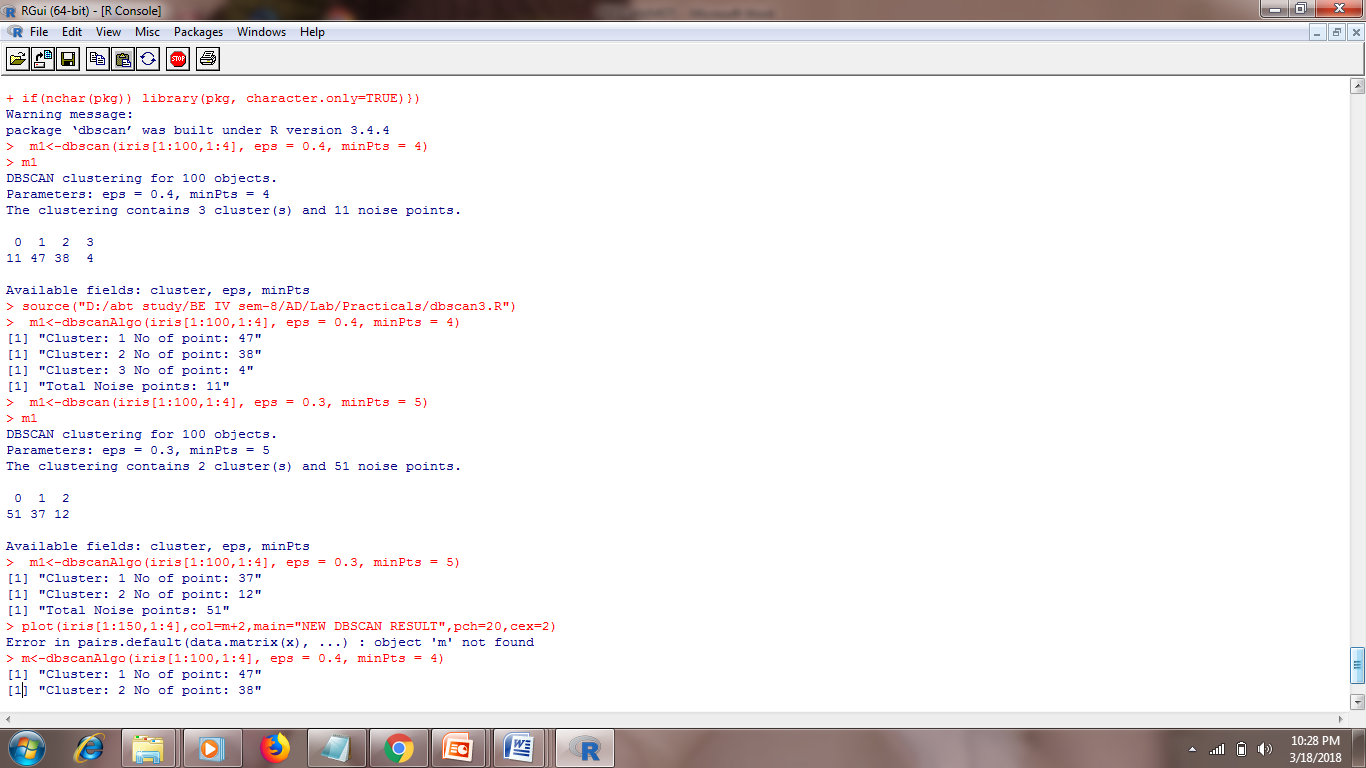
}

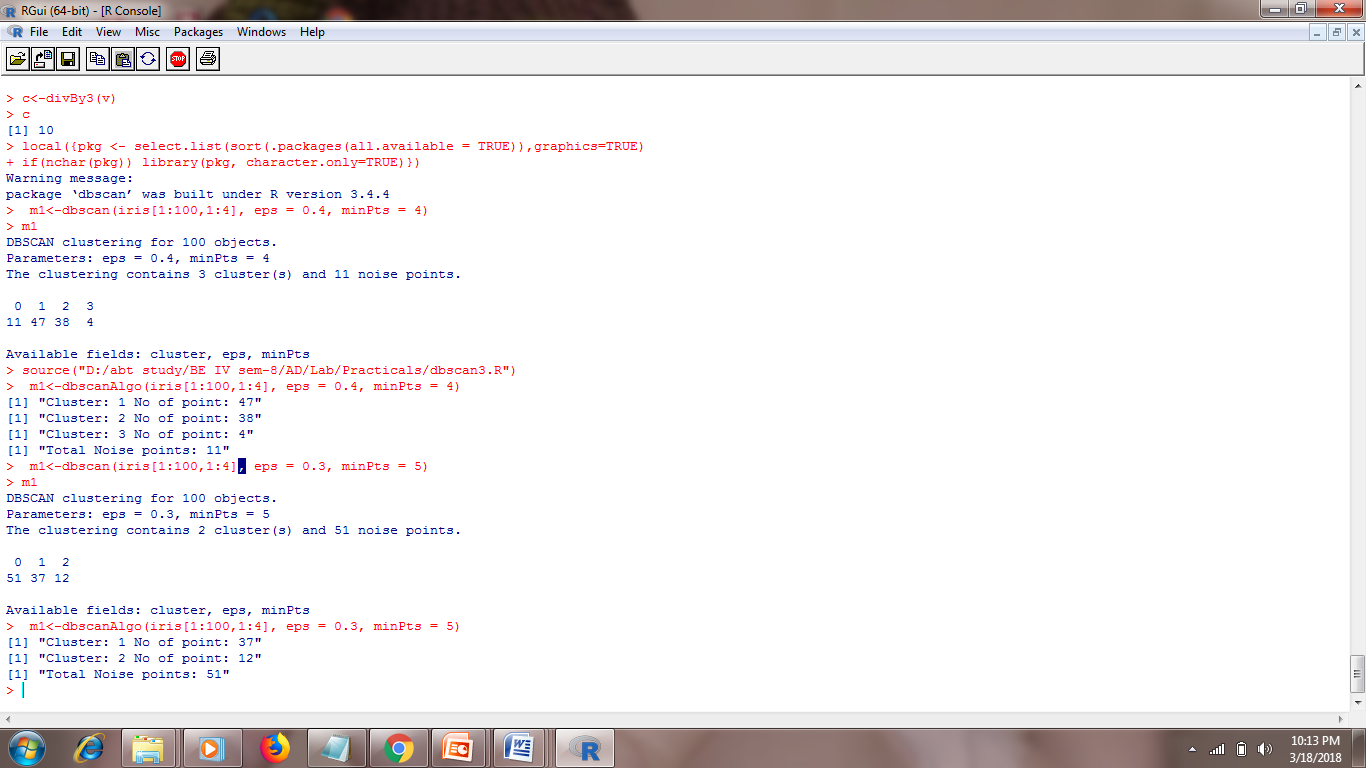
print(paste("Total Noise points:",noiseptscount))

return (clusterno)

}

**Output**

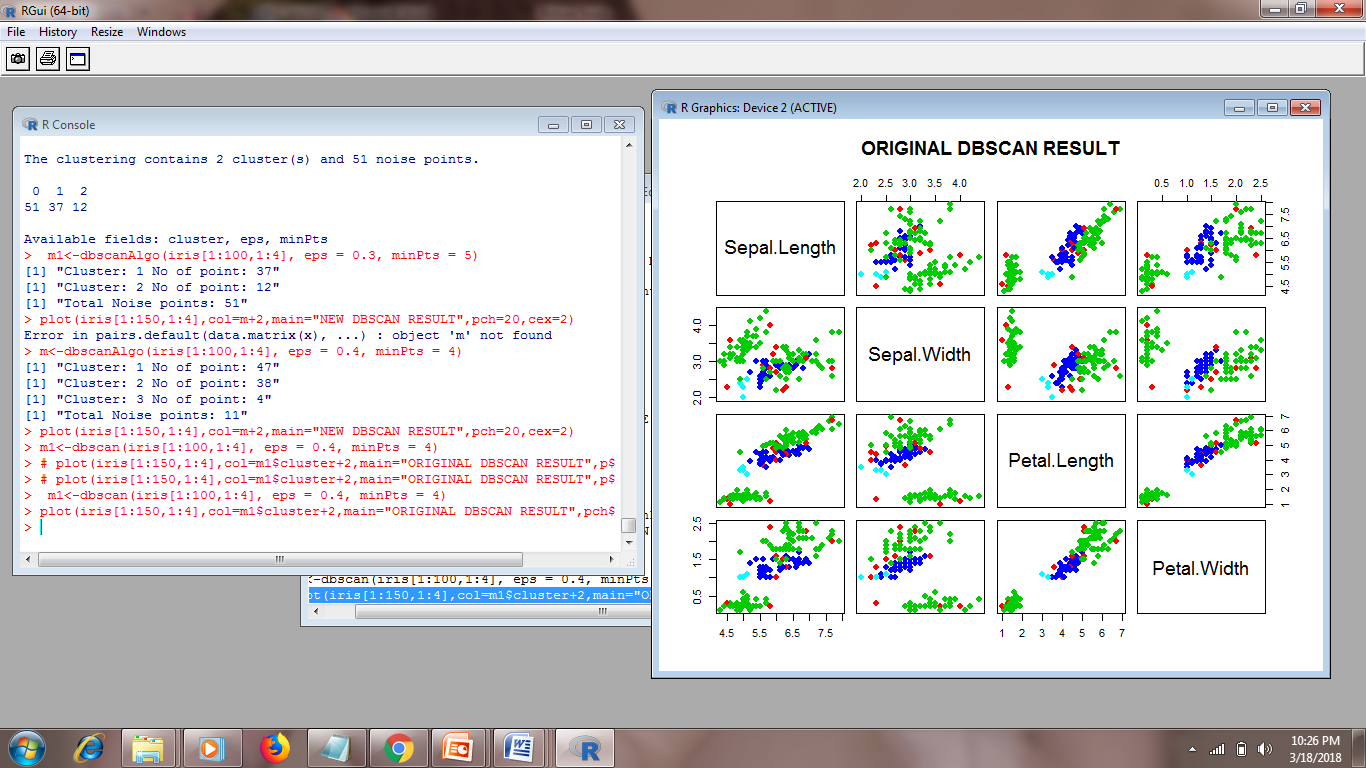
****

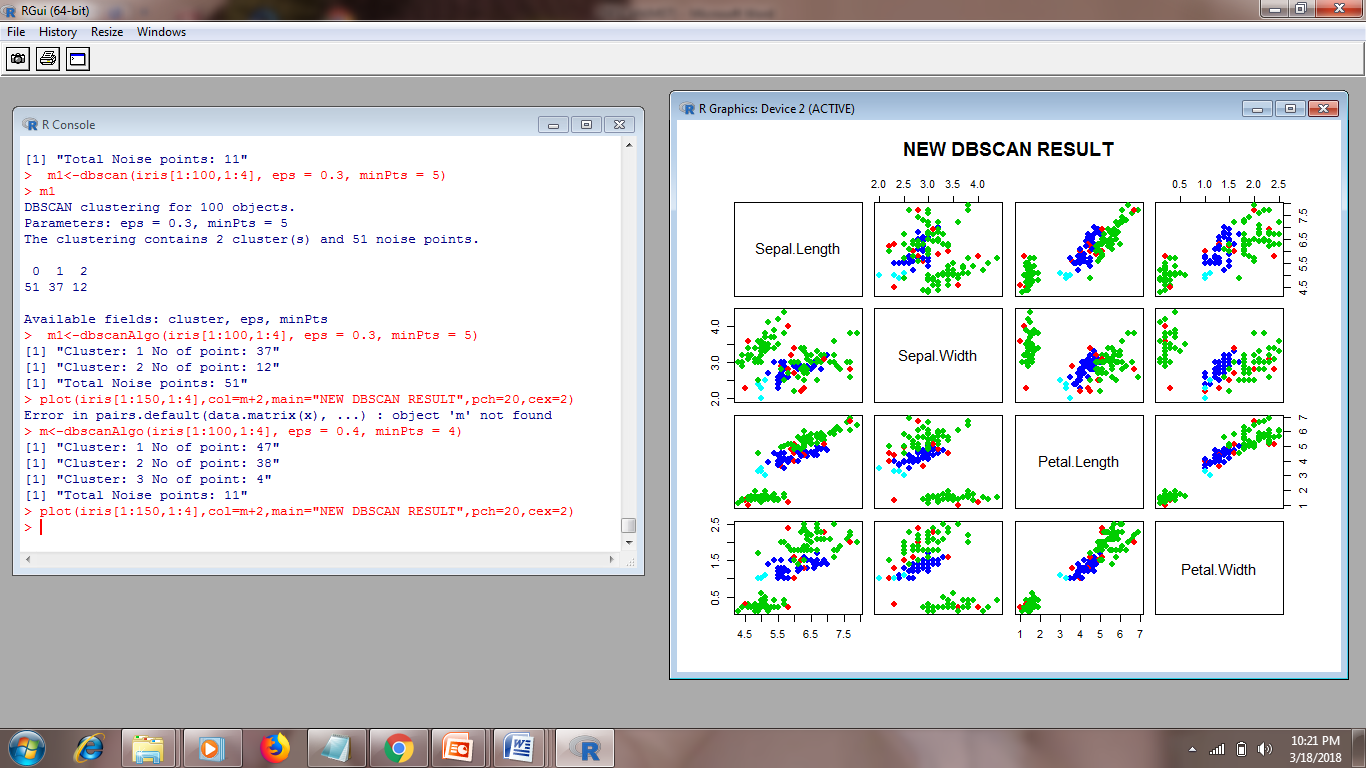
****

**Space Complexity :** o(n2)

**Time Complexity :** o(n2)

**Graphs:**

****

****

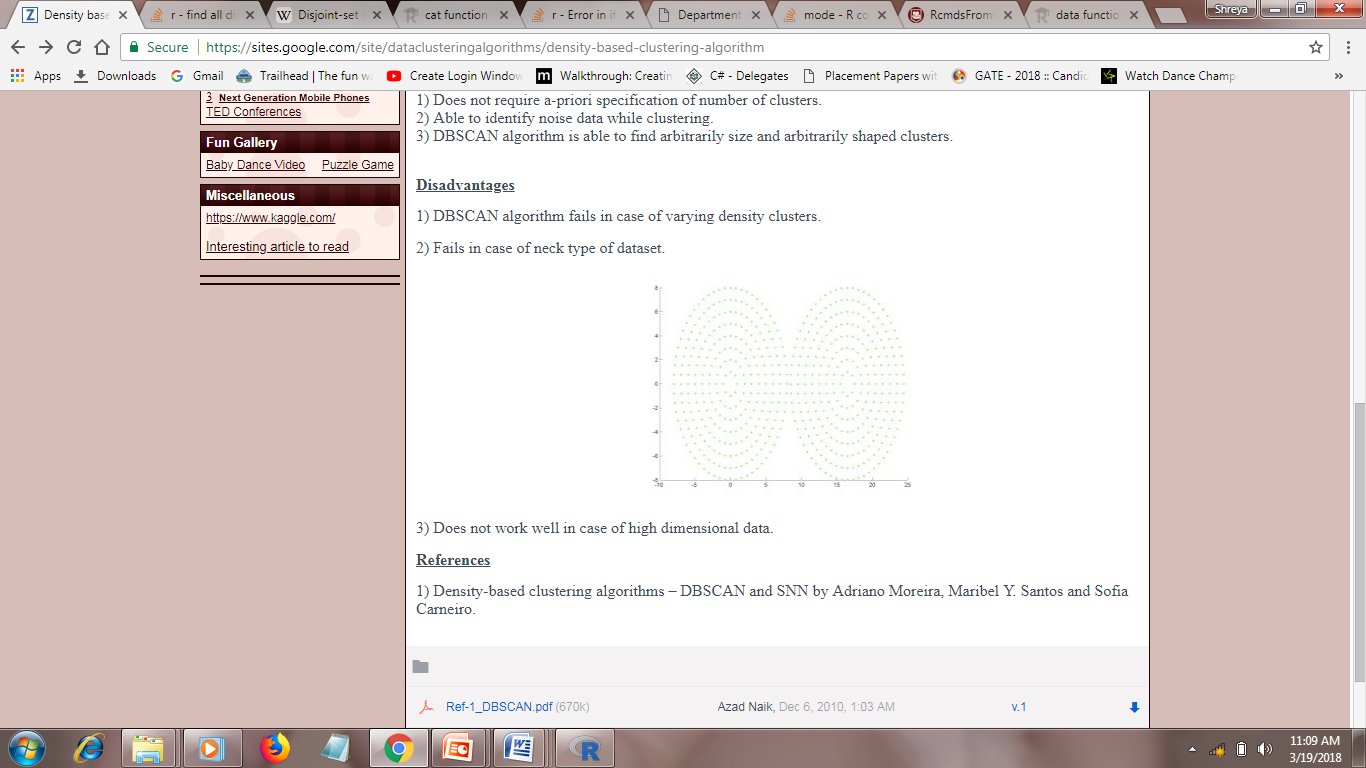
**Advantages**

1) Does not require a-priori specification of number of clusters.  
2) Able to identify noise data while clustering.  
3) DBSCAN algorithm is able to find arbitrarily size and arbitrarily shaped clusters.

**Disadvantages**

1) DBSCAN algorithm fails in case of varying density clusters.

2) Fails in case of neck type of dataset.



3)Does not work well in case of high dimensional data.

***k*-Means: A Centroid-Based Technique**

The *k*-means algorithm defines the centroid of a cluster as the mean value of the points within the cluster. It proceeds as follows. First,it randomly selects *k* of the objects in *D*, each of which initially represents a cluster meanor center. For each of the remaining objects, an object is assigned to the cluster to whichit is the most similar, based on the Euclidean distance between the object and the clustermean.

The *k*-means algorithm then iteratively improves the within-cluster variation.

For each cluster, it computes the new mean using the objects assigned to the cluster in

the previous iteration. All the objects are then reassigned using the updated means as

the new cluster centers. The iterations continue until the assignment is stable, that is,

the clusters formed in the current round are the same as those formed in the previous

round.

**Algorithm: *k***-**means.** The *k*-means algorithm for partitioning, where each cluster’s center

is represented by the mean value of the objects in the cluster.

**Input:**

*k*: the number of clusters,

*D*: a data set containing *n* objects.

**Output:** A set of *k* clusters.

**Method:**

(1) arbitrarily choose *k* objects from *D* as the initial cluster centers;

(2) **repeat**

(3) (re)assign each object to the cluster to which the object is the most similar,

based on the mean value of the objects in the cluster;

(4) update the cluster means, that is, calculate the mean value of the objects for

each cluster;

(5) **until** no change;

**Program:**

kmean <- function(data,k){

data <- as.data.frame(data)

#vector to store cluster number for each object

clustring\_vector <- vector(length=nrow(data))

#selecting first k object

centroid <- data[sample(1:nrow(data),size=k),]

centroid\_Old <- data[sample(1:nrow(data),size=k),]

#repeat for iteration

repeat{

#if old cluster mean and new are same stop algo

if(all(centroid==centroid\_Old)){break()}

#loop for finding clustring\_vector for each object

for(i in 1:nrow(data)){

#vector to store distance from each centroid

distance <- c()

for(j in 1:nrow(centroid)){

distance <- c(distance,dist(rbind(data[i,],centroid[j,]),method="euclidean"))

}

#stroring minimum distance cluster index

clustring\_vector[i] <- order(distance)[1]

}

#loops for finding new cluster mean

centroid\_Old <- centroid

for(j in 1:k){

#temporary data frame to store object of particular cluster to find their mean

temp <- data.frame()

#vector fot storing mean for each column

mn <- c()

#loop for storing object of particular cluster in `temp`

for(i in 1:nrow(data)){

if(clustring\_vector[i] == j){

temp <- rbind(temp,data[i,])

}

}

#using apply

mn <- apply(temp[,1:ncol(temp)],2,mean)

#print(temp)

tdf <- data.frame()

tdf<- rbind(tdf,mn)

centroid[j,] <- tdf

}

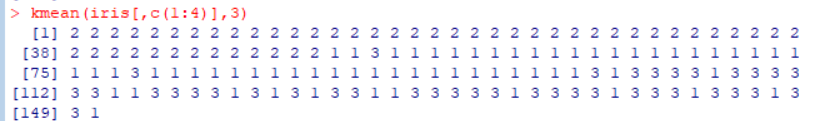
}

plot(data,col=clustring\_vector,main="K - Mean", pch=20, cex=2)

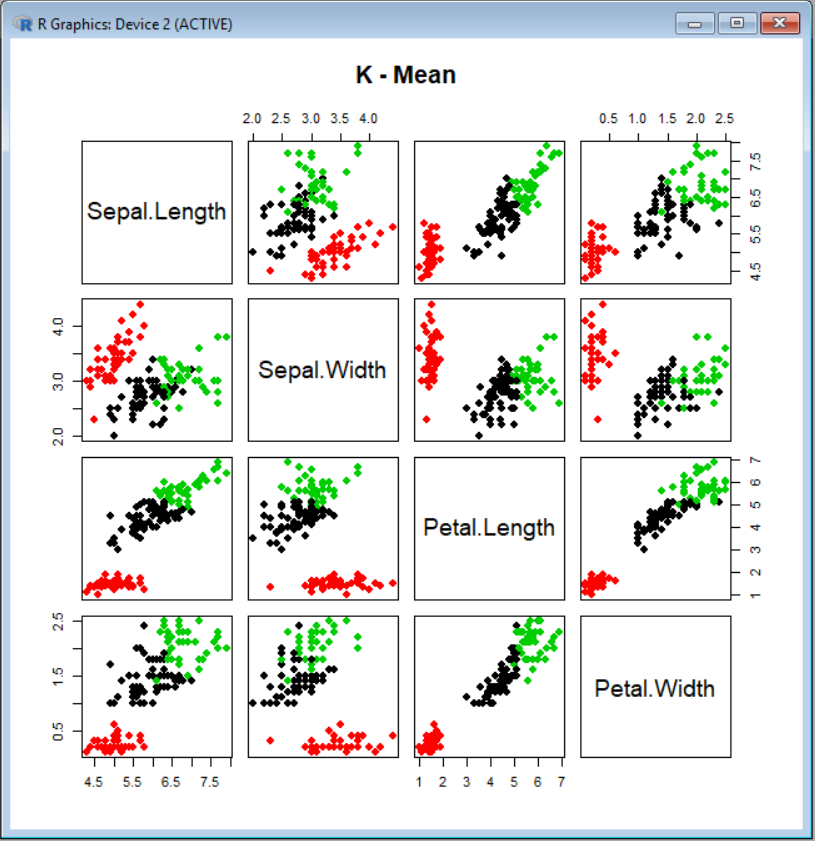
return (clustring\_vector)

}

**Output:**

****

**Output Graph:**

****

**K-Means Advantages :**1) If variables are huge, then  K-Means most of the times computationally faster than hierarchical clustering, if we keep k smalls.  
  
2) K-Means produce tighter clusters than hierarchical clustering, especially if the clusters are globular.  
  
**K-Means Disadvantages :**1) Initially selected k-values should not belong to same cluster.  
2) K-mean only detect spherical cluster.  
3) K-mean does not detect noise or outliers.

**Complexity of k-mean:**

**Time complexity**: **O(kn)**

where k = number of cluster

n = number of data items

**Space complexity: O(n)**

Where n = number of data items

Shared Nearest Neighbours (sNN)

**Algorithm**

matchCol <- function(x,y,nbr,t){

j=0

for(i in 1:ncol(nbr)){

if((nbr[x,i] == nbr[y,i]) && nbr[x,i]==TRUE){

j = (j+1)

}

}

if(j>=t){

return (TRUE)

}

else{

return (FALSE)

}

}

mySNN <- function(x,eps,minPts,srdPts){

#no. of rows cols

row <- nrow(x)

# all distances

distance <- as.matrix(dist(x,"euclidean"))

# neighbour[i,j] contains 1 if j is in neighborhood of i, OR 0

neighbour <- matrix(rep(FALSE,row\*row),nrow=row,ncol=row)

# cluster[i] contains in which cluster record i belongs

cluster <- rep(1,row)

# status[i] contains 0,1,2 if i is..

# 1 if Core point

# 2 if Border point

# 0 if Noise point

status <- rep(0,row)

# find core points

# core point vector

corepts<-c()

for(i in 1:row){

count = 0

for(j in 1:row){

if(i != j && distance[i,j] <= eps){

neighbour[i,j] = TRUE

count = count + 1

}

}

if(count >= minPts){

status[i] <- 1

corepts = union(corepts,i)

}

}

if(length(corepts)==0){

return (cat("No core point found!!\nChoose eps,minPts properly."))

}

# mark border point

for(i in corepts){

for(j in 1:row){

if(neighbour[i,j] && status[j] != 1){

status[j] = 2

}

}

}

# merge clusters

stack = c()

unvisited = corepts

cl=2

while(length(unvisited)!=0){

#push 1st of unvisited

stack = union(stack,unvisited[1])

unvisited = unvisited[-1]

while(length(stack)!=0){

#pop last into temp1

temp1 = stack[length(stack)]

stack = stack[-length(stack)]

for(i in unvisited){

if(neighbour[temp1,i] && i!=temp1 && matchCol(temp1,i,neighbour,srdPts)){

stack = union(stack,i)

unvisited = unvisited[-match(i,unvisited)]

# 1st point

cluster[temp1] = cl

for(j in 1:length(cluster)){

if(status[j]==2 && neighbour[temp1,j]){

cluster[j] = cl

}

}

# 2nd point

cluster[i] = cl

for(j in 1:length(cluster)){

if(status[j]==2 && neighbour[i,j]){

cluster[j] = cl

}

}

}

}

}

cl = cl + 1

}

#print(distance)

#print(neighbour)

cat("\nSTATUS\n")

print(status)

cat("\nCore Points\n")

print(corepts)

cat("\nCLUSTER\n")

print(cluster)

return (cluster)

}

**Time Complexity**

O(n2)

**Space Complexity**

O(n2)

Where, n = total records

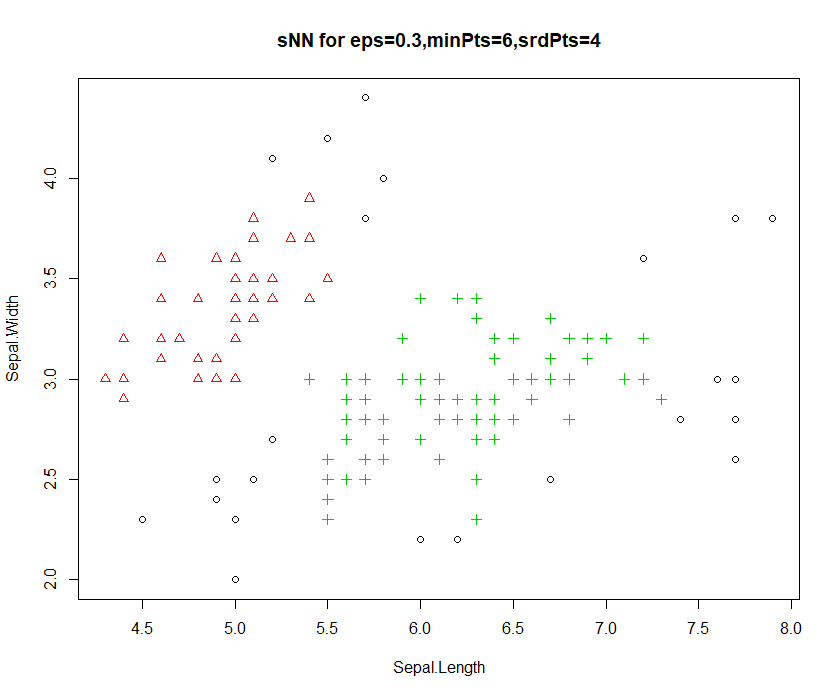
**Example**

y<-iris[,c(1,2)]

res=mySNN(y,0.3,6,4)

plot(y,col=res,pch=res,main="sNN for eps=0.3,minPts=6,srdPts=4")

**Output**



**K- NEAREST NEIGHBOURS**

library("xlsx")

knn.df=read.xlsx("Cryotherapy.xlsx", sheetIndex = 1)

#print(knn.df)

set.seed(100)

knn.df=knn.df[sample(nrow(knn.df)),]

train.df=knn.df[1:as.integer(0.7\*90),]

test.df=knn.df[as.integer(0.7\*90+1):90,]

knn\_predict <- function(test\_data, train\_data, k\_value){

pred <- c() #empty pred vector

#LOOP-1

for(i in c(1:nrow(test\_data))){ #looping over each record of test data

eu\_dist =c() #eu\_dist & eu\_char empty vector

eu\_char = c()

good = 0 #good & bad variable initialization with 0 value

bad = 0

#LOOP-2-looping over train data

for(j in c(1:nrow(train\_data))){

#adding euclidean distance b/w test data point and train data to eu\_dist vector

eu\_dist <- c(eu\_dist, dist(rbind(test\_data[i,], train\_data[j,]), method = "euclidean"))

#adding class variable of training data in eu\_char

eu\_char <- c(eu\_char, as.character(train\_data[j,][[7]]))

}

eu <- data.frame(eu\_char, eu\_dist) #eu dataframe created with eu\_char & eu\_dist columns

eu <- eu[order(eu$eu\_dist),] #sorting eu dataframe to gettop K neighbors

eu <- eu[1:k\_value,] #eu dataframe with top K neighbors

#print("eu is")

#print(eu)

#Loop 3: loops over eu and counts classes of neibhors.

for(k in c(1:nrow(eu))){

if(as.character(eu[k,"eu\_char"]) == 0){

good = good + 1

}

else

bad = bad + 1

}

# Compares the no. of neighbours with class label good or bad

if(good > bad){ #if majority of neighbours are good then put "g" in pred vector

pred <- c(pred, 0)

}

else if(good < bad){

#if majority of neighbors are bad then put "b" in pred vector

pred <- c(pred, 1)

}

}

return(pred) #return pred vector

}

accuracy <- function(test\_data){

correct = 0

for(i in c(1:nrow(test\_data))){

if(test\_data[i,7] == test\_data[i,8]){

correct = correct+1

}

}

accu = correct/nrow(test\_data) \* 100

return(accu)

}

confusion<- function(test\_data){

nono=0

noyes=0

yesno=0

yesyes=0

for(i in c(1:nrow(test\_data))){

if(test\_data[i,7] == 0 && test\_data[i,8]==0){

nono = nono+1

}else if(test\_data[i,7] == 0 && test\_data[i,8]==1){

noyes = noyes+1

}else if(test\_data[i,7] == 1 && test\_data[i,8]==0){

yesno = yesno+1

}else if(test\_data[i,7] == 1 && test\_data[i,8]==1){

yesyes = yesyes+1

}

}

n=nrow(test\_data)

conf=matrix(,nr=3,nc=3)

conf[1,1]=n

conf[1,2]="Predicted Yes"

conf[1,3]="Predicted No"

conf[2,1]="Actual Yes"

conf[2,2]= yesyes

conf[2,3]= yesno

conf[3,1]="Actual No"

conf[3,2]= noyes

conf[3,3]= nono

print(conf)

}

K = 3

predictions <- knn\_predict(test.df, train.df, K) #calling knn\_predict()

test.df[,8] <- predictions #Adding predictions in test data as 7th column

print(test.df)

print(accuracy(test.df))

#confusion(test.df)

**Confusion Matrix:**

K=3, Euclidean distance measure

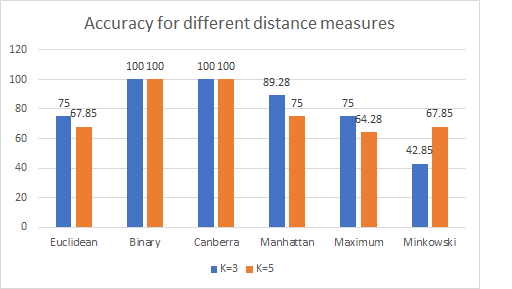
|  |  |  |  |
| --- | --- | --- | --- |
| N=28 | Predicted Yes | Predicted No |  |
| Actual Yes | 11 | 3 | 14 |
| Actual No | 4 | 10 | 14 |

K=5, Euclidean distance measure

|  |  |  |  |
| --- | --- | --- | --- |
| N=28 | Predicted Yes | Predicted No |  |
| Actual Yes | 8 | 6 | 14 |
| Actual No | 3 | 11 | 14 |

**Accuracy table for different distance measures:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Distance** | **Euclidean** | **Binary** | **Canberra** | **Manhattan** | **Maximum** | **Minkowski** |
| **K=3** | 75 | 100 | 100 | 89.28 | 75 | 42.85 |
| **K=5** | 67.85 | 100 | 100 | 75 | 64.28 | 67.85 |



**Advantages:**

* Very simple and easy to implement
* Works on basic recognition problems

**Disadvantages:**

* It is lazy learning algorithm
* The value of k needs to be known before hand
* It performs sorting on the data which is time consuming
* It is not robust to noisy data

**Modified KNN Analysis**

**K-Value: K Nearest Neighbors.**

**T-Value: Number of Tuples from each class which will be used to calculate distance from test data.**

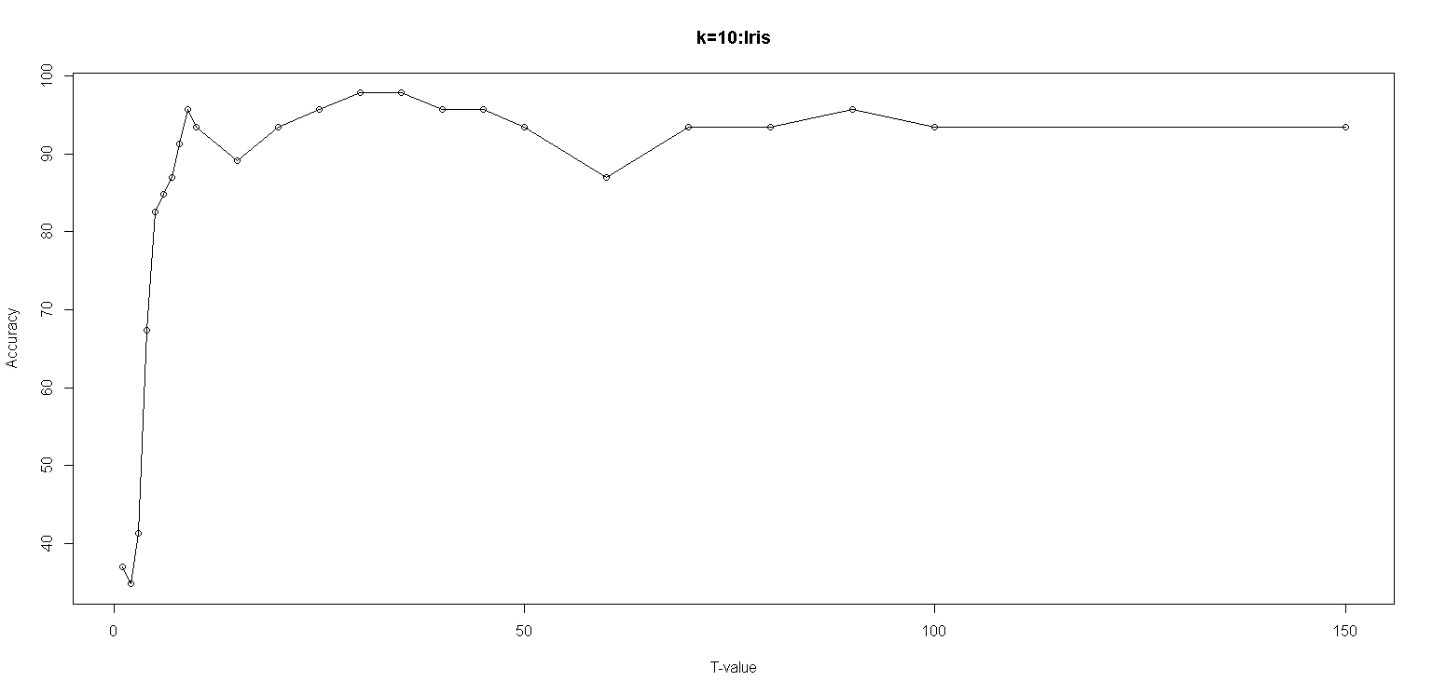
**Accuracy: (TRUE Predictions/Total Predictions)\*100.**

**Modification:**

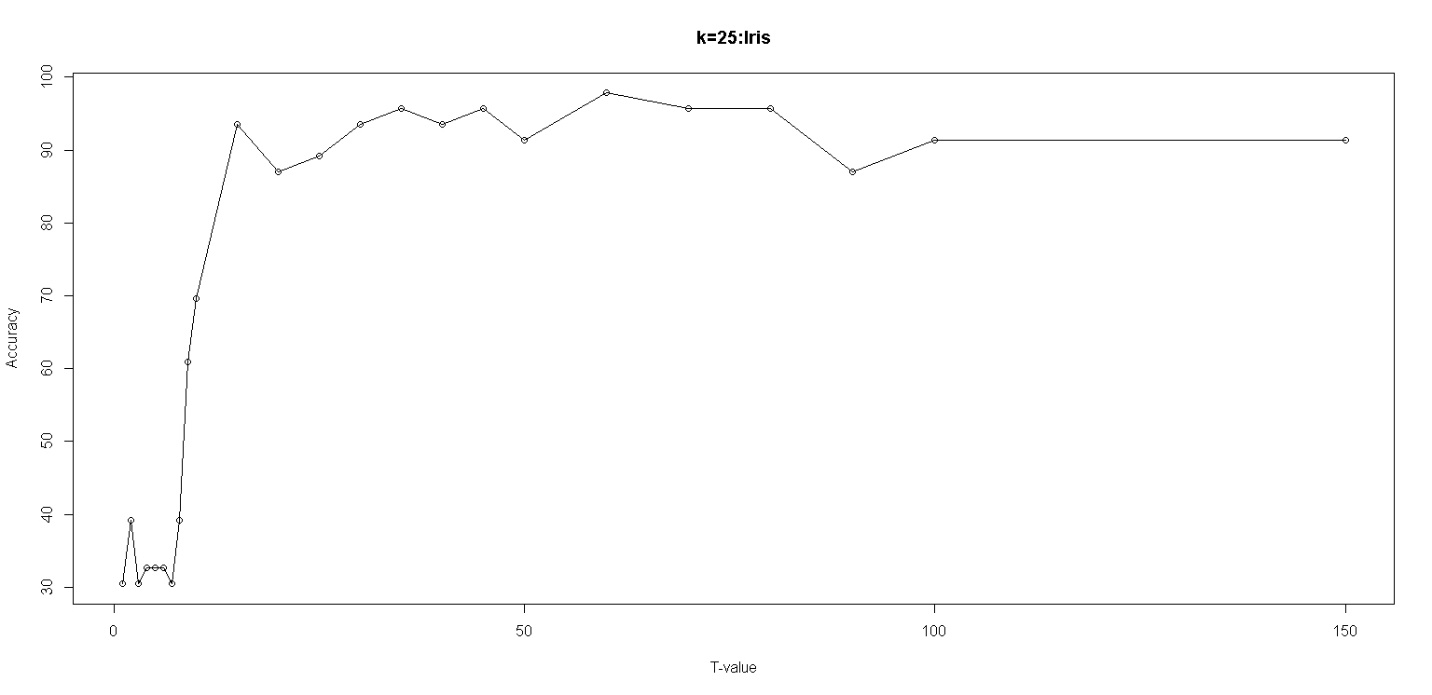
**Modified KNN covers the ability of simple KNN. Moreover, it adds extra features. Instead of calculating distances with all the train data elements, modified\_knn extracts ‘t’ tuples from each class from train data set. The distance of test data is calculated with these tuples.**

**DATASET: Iris**

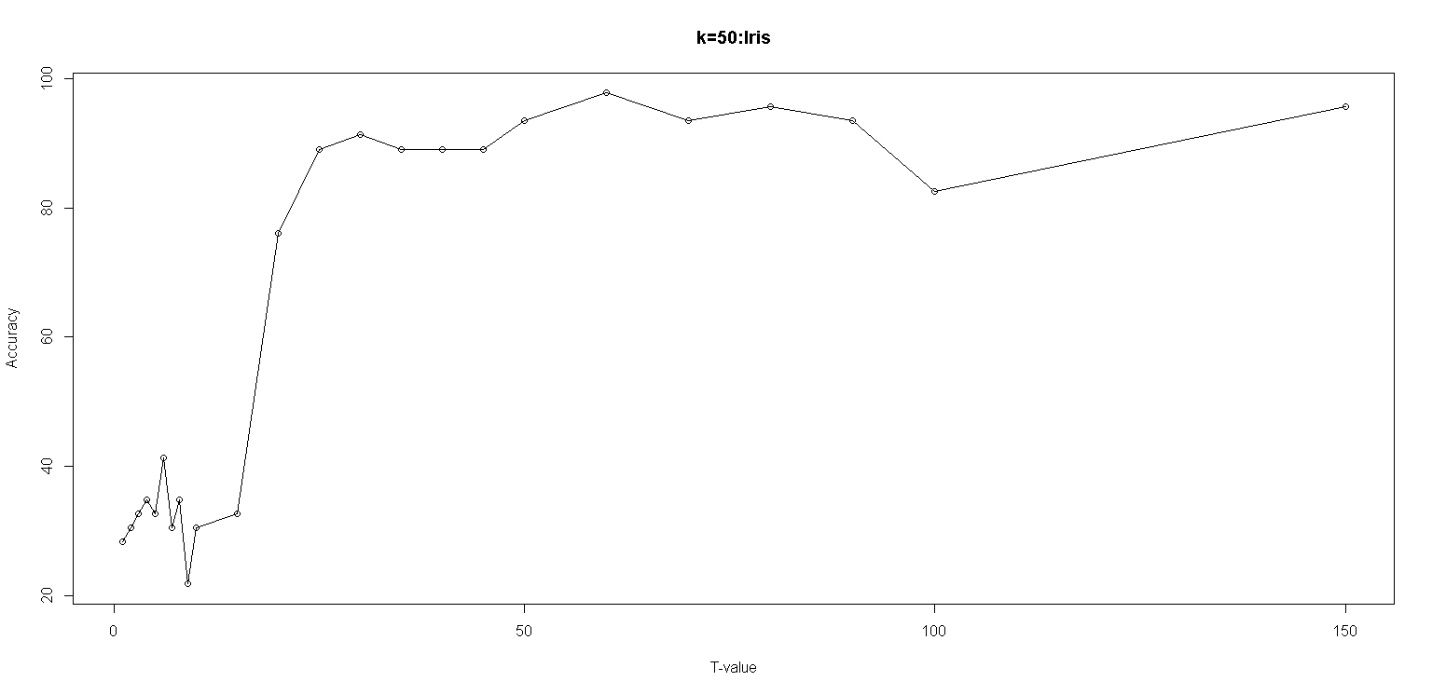
|  |  |  |
| --- | --- | --- |
| K-Value | T-Value | Accuracy |
| 10 | 1 | 36.95652 |
| 10 | 2 | 34.78261 |
| 10 | 3 | 41.30435 |
| 10 | 4 | 67.39130 |
| 10 | 5 | 82.60870 |
| 10 | 6 | 84.78261 |
| 10 | 7 | 86.95652 |
| 10 | 8 | 91.30435 |
| 10 | 9 | 95.65217 |
| 10 | 10 | 93.47826 |
| 10 | 15 | 89.13043 |
| 10 | 20 | 93.47826 |
| 10 | 25 | 95.65217 |
| 10 | 30 | 97.82609 |
| 10 | 35 | 97.82609 |
| 10 | 40 | 95.65217 |
| 10 | 45 | 95.65217 |
| 10 | 50 | 93.47826 |
| 10 | 60 | 86.95652 |
| 10 | 70 | 93.47826 |
| 10 | 80 | 93.47826 |
| 10 | 90 | 95.65217 |
| 10 | 100 | 93.47826 |
| 10 | 150 | 93.47826 |



|  |  |  |
| --- | --- | --- |
| K-Value | T-Value | Accuracy |
| 25 | 1 | 30.43478 |
| 25 | 2 | 39.13043 |
| 25 | 3 | 30.43478 |
| 25 | 4 | 32.60870 |
| 25 | 5 | 32.60870 |
| 25 | 6 | 32.60870 |
| 25 | 7 | 30.43478 |
| 25 | 8 | 39.13043 |
| 25 | 9 | 60.86957 |
| 25 | 10 | 69.56522 |
| 25 | 15 | 93.47826 |
| 25 | 20 | 86.95652 |
| 25 | 25 | 89.13043 |
| 25 | 30 | 93.47826 |
| 25 | 35 | 95.65127 |
| 25 | 40 | 93.47826 |
| 25 | 45 | 95.65217 |
| 25 | 50 | 91.30435 |
| 25 | 60 | 97.82609 |
| 25 | 70 | 95.65217 |
| 25 | 80 | 95.65217 |
| 25 | 90 | 86.9562 |
| 25 | 100 | 91.30435 |
| 25 | 150 | 91.30435 |

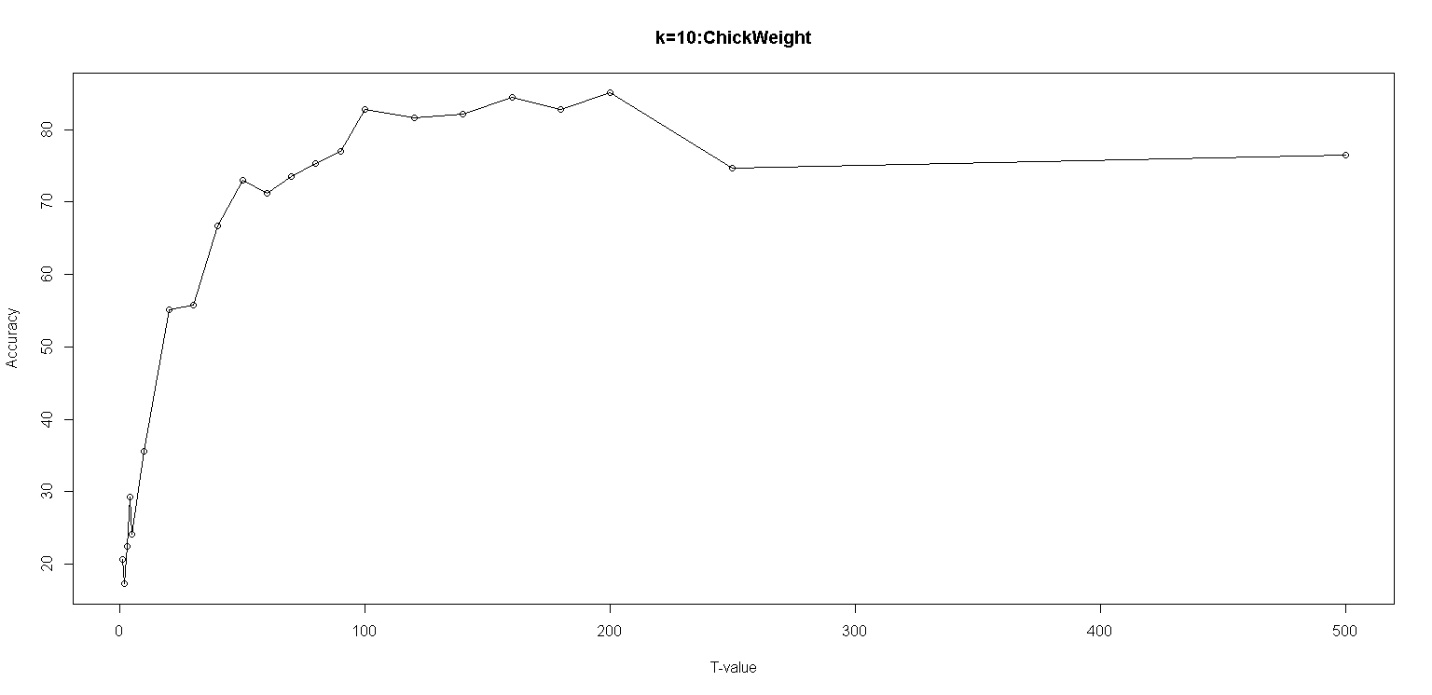


|  |  |  |
| --- | --- | --- |
| K-Value | T-Value | Accuracy |
| 50 | 1 | 28.26087 |
| 50 | 2 | 30.43478 |
| 50 | 3 | 32.60870 |
| 50 | 4 | 34.78261 |
| 50 | 5 | 32.60870 |
| 50 | 6 | 41.30435 |
| 50 | 7 | 30.43478 |
| 50 | 8 | 34.78261 |
| 50 | 9 | 21.73913 |
| 50 | 10 | 30.43478 |
| 50 | 15 | 32.60870 |
| 50 | 20 | 76.08696 |
| 50 | 25 | 89.13043 |
| 50 | 30 | 91.30435 |
| 50 | 35 | 89.13043 |
| 50 | 40 | 89.13043 |
| 50 | 45 | 89.13043 |
| 50 | 50 | 93.47826 |
| 50 | 60 | 97.82609 |
| 50 | 70 | 93.47826 |
| 50 | 80 | 95.65217 |
| 50 | 90 | 93.47826 |
| 50 | 100 | 82.60870 |
| 50 | 150 | 95.65217 |

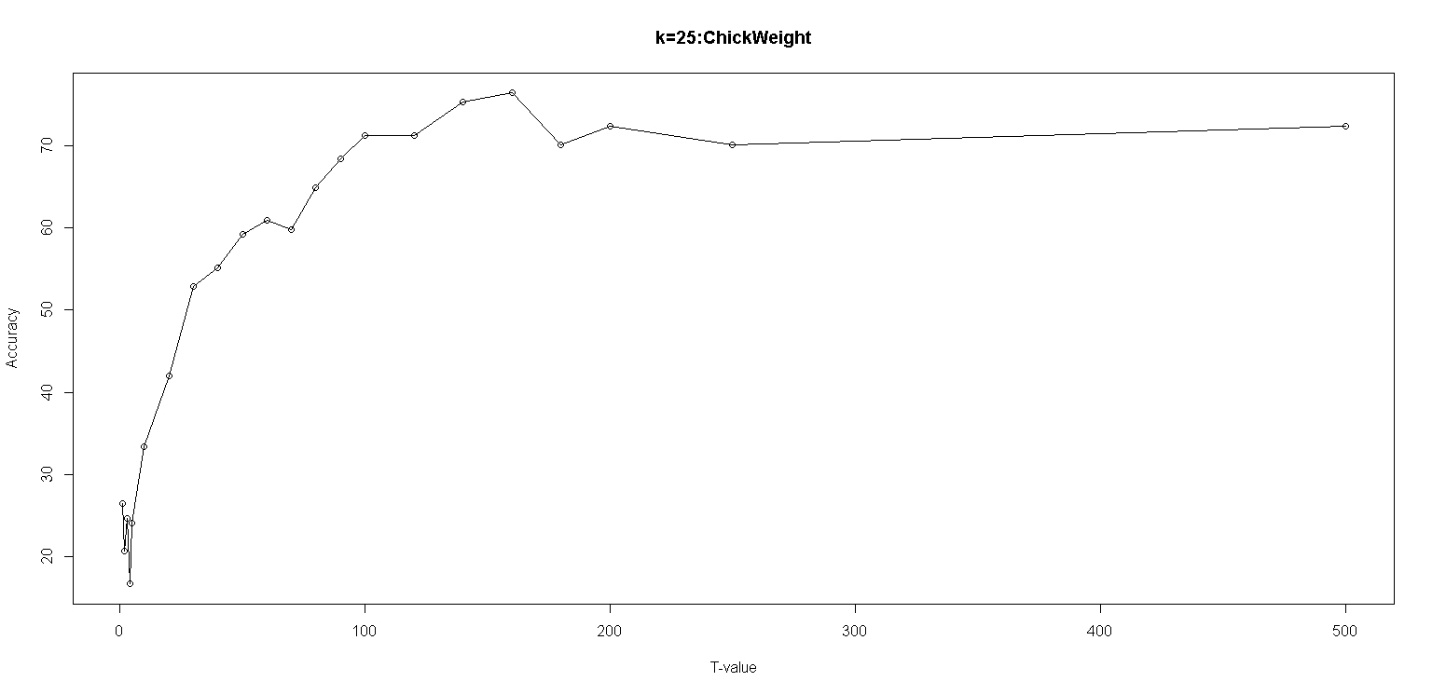


**DATASET: ChickWeight**

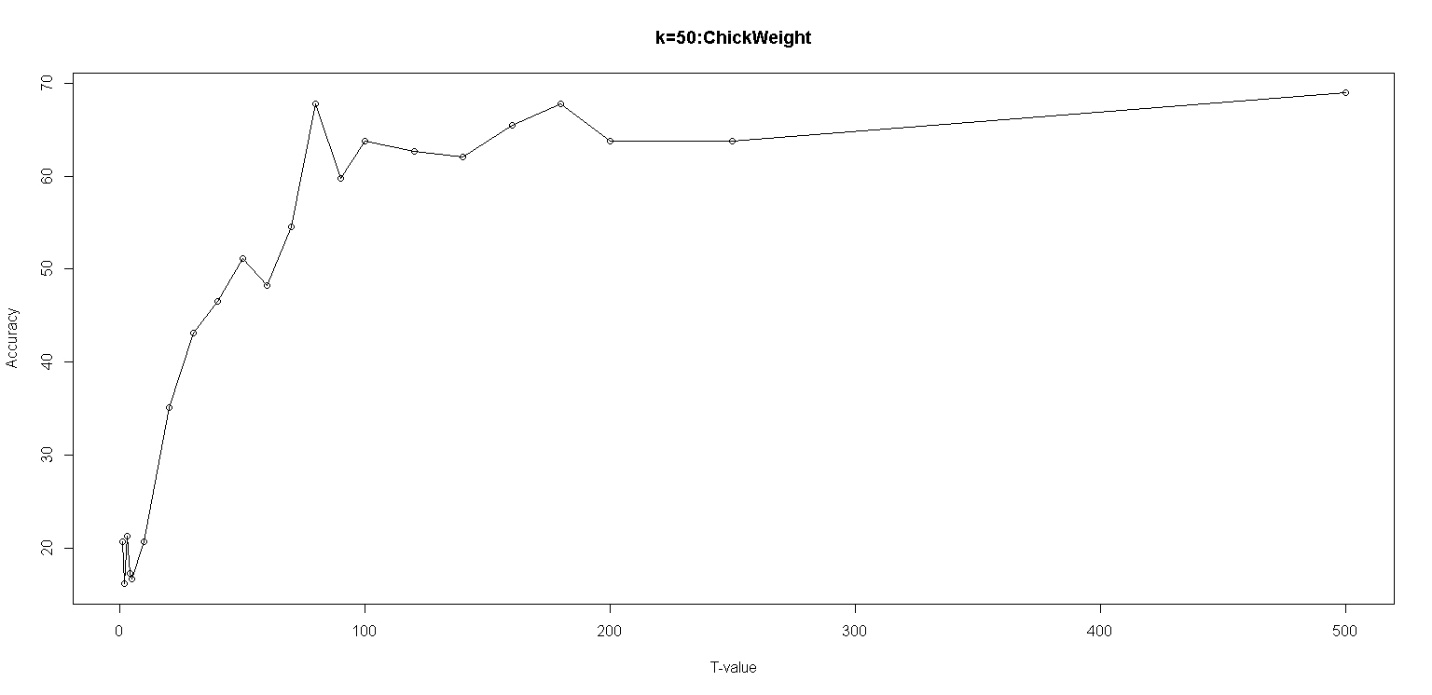
|  |  |  |
| --- | --- | --- |
| K-Value | T-Value | Accuracy |
| 10 | 1 | 20.68966 |
| 10 | 2 | 17.24138 |
| 10 | 3 | 22.41739 |
| 10 | 4 | 29.31034 |
| 10 | 5 | 24.13793 |
| 10 | 10 | 35.63218 |
| 10 | 20 | 55.17241 |
| 10 | 30 | 55.74713 |
| 10 | 40 | 66.66667 |
| 10 | 50 | 72.98891 |
| 10 | 60 | 71.26437 |
| 10 | 70 | 73.56322 |
| 10 | 80 | 75.28736 |
| 10 | 90 | 77.01149 |
| 10 | 100 | 82.75862 |
| 10 | 120 | 81.60920 |
| 10 | 140 | 82.18391 |
| 10 | 160 | 84.48276 |
| 10 | 180 | 82.75862 |
| 10 | 200 | 85.05747 |
| 10 | 250 | 74.71264 |
| 10 | 500 | 76.43678 |



|  |  |  |
| --- | --- | --- |
| K-Value | T-Value | Accuracy |
| 25 | 1 | 26.43678 |
| 25 | 2 | 20.68966 |
| 25 | 3 | 24.71264 |
| 25 | 4 | 16.66667 |
| 25 | 5 | 24.13793 |
| 25 | 10 | 33.33333 |
| 25 | 20 | 41.95402 |
| 25 | 30 | 52.87356 |
| 25 | 40 | 55.17241 |
| 25 | 50 | 59.19540 |
| 25 | 60 | 60.91954 |
| 25 | 70 | 59.77011 |
| 25 | 80 | 64.94253 |
| 25 | 90 | 68.39080 |
| 25 | 100 | 71.26437 |
| 25 | 120 | 71.26437 |
| 25 | 140 | 75.28736 |
| 25 | 160 | 76.43678 |
| 25 | 180 | 70.11494 |
| 25 | 200 | 72.41379 |
| 25 | 250 | 70.11494 |
| 25 | 500 | 72.41379 |

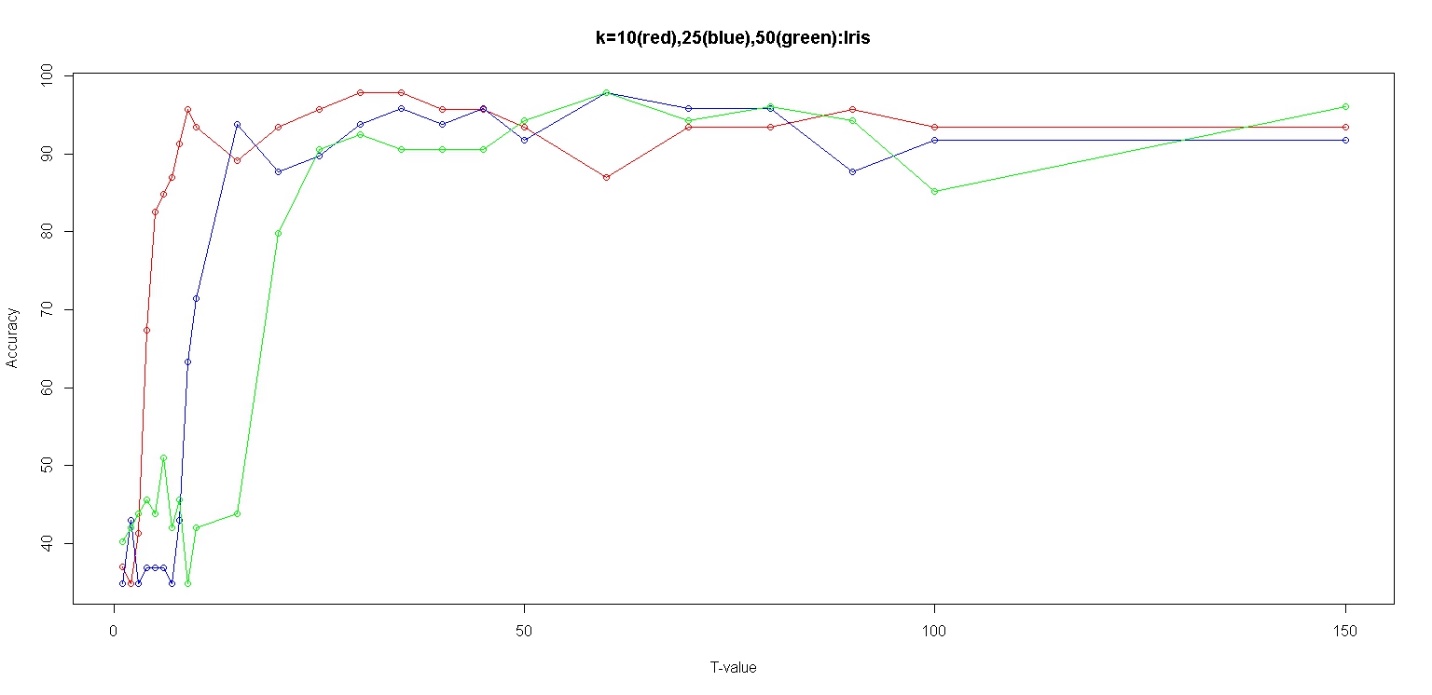


|  |  |  |
| --- | --- | --- |
| K-Value | T-Value | Accuracy |
| 50 | 1 | 20.68966 |
| 50 | 2 | 16.09195 |
| 50 | 3 | 21.26437 |
| 50 | 4 | 17.24138 |
| 50 | 5 | 16.66667 |
| 50 | 10 | 20.68966 |
| 50 | 20 | 35.05747 |
| 50 | 30 | 43.10345 |
| 50 | 40 | 46.55172 |
| 50 | 50 | 51.14943 |
| 50 | 60 | 48.27586 |
| 50 | 70 | 54.59770 |
| 50 | 80 | 67.81609 |
| 50 | 90 | 59.77011 |
| 50 | 100 | 63.79310 |
| 50 | 120 | 62.64368 |
| 50 | 140 | 62.06897 |
| 50 | 160 | 65.51274 |
| 50 | 180 | 67.81609 |
| 50 | 200 | 63.79310 |
| 50 | 250 | 63.79310 |
| 50 | 500 | 68.96552 |

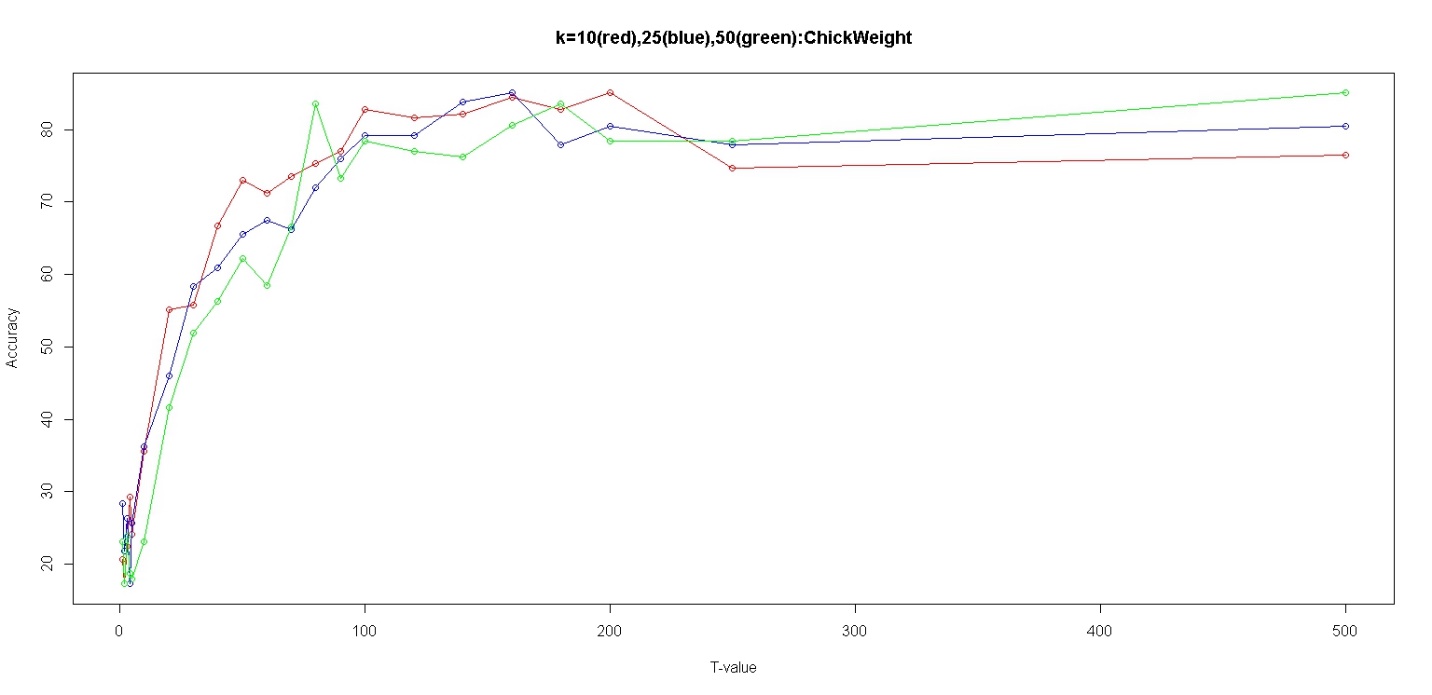


**COMBINED GRAPH ANALYSIS**

1. **IRIS**



1. **CHICKWEIGHT**

****

**Advantages over simple KNN:**

1. **Proper Handling of Biased Dataset:**

**Suppose, a data set has 1000 tuples and 3 classes.**

**500 tuples belong to class1.**

**400 tuples belong to class2.**

**100 tuples belong to class3.**

**When a tuple from test data set is computed for distances with all 1000 tuples in training data set. Biasing is most probable.**

**In Modified\_KNN, if we select T-Value = 50.**

**50 tuples from each of the classes i.e. class1, class2, class3 will be extracted at random and with these tuples the test data distance computation will take place.**

**This approach eliminates possibility of biasing and sheds improved results.**

1. **Reduced Time Complexity:**

**From the graphs above, it can be inferred that accuracy gets saturated after a threshold value is met.**

**There is no need to process entire train data set for each tuple. Less number of tuples give the same accurate results.**

1. **Reduced Space Complexity:**

**Obviously, if we process less number of tuples, temporary storage requirement also reduces proportionally. Therefore, Space Complexity is also reduced.**

**Code:**

knn.df <- read.csv('D:\\Study\\Projects\\AD\\iris-species\\Iris.csv', header = FALSE, sep = ',')

class\_labels<-c("Iris-setosa","Iris-versicolor","Iris-virginica")

#chick.df<- ChickWeight

#class\_labels<-c(1,2,3,4)

Mode = function(x){

ta = table(x)

tam = max(ta)

mod = 0

if (all(ta == tam))

mod = names(ta)[ta == tam]

else

if(is.numeric(x))

mod = as.numeric(names(ta)[ta == tam])

else

mod = names(ta)[ta == tam]

return(mod)

}

#my knn

#data : datase,t: no of tuples to extract, class\_labels: vector\_to\_store\_labels,col: column\_no having class\_values

restricted\_knn <- function(data,k,t,class\_lablels,col){

#sampling data & constructing train and test datasets

data<- data[sample(nrow(data)),]

train\_data <- data[1:as.integer(0.7\*nrow(data)),]

test\_data <- data[as.integer(0.7\*nrow(data) +1):nrow(data),]

#predicted class values for test\_data

pred\_col <- c()

#loop runs for each tuple in test\_data

for(i in c(1:nrow(test\_data))){

#different class data\_sets

a<-data.frame()

#extract t tuples of each class from train\_data

for(x in c(1:length(class\_labels)))

{

temp <- train\_data[train\_data[col] == class\_labels[x],]

if(nrow(temp) > t)

temp <- temp[sample(nrow(temp)),]

temp <- temp[c(1:t),]

a<-rbind(a,temp)

}

#distance: distance b/w cur\_test\_data and all\_train\_data

distance <- c()

#class: respective train\_data classes

class <- c()

for(j in 1:nrow(a)){

#removing the col that contains class values

v<- c(1:ncol(a))

v<- v[-col]

m <- matrix(c(test\_data[i,v],a[j,v]),ncol=ncol(a)-1,byrow=TRUE)

distance <- c(distance,dist(m,method="euclidean"))

class <- c(class,as.character(a[j,][[col]]))

}

#merging distance and class in one data frame and then sorting wrt distance in ascending order

eu <- data.frame(distance,class)

eu <- eu[order(eu$distance),]

#selecting top k distances and their classes

if(k < nrow(eu))

eu <- eu[1:k,]

pred\_col <- c(pred\_col,max(Mode(eu$class)))

}

test\_data["PREDICTED\_CLASS"] <- NA

test\_data$PREDICTED\_CLASS <- pred\_col

test\_data["JUDGEMENT"] <- as.logical(test\_data[col] == test\_data$PREDICTED\_CLASS)

return(test\_data)

}

#col1 actual value column, col2 predicted value column

accuracy <- function(test\_data,col1,col2){

correct = 0

for(i in c(1:nrow(test\_data))){

if(test\_data[i,col1] == test\_data[i,col2]){

correct = correct+1

}

}

accu = correct/nrow(test\_data) \* 100

return(accu)

}

#tvec for ChickWeight

#tvec<-c(1,2,3,4,5,10,20,30,40,50,60,70,80,90,100,120,140,160,180,200,250,500)

#tvec for Iris

tvec<-c(1,2,3,4,5,6,7,8,9,10,15,20,25,30,35,40,45,50,60,70,80,90,100,150)

analysis<-function(data,col1,col2,k,tvec)

{

c1<-rep(k,length(tvec))

c3<-c()

for(i in c(1:length(tvec)))

{

a<-restricted\_knn(data,k,tvec[i],class\_labels,col1)

ac<-accuracy(a,col1,col2)

c3<-c(c3,ac)

}

pp<-cbind(c1,tvec,c3)

return(pp)

}

#PLOT GRAPHS

#plot(tvec,ansIris10[,3],main="k=10:Iris",xlab="T-value",ylab="Accuracy",type="o")