STAT37810 Final Project

Wenjing Xu, Yanfei Zhou, Yijia Zhao

Question 1 Metropolis-Hastings

Part 1

Implement Metropolis-Hastings algorithm to sample from a Beta distribution with parameters (6,4), we need the following steps.

1. Choose a starting value θ_0. This will be generated from a uniform distribution (as required by question setting).
2. At each iteration, draw a candidate θ^* from a jumping functin with the form $\phi_{prop}|\phi_{old}$ ~ $Beta(c\phi_{old}, c(1-\phi_{old})$. This will help us to define the proposal function.
3. Define an acceptence ratio, so that if we accept θ^* and add it into the chain with the probability given by min{1, acceptance ratio}.
4. If θ^* is not accepted, then stay at that point.
5. Repeat the process until θ does not change.

First we define the target function whose domain only takes values x from [0,1], and return the density function of beta(x,6,4), and return 0 when x>1 or x<0.

```
target = function(x){
  if (any(x>1) | any(x<0)){
    return(0)}
  else{
    return (dbeta(x,6,4))
  }
}
```

Then we define the proposal function which generate one sample from the distribution $\phi_{prop}|\phi_{old} \sim Beta(c\phi_{old}, c(1-\phi_{old}))$, where ϕ_{old} is the last sample we had and c is a constant.

```
proposalfunction<-function(x){
  return(rbeta(1,shape1=c*x,shape2=c*(1-x)))
}
```

The code for Metropolis-Hastings algorithm is shown as follows. Set the constant c to be 1, the number of iteration to be 10000, and randomly generate a first starting value from a uniform distribution $[0, 1]$

```
set.seed(1234)
num_iteration=10000
c=1
# initialize an arbitrary point to be the staring values, and set the number of iterations
start=runif(1,0,1)

run_metropolis_MCMC <- function(startvalue, iterations){
  # create a chain to store all the samples generated in each iterations
  chain = rep(0,iterations)
  # assign the first element in the chain to be the staring value
  chain[1] = startvalue
  for (i in 1:iterations){
    # set the last sample we generated in chain as currentx
    currentx = chain[i]
    # generate a candidate x* (here named proposal) using the jumping distribution given i
n the question, with parameter phi equals current x
    proposal = proposalfunction(chain[i])
    # calculate the acceptance ratio (here named probab), which will be used to decide whe
ther to accept or reject the candidate
    probab = (target(proposal)/target(chain[i]))*(dbeta(chain[i],shape1=c*proposal,shape2=
c*(1-proposal)/dbeta(proposal,shape1=c*chain[i],shape2=c*(1-chain[i]))))
    # generate a uniform random number u on [0,1], and compare u with probab to determine
 either to keep the  candidate in chain or drop it
    if (runif(1) < probab){
      # if u<= acceptance ratio, accept the candidate by setiing next element in the chain
 to proposal
      chain[i+1] = proposal
    }else{
      # if u<= acceptance ratio, reject the candidate proposal and set next element same a
s last element(currentx)
      chain[i+1] = chain[i]
    }
  }
  #return the chain which stores all generated samples from the target distribution
  return(chain)
}
```

Calculate how many different samples we have drawn from running the algorithm once, (equivalent to saying how many newly generated samples have been accepted) store the ratio as accpetance_rate

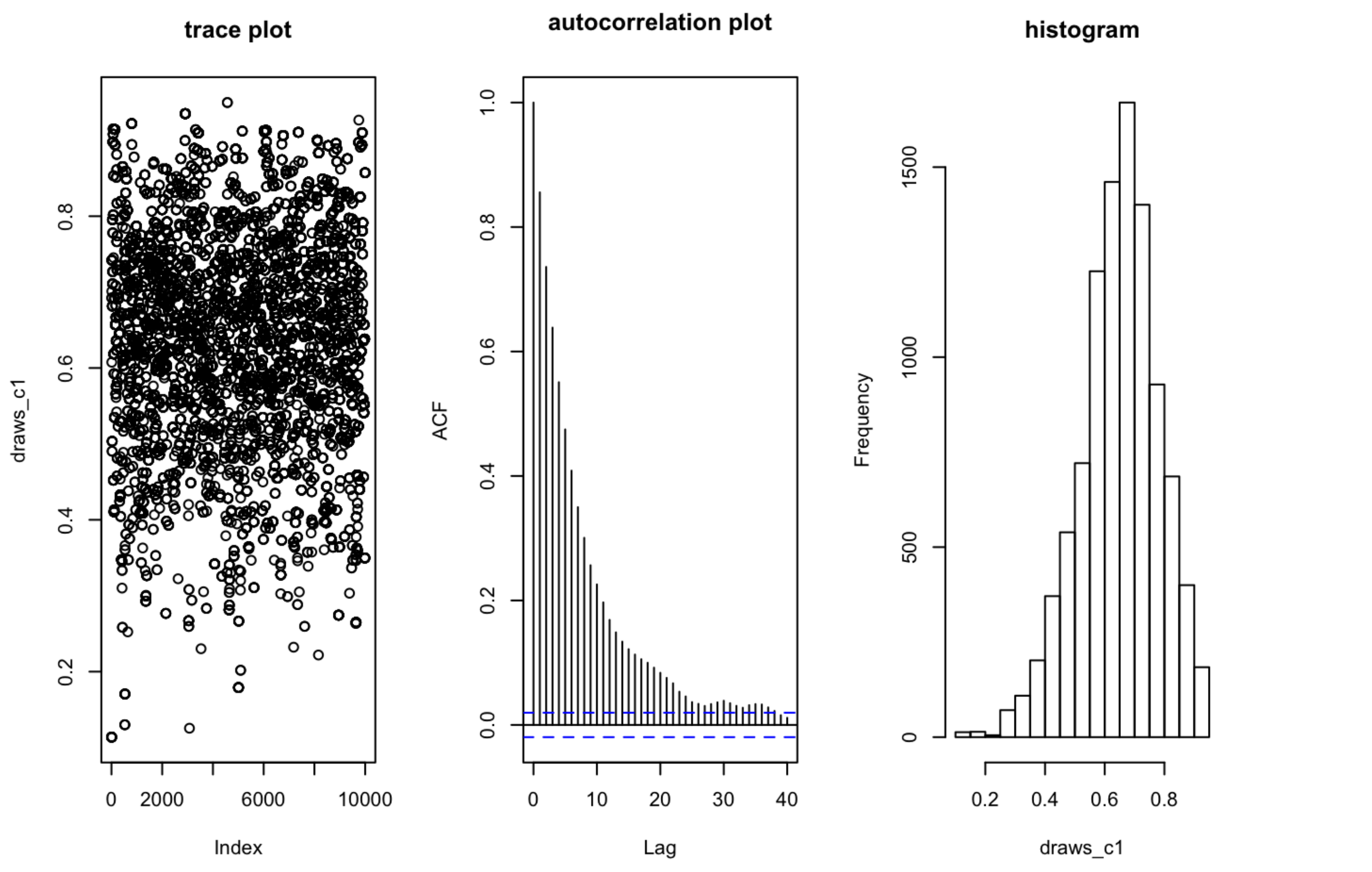
```
set.seed(3)
draws_c1 = run_metropolis_MCMC(startvalue=start, num_iteration)
acceptance_rate_c1 = 1-mean(duplicated(draws_c1))
acceptance_rate_c1
```

```
## [1] 0.1870813
```

Part 2

We provide a trace plot of this sampler and an autocorrelation plot, as well as a histogram of the draw to evaluate the performance of the sampler.

```
par(mfrow=c(1,3))  #1 row, 3 columns
plot(draws_c1, main="trace plot"); acf(draws_c1,main="autocorrelation plot"); hist(draws_c
1,main="histogram")
```

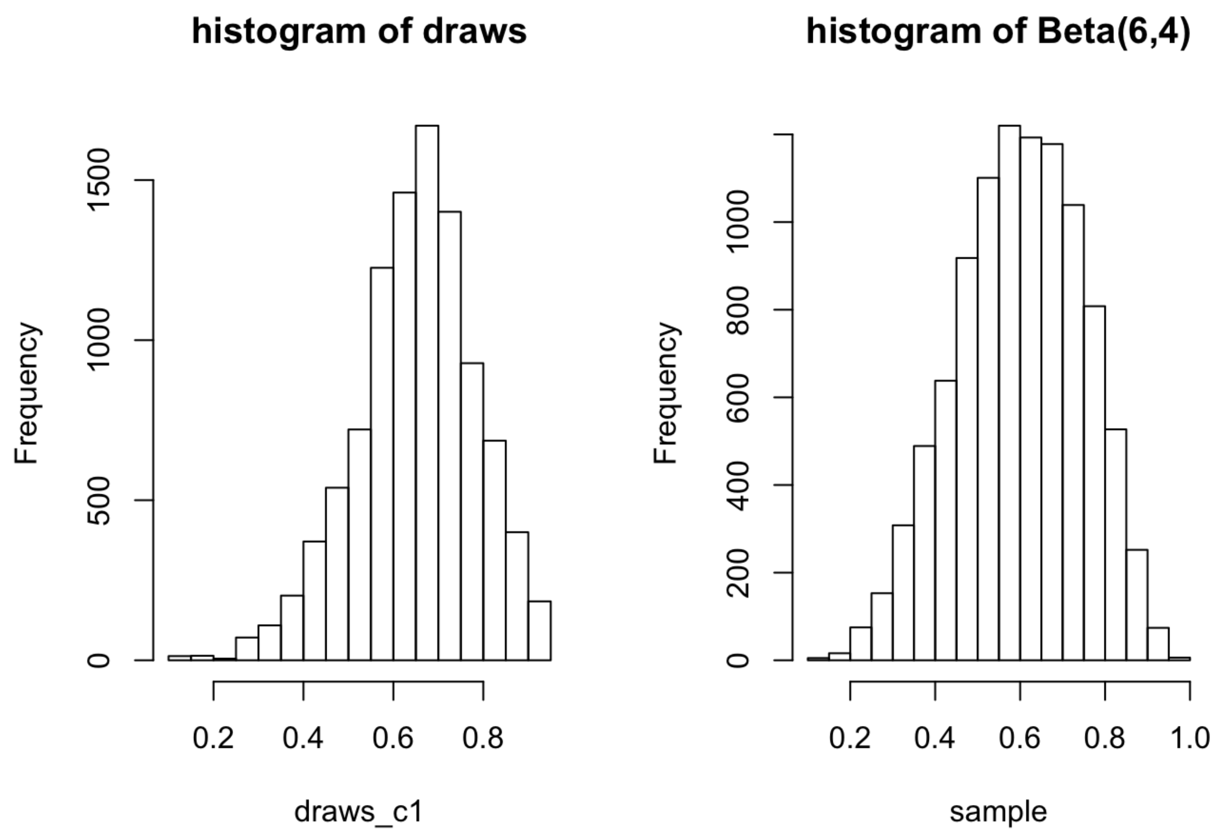


From the trace plot we can see, the sampler we generated are spread pattenlessly across index and the values are always between [0,1], however, the values are concentrated between [0.4,0.9]; from autocorrelation plot we see the autocorelation between consecutive samplers are high but decreases as the lag increases. This make sense as the next sample point we decided to add into the resulting vector is highly dependent on the previous sample, but as the simulation proceeds forward, the correlation between next samplers and further previous samplers are decreasing.

Compare the histogram of draws with target distribution of Beta(6,4).

- First, graphically compare these two.

```
par(mfrow=c(1,2))
hist(draws_c1, main = "histogram of draws",nclass = 15)
sample <- rbeta(num_iteration,6,4)
hist(sample, main = "histogram of Beta(6,4)",nclass = 15)
```



We found that our algorithm generate a sample whose histogram is very similar to the beta distribution histogram with the required shape parameters.

- Second, use Kolmogorov-Smirnov statistic to compare these two.

```
ks.test(draws_c1,"pbeta",6,4)
```

```
## Warning in ks.test(draws_c1, "pbeta", 6, 4): ties should not be present for
## the Kolmogorov-Smirnov test
```

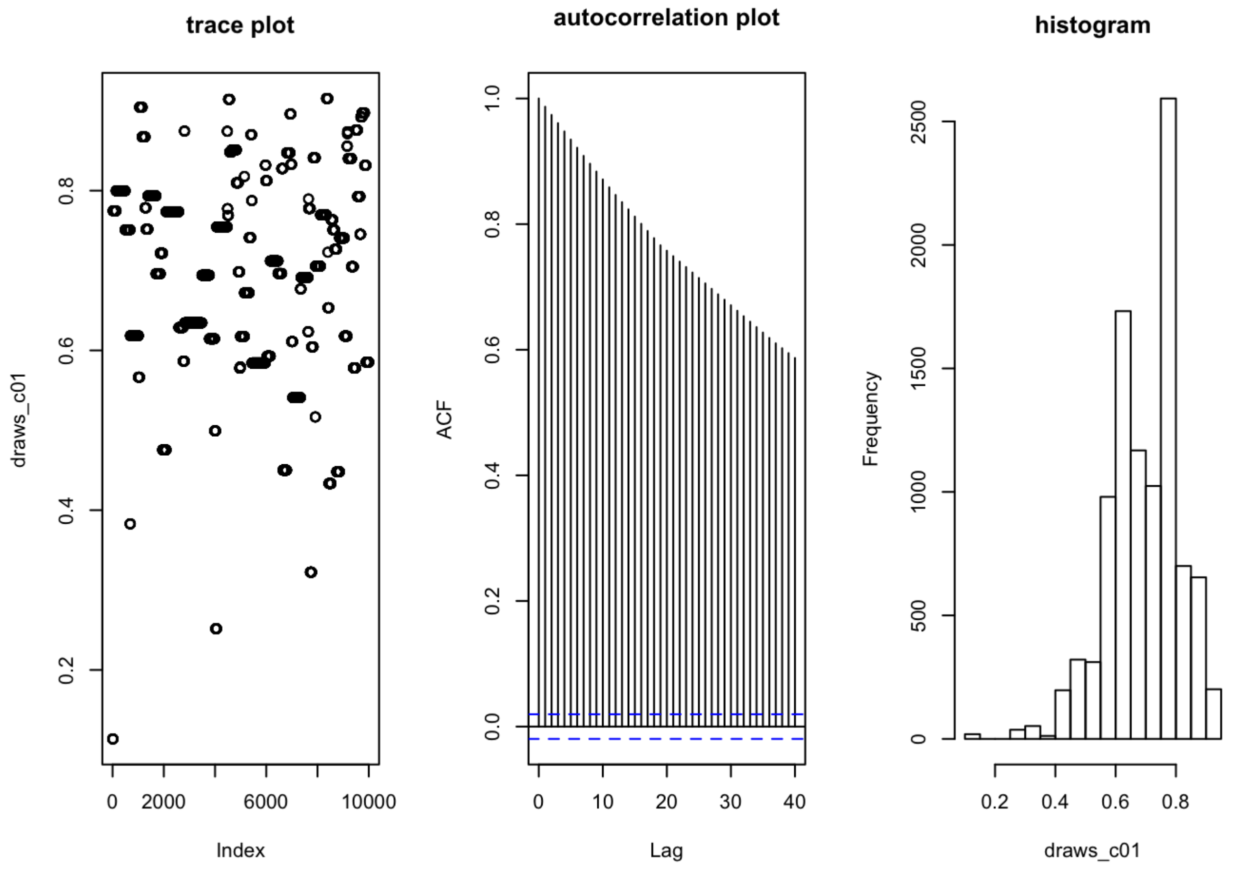
```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  draws_c1
## D = 0.16419, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

The Kolmogorov–Smirnov statistic measures a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution. Because the p-value is extremely small, we reject the null hypothesis that the sample is drawn from the reference (Beta(6,4)) distribution. Notice that we got the error message because we have replicated values in the sample we drawn.

Part 3

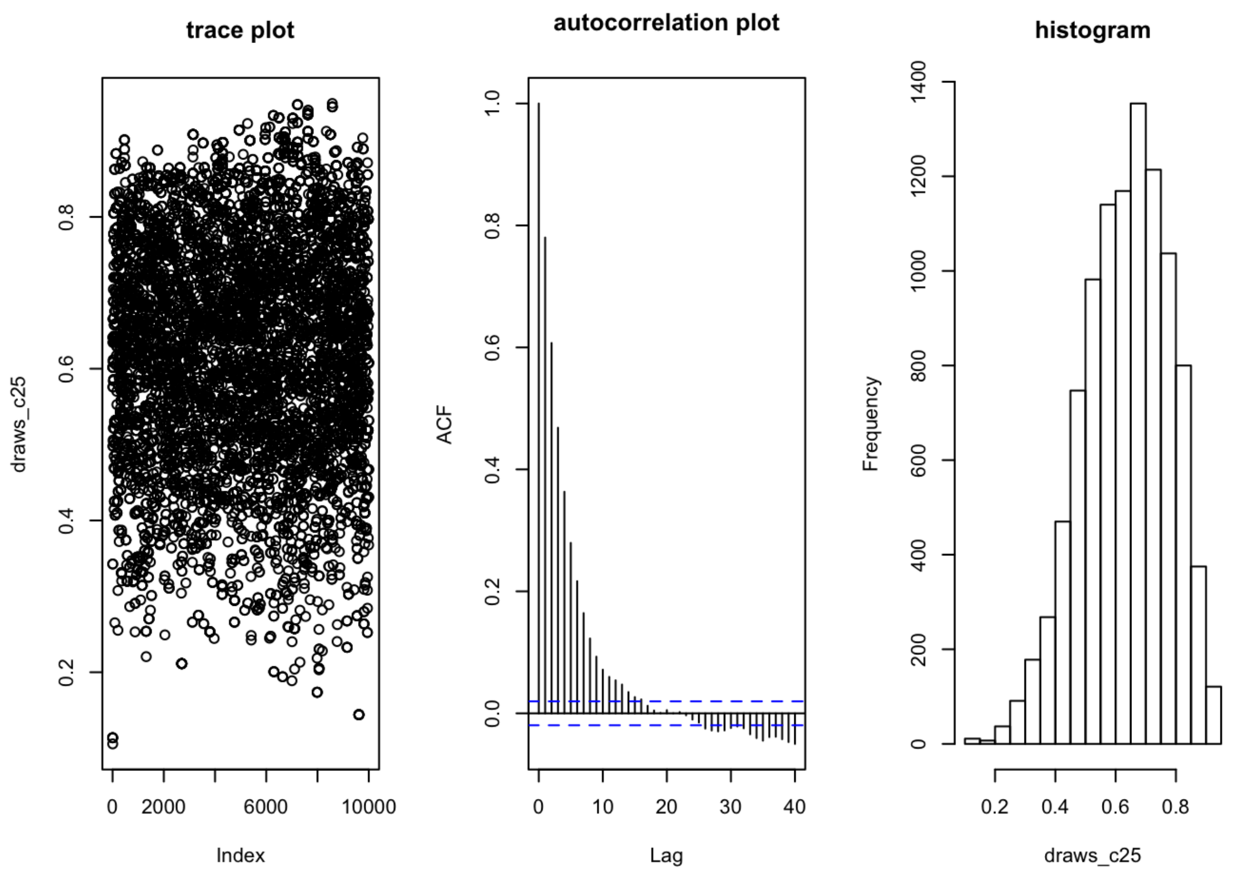
Re-run this sampler with c = 0.1, c = 2.5 and c = 10, and compare their results.

```
c=0.1
draws_c01 = run_metropolis_MCMC(startvalue=start, num_iteration)
par(mfrow=c(1,3))
plot(draws_c01,main="trace plot"); acf(draws_c01,main="autocorrelation plot"); hist(draws_
c01,main="histogram")
```



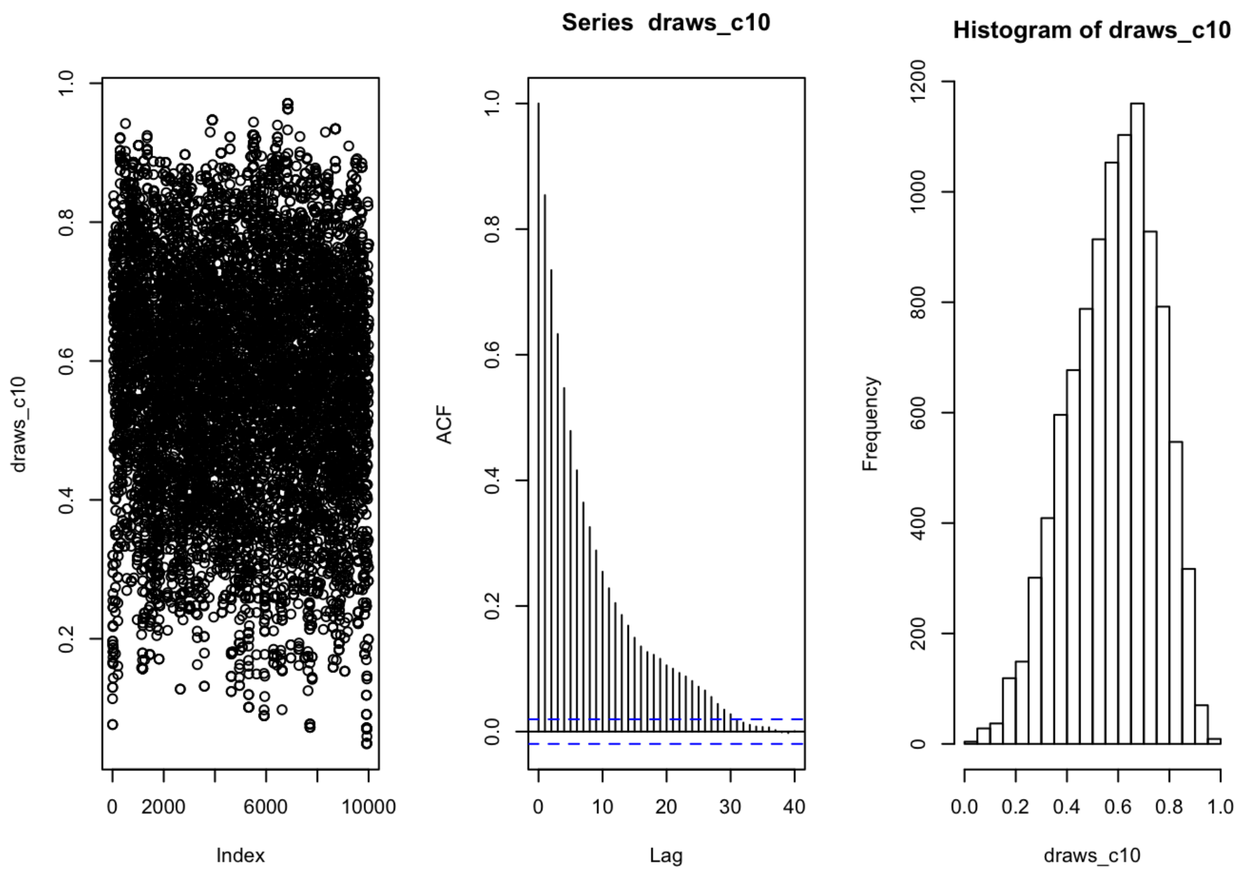
```
acceptance_rate_c01 = 1-mean(duplicated(draws_c01))
```

```
c=2.5
draws_c25 = run_metropolis_MCMC(startvalue=start, num_iteration)
par(mfrow=c(1,3))  #1 row, 3 columns
plot(draws_c25,main='trace plot'); acf(draws_c25,main='autocorrelation plot'); hist(draws_
c25,main='histogram')
```



```
acceptance_rate_c25 = 1-mean(duplicated(draws_c25))
```

```
c=10
draws_c10 = run_metropolis_MCMC(startvalue=start, num_iteration)
par(mfrow=c(1,3))  #1 row, 3 columns
plot(draws_c10); acf(draws_c10); hist(draws_c10)
```



```
acceptance_rate_c10 = 1-mean(duplicated(draws_c10))
```

Compare the acceptance rate for different c values,

```
paste0("acceptance rate when c=0.1: ",acceptance_rate_c01)
```

```
## [1] "acceptance rate when c=0.1: 0.00879912008799122"
```

```
paste0("acceptance rate when c=1: ",acceptance_rate_c1)
```

```
## [1] "acceptance rate when c=1: 0.187081291870813"
```

```
paste0("acceptance rate when c=2.5: ",acceptance_rate_c25)
```

```
## [1] "acceptance rate when c=2.5: 0.399260073992601"
```

```
paste0("acceptance rate when c=10: ",acceptance_rate_c10)
```

```
## [1] "acceptance rate when c=10: 0.650734926507349"
```

Comparing the acceptance rate of simulation for different values of c, we see the acceptance rate increases as c increases, meaning the algorithm are more efficient when c has larger value. From the trace plots we can also see, when $c = 10$, we generate most data points for the target distribution, with same number of iterations.

Compare the histogram plots, the histograms tend to be more consistent with the target beta(6,4) distribution as we increase the value of c.

Q2. Gibbs Sampling

Step 1: Use Inverse Transform Sampling to generate the samples from the conditional distribution

From $P(x|y) \propto ye^{-yx}, 0 < x < B < \infty$, we have $\int_0^B Kye^{-yx} = 1$, so $K = \frac{1}{1-e^{-yB}}$

The CDF of this conditional distribution is $F(x|y) = \frac{1}{1-e^{-yB}}(1 - e^{-yx})$

Further, the inverse of this CDF is $F^{-1}_{(x|y)}(u) = -\frac{ln[1-u(1-e^{-yB})]}{y}$

Similarly, we have $F^{-1}_{(y|x)}(u) = -\frac{ln[1-u(1-e^{-xB})]}{x}$

These inverse CDF's will be used to generate joint sample from the given conditional distribution.

Step 2: We will then use Gibbs sampler to estimate the marginal distribution, and store results in a matrix.

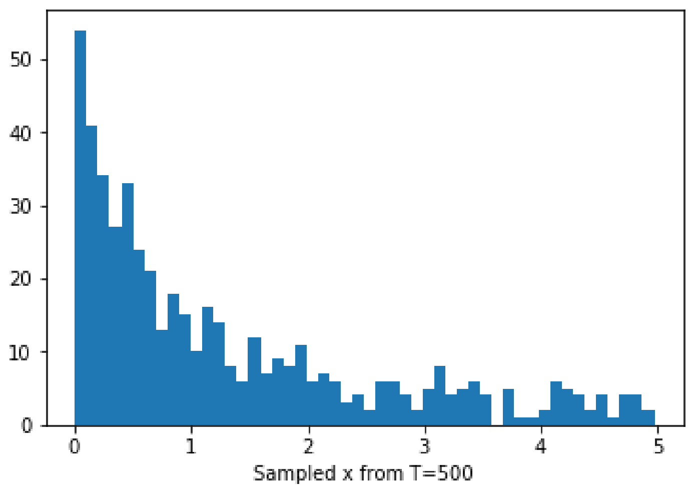
In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

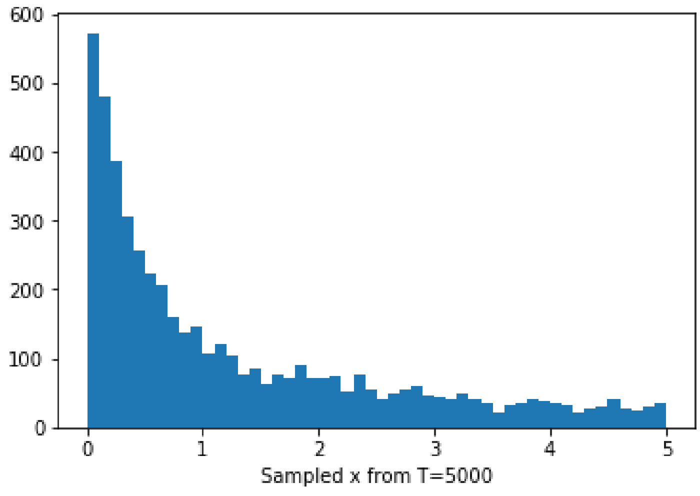
In [2]:
```python
## First define a functino that will give us the inverse CDF of the given
 conditional distribution.
def inverse_sampler(B,theta):
    u = np.random.random()
    norm_constant = 1-np.exp(-B*theta)
    x = -np.log(1-u*norm_constant)/theta
    return(x)

## Second, define a funtion that will return updated value of x and y base
d on the given conditional distribution.
## The sequence we get can be used to approximate the joint distribution.
def Gibbs_sampler(B,sample_size, start_value=1):
    sample_chain = np.zeros(sample_size*2).reshape(2,sample_size)  # defin
e a matrix to store results
    start_y = start_value   # give a start value of y
    for i in range(sample_size):
        sample_chain[0,i] = inverse_sampler(B,start_y)   # update x value
 and store it in the first row of the matrix
        sample_chain[1,i] = inverse_sampler(B,sample_chain[0,i]) # update
 y value and store it in the second row of the matrix
        start_y = sample_chain[1,i]
    return(sample_chain)
```

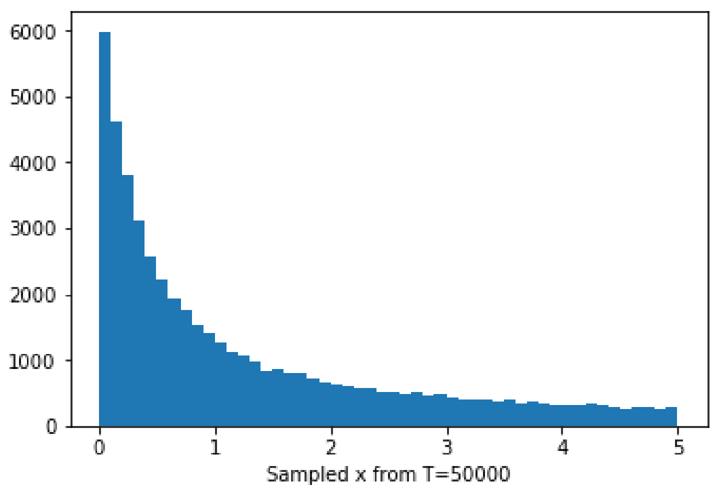
Therefore, the `sample_chain` matrix will give us the approximate joint distribution of x and y based on the given conditional distribution.

Step 3: For B=5, and for sample sizes T = 500,5000,50000, plot the histogram of values for x.

In [3]:
```
chain1 = Gibbs_sampler(5,500)
chain2 = Gibbs_sampler(5,5000)
chain3 = Gibbs_sampler(5,50000)

plt.hist(chain1[0,:],bins=50)
plt.xlabel("Sampled x from T=500")
plt.show()

plt.hist(chain2[0,:],bins=50)
plt.xlabel("Sampled x from T=5000")
plt.show()

plt.hist(chain3[0,:],bins=50)
plt.xlabel("Sampled x from T=50000")
plt.show()
```







As these three histograms show, with more numbers of iterations and larger sample size, the marginal distribution of X becomes more closer to exponetial distribution.

Step 4: Provide an estimate of the expectation of X

Based on our simulated sample, it is straightforward to calculate the expectation of X from its marginal distribution.

```
In [4]: print ("E(x) estimated from 500 sampler = %s" % (np.mean(chain1[0,:])))
        print ("E(x) estimated from 5000 sampler = %s" % (np.mean(chain2[0,:])))
        print ("E(x) estimated from 50000 sampler = %s" % (np.mean(chain3[0,:])))
```

```
E(x) estimated from 500 sampler = 1.29550437883
E(x) estimated from 5000 sampler = 1.28369853795
E(x) estimated from 50000 sampler = 1.2596060758
```

Based on the samples of X from our Gibbs sampler, an estimate of the expectation of X that we can give is about 1.26.

Question 3

(1) Algorithm for K-means clustering

To implement K-Means algorithm, we should do following steps.

- Randomly choose K points (which is 3 here) as the initial centers.
- For each observation, determine which cluster center is the cloest to it using Euclidean distance measure. Then, assign that observation to the corresponding cluster.
- Compute the mean location of the points in each cluster. Set this mean location as the new cluster center point.
- Repeat step 2 and 3, until there is no more change for clusters.

We now write a K-means sampler which takes data for clustering(initial type not included) and number of clusters as parameters and returns a vector showing which cluster each point belongs to.

```r
k_means_cluster <- function(cluster_data,num_cluster){
  cluster_data <- as.matrix(cluster_data)
  num_points <- nrow(cluster_data)
  num_dim <- ncol(cluster_data)
  #sample from dataset to get initial mean of three clusters
  start_mean <- cluster_data[sample(1:num_points,num_cluster,replace = F),]
  new_cluster_label <- numeric(num_points)
  old_cluster_label <- rep(1,num_points)
  #stop iterating when classification does not change
  while (any(old_cluster_label != new_cluster_label)){
    old_cluster_label <- new_cluster_label
    distance <- matrix(0,num_points,num_cluster)
    #calculate each data points' Euclidean distance to each old mean of clusters
    for (j in 1:num_cluster){
      distance[,j] <- apply((cluster_data - matrix(start_mean[j,],nrow = num_points,
                                                    ncol = num_dim,byrow = TRUE))^2,1,sum)
    }
    #find shortest distance and assign each data point to a new cluster
    new_cluster_label <- apply(distance,1,which.min)
    #calculate new mean of new clusters
    for (j in 1:num_cluster){
      start_mean[j,] <- apply(cluster_data[which(new_cluster_label==j),],2,mean)
    }
  }
  #return a vector indicating each data point's cluster
  return(new_cluster_label)
}
```

Now we use this classifier to cluster the wines into 3 groups.

```r
library("rattle")
```

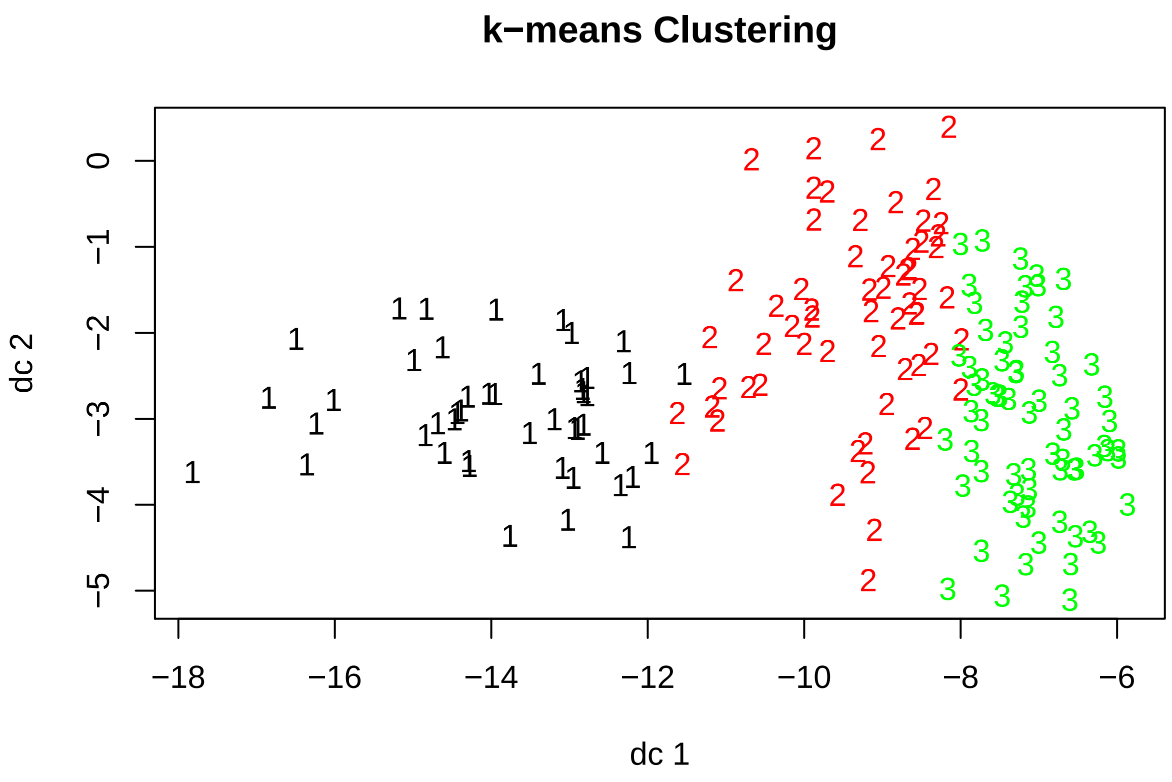
```
## Rattle: A free graphical interface for data science with R.
## Version 5.2.0 Copyright (c) 2006-2018 Togaware Pty Ltd.
## Type 'rattle()' to shake, rattle, and roll your data.
```

```r
data(wine)
set.seed((3.1))
```

```
num_cluster <- 3
newdata <- wine[,-1] #exclude original Type
label <- k_means_cluster(newdata,num_cluster) #result of k-means
```

The result of k-means clustering is shown below.

```
library(fpc)
plotcluster(newdata,label,main="k-means Clustering")
```



The data points seem well seperated by the classifier although same boundary points are close.

(2) Quantify how well the result corresponds to original types

Now we want to quatify how well the result correspond to the original data. To quantify this, we first need to find out which original types do these new clusters correspond to. To match them, we first calculate the proportion of three original types in each new cluster and match each new cluster with the type whose propotion is largest in this cluster. In case that two new clusters may match the same original type, we match the new cluster of largest number in proportion matrix first. Then match the cluster of largest number in reduced 2*2 proportion matrix. And the remaining cluster matches the remaining type. Hence the match between old types and new clusters are built.

```
#the function of calculating the proportion matrix
matching_matrix <- function(old_type,new_type,num_cluster){
  proportion <- matrix(0,num_cluster,num_cluster)
  for(i in 1:num_cluster){
    index <- (new_type==i)
    for (j in 1:num_cluster){
      proportion[i,j] <- sum(old_type[index]==j)/sum(index)
    }
  }
  return(proportion)
}
#proportion matrix for wine[,-1]
matching_matrix(wine$Type,label,num_cluster)
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.9787234 0.0212766 0.0000000
## [2,] 0.2096774 0.3225806 0.4677419
## [3,] 0.0000000 0.7246377 0.2753623
```

From the proportion matrix, we can see that new cluster 1 matches type 1, new cluster 2 matches type 3, new cluster 3 matches type 2.

```
#decide actual type of each new cluster matches according to proportion matrix
actual <- c(1,3,2)
#the function assigns actual label to points
new_cluster <- function(new_type,actual_type){
  new_index <- numeric(length(new_type))
   for (i in 1:length(actual_type)){
     new_index[(new_type==i)] <- actual_type[i]
   }
   return(new_index)
}
#relabel data points to match
new_label <- new_cluster(label,actual)
```

So we can further quantify how well the result fits original types using the number of points whose type(label) does not change before and after k-means clustering.

```
#The function gives the propotion whose label does not change
right_rate <- function(old_type,new_cluster){
 return(sum(old_type==new_cluster)/length(old_type))
}
right_rate(wine$Type,new_label)
```

```
## [1] 0.7022472
```

So the right rate of the classifier for wine[,-1] is 0.7022472. We think this result is well seperated.

(3) The effect of scaling

```
#scale the data
newdata1 <- scale(wine[,-1])
apply(newdata1,2,mean)
```

```
##         Alcohol         Malic           Ash    Alcalinity
##   -8.591766e-16 -6.776446e-17  8.045176e-16 -7.720494e-17
##       Magnesium       Phenols    Flavanoids Nonflavanoids
##   -4.073935e-17 -1.395560e-17  6.958263e-17 -1.042186e-16
## Proanthocyanins         Color           Hue      Dilution
##   -1.221369e-16  3.649376e-17  2.093741e-16  3.003459e-16
##         Proline
##   -1.034429e-16
```

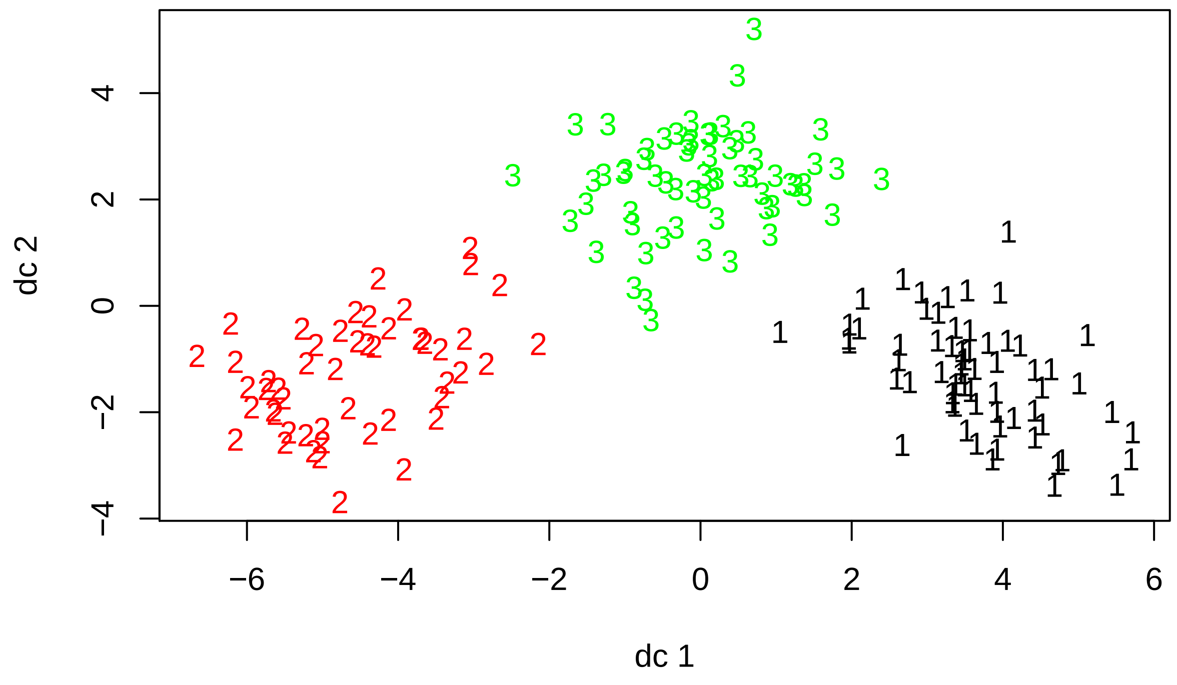
```
apply(newdata1,2,sd)
```

```
##         Alcohol           Malic             Ash      Alcalinity 
##               1               1               1               1 
##       Magnesium         Phenols      Flavanoids  Nonflavanoids 
##               1               1               1               1 
## Proanthocyanins           Color             Hue        Dilution 
##               1               1               1               1 
##         Proline 
##               1
```

The function scale() transform all data points into a family of points with mean 0 and standard error 1.

Now we repeat the calculation process for scaled data.

```
#cluster the scaled data
set.seed(3.2)
label1 <- k_means_cluster(newdata1,num_cluster)
plotcluster(newdata1,label1)
```



From the plot, we can see that these data points are well seperated. The distances between clusters seem larger than those in Part 1. Now we show this in calculation.

```
matching_matrix(wine$Type,label1,num_cluster)
```

```
##          [,1]       [,2]      [,3]
## [1,] 0.921875 0.07812500 0.0000000
## [2,] 0.000000 0.05882353 0.9411765
## [3,] 0.000000 1.00000000 0.0000000
```

So new cluster 1 matches type 1, new cluster 2 matches type 3, new cluster 3 matches type 2.

```
#match clusters with old corresponding types
actual <- c(1,3,2)
new_label1 <- new_cluster(label1,actual)
right_rate(wine$Type,new_label1)
```

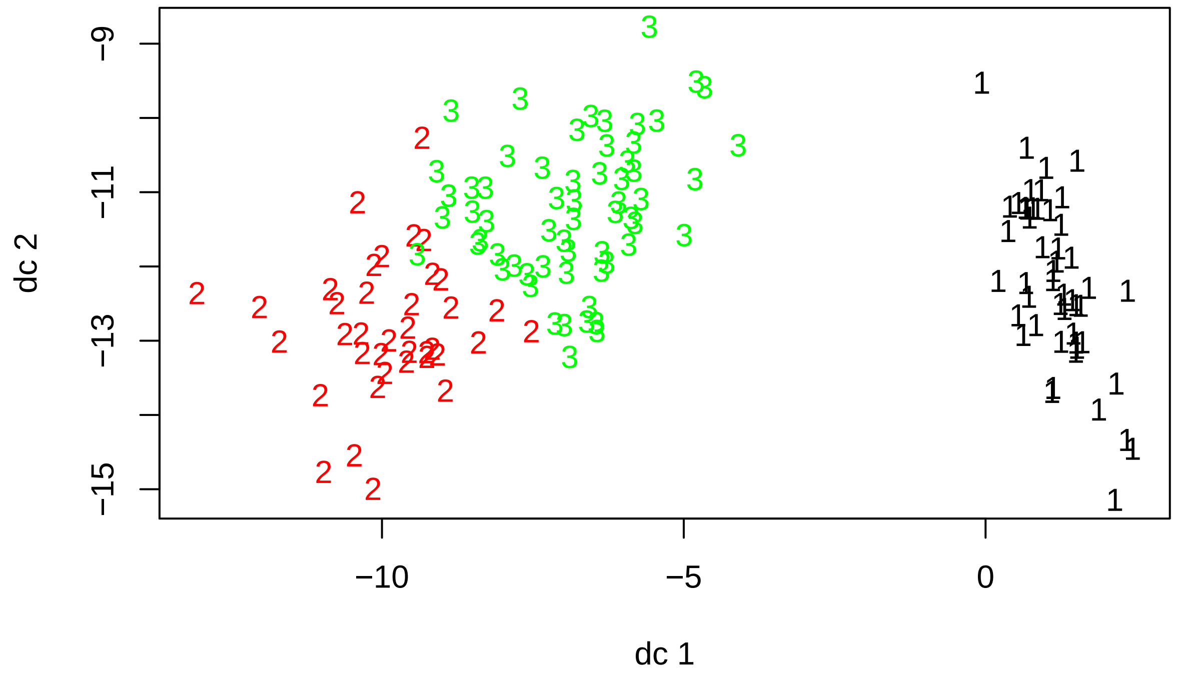
```
## [1] 0.9550562
```

So the right rate of the classifier for scaled data wine[,-1] is 0.9550562 which is obviously larger than the right rate for original data.

Therefore, in this case, using scaled data can help us improve the performance of k-means clustering, making more data points' new cluster matches original type.

(4) Application to iris dataset

```
#k-means clustering for iris data
data("iris")
levels(iris$Species) <- c(1,2,3)
newdata <- iris[,-5]
set.seed(3.3)
label <- k_means_cluster(newdata,num_cluster)
plotcluster(newdata,label)
```



The data points of new cluster 1 seem well seperated by the classifier. Other points seem not as well seperated as cluster 1 but are good in general although the boudary of new cluster 2 and 3 is not very obvious.

```
matching_matrix(iris$Species,label,num_cluster)
```

```
##      [,1]       [,2]      [,3]
## [1,]    1 0.00000000 0.0000000
## [2,]    0 0.05263158 0.9473684
## [3,]    0 0.77419355 0.2258065
```

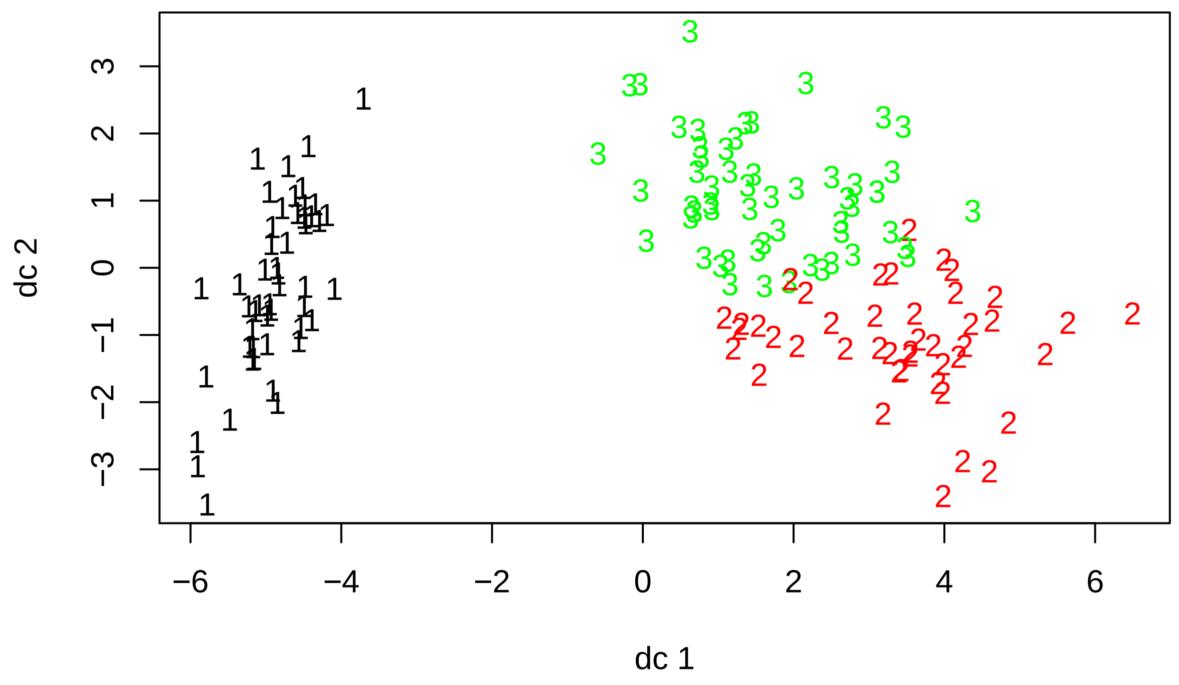
So new cluster 1 matches type 1, new cluster 2 matches type 3, new cluster 3 matches type 2.

```
actual <- c(1,3,2)
new_label <- new_cluster(label,actual)
right_rate(iris$Species,new_label)
```

```
## [1] 0.8933333
```

So the right rate of the classifier for iris[,-5] is 0.8933333, which indicates that the classifier works well for this dataset.

```
set.seed(3.4)
newdata1 <- scale(iris[,-5])
label1 <- k_means_cluster(newdata1,num_cluster)
plotcluster(newdata1,label1)
```



It can be seen from this plot that these data points are well seperated in general but the small problem in results from original iris data still exists. So we might guess the effect of scaling may not be very obvious in this case or even worse.

```
matching_matrix(iris$Species,label1,num_cluster)
```

```
##      [,1]      [,2]      [,3]
## [1,]    1 0.0000000 0.0000000
## [2,]    0 0.2500000 0.7500000
## [3,]    0 0.6964286 0.3035714
```

So new cluster 1 matches type 1, new cluster 2 matches type 3, new cluster 3 matches type 2.

```
actual <- c(1,3,2)
new_label1 <- new_cluster(label1,actual)
right_rate(iris$Species,new_label1)
```

```
## [1] 0.8133333
```

We observe that the right rate declines a little bit for scaled data. So scaling can not improve the accuracy of k-means clustering in this case. We may conclude that whether scaling can help to improve k-means clutering's performance depends on the structure of the dataset.