Final Pair Exercises - R

Tianheng Huang & Hsiang Wang

11/2/2020

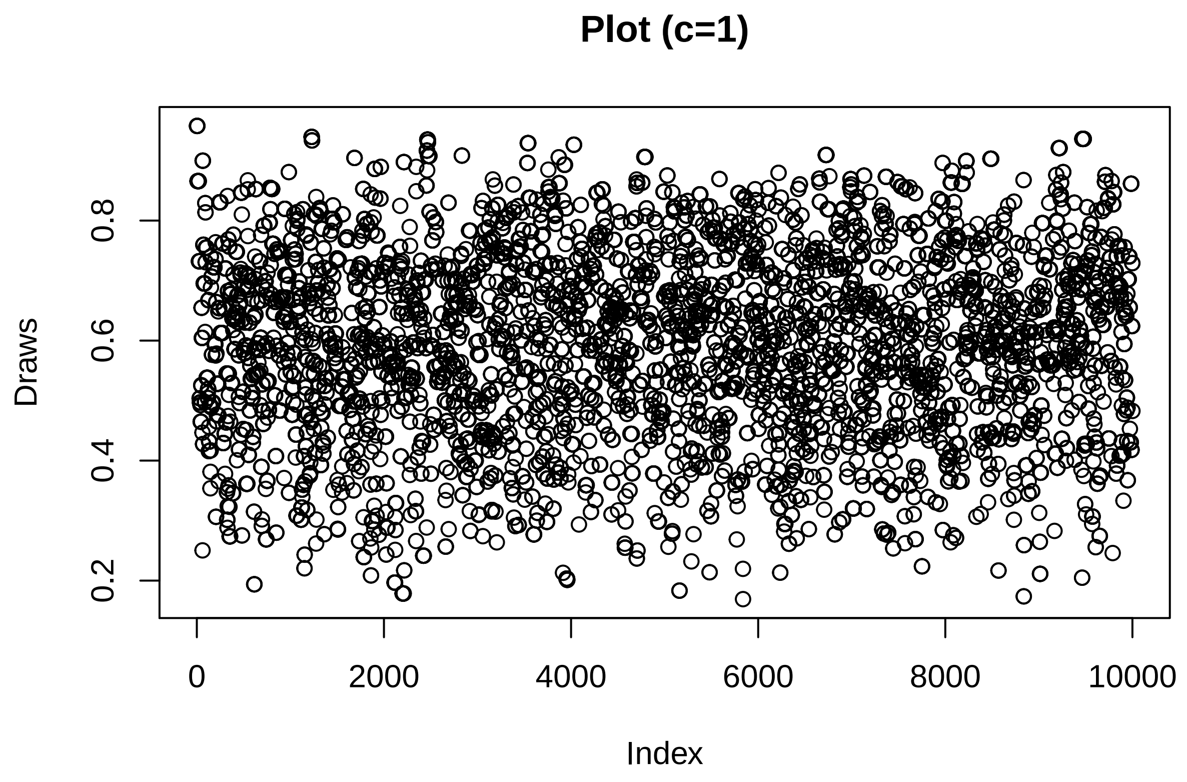
Metropolis-Hastings:

1. First, our function mh.beta takes the arguments n and c, which are respectively the number of iterations and the parameter controlling the shapes in the Beta proposal function. Then, it generates a starting value for the sampler from the uniform distribution between 0 and 1 and creates an empty vector to store the draws. Afterwards, it defines a subfunction phi.update taking phi, the current value of the sampler, and c as arguments. The subfunction creates a purposed value for the sampler based on the Beta distribution $\phi_{prop}|\phi_{old} = Beta(c\phi_{old}, c(1-\phi_{old})$, calculates the acceptance probability using the equation r=$\frac{p_{Beta}(\phi_{prop}|\alpha=6,\beta=4)/p_{Beta}(\phi_{prop}|c\phi_{old},c(1-\phi_{old}))}{p_{Beta}(\phi_{old}|\alpha=6,\beta=4)/p_{Beta}(\phi_{old}|c\phi_{prop},c(1-\phi_{prop}))}$, generates a random number from the uniform distribution between 0 and 1, and accepts the proposed value only if the random number is smaller or equal to the acceptance probability. Finally, the function just keeps making draws using the subfunction and stores the draws in the empty vector, ending up returning the vector.

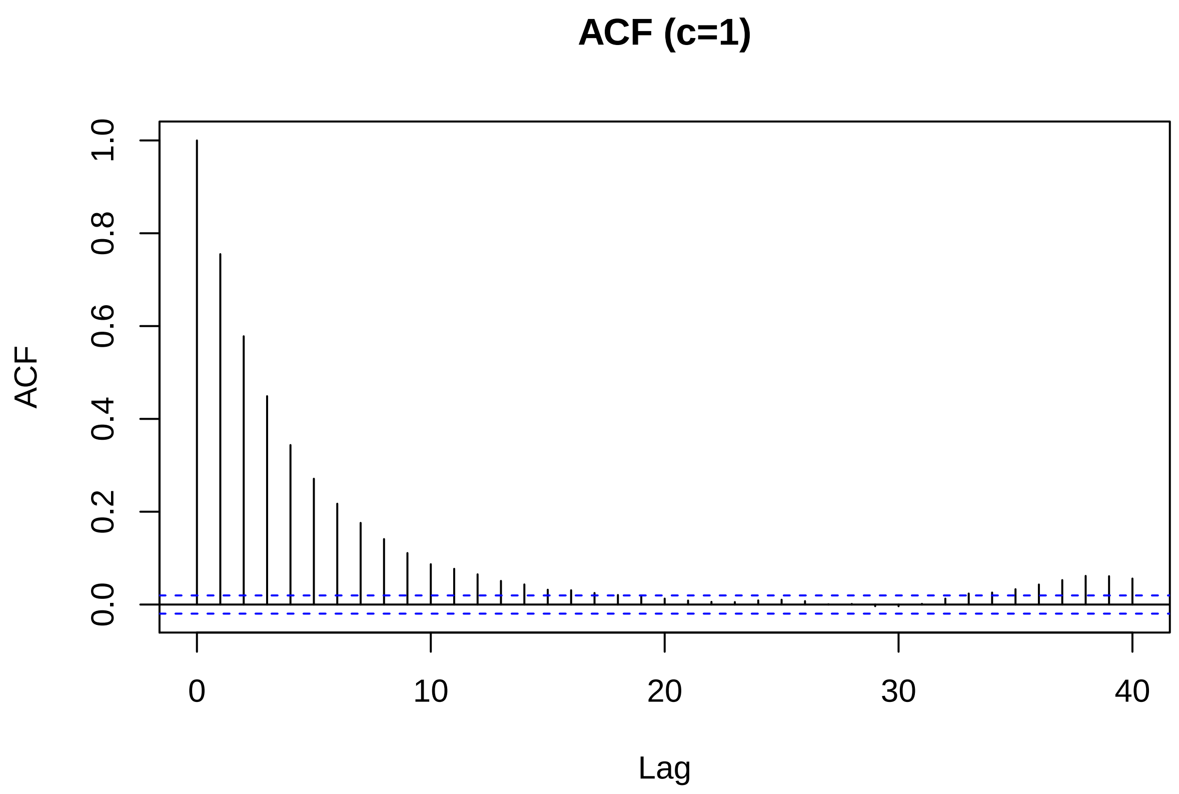
```r
mh.beta <- function(n, c) {
  phi=runif(1,0,1)
  draws <- c()
  phi.update <- function(phi, c) {
    phi.p <- rbeta(1,c*phi,c*(1-phi))
    r <- dbeta(phi.p, 6,4)/dbeta(phi,6,4)/dbeta(phi.p,c*phi,c*(1-phi))*
      dbeta(phi,c*phi.p,c*(1-phi.p))
    if (runif(1) <= r){
      phi.p
    }else{
      phi
    }
  }
  for (i in 1:n) {
    draws[i] <- phi <- phi.update(phi, c)
  }
  return(draws)
}
```

2. We have made the required plots below.

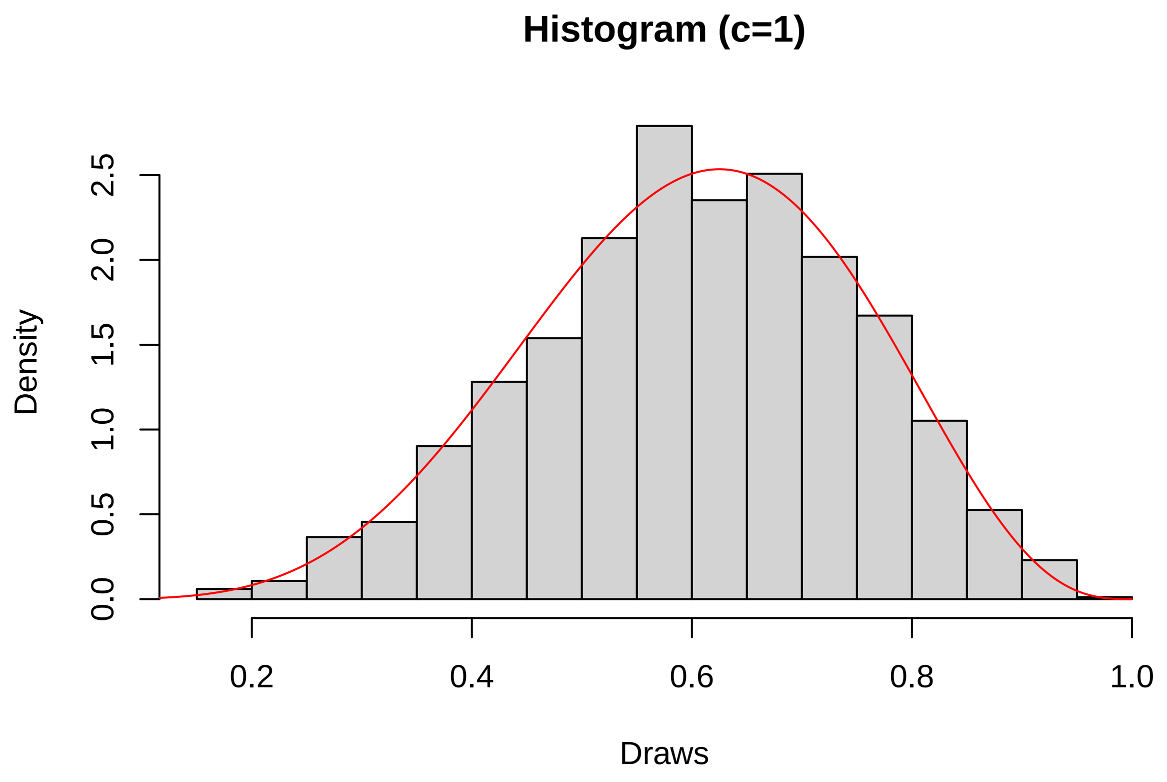
```r
d1=mh.beta(10000,1)
plot(d1,main="Plot (c=1)",ylab="Draws")
```



```r
acf(d1,main="ACF (c=1)")
```



```r
x<-seq(0,1,length=10000)
hist(d1,main="Histogram (c=1)",xlab="Draws",freq=F)
lines(x,dbeta(x,6,4),col="red")
```



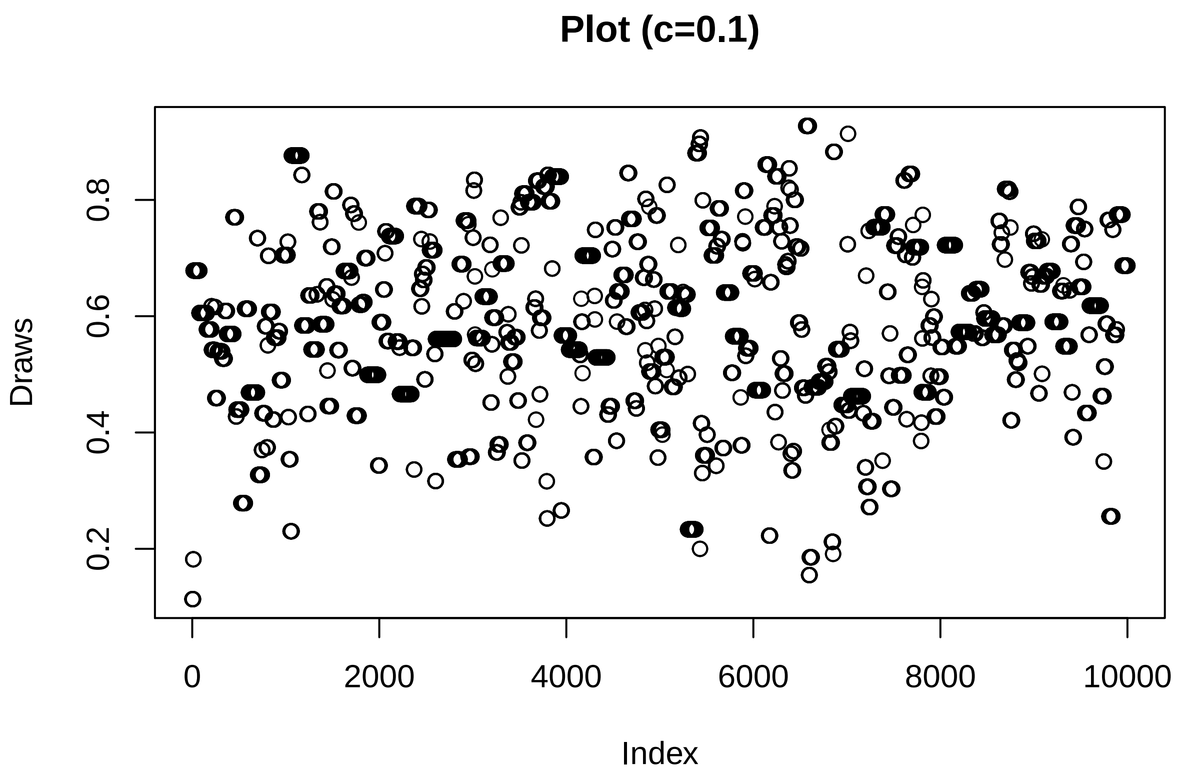
```r
ks1<-ks.test(d1,"pbeta",6,4)
```

```
## Warning in ks.test(d1, "pbeta", 6, 4): ties should not be present for the
## Kolmogorov-Smirnov test
```

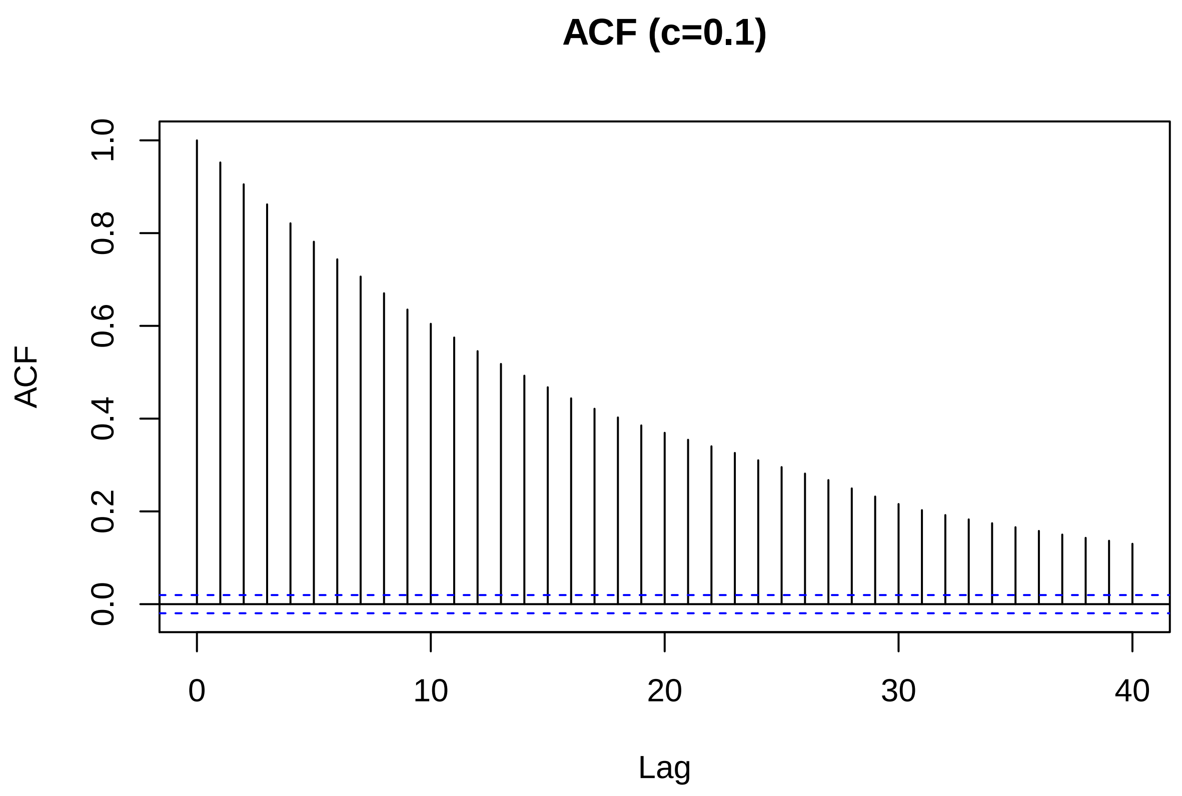
The histogram roughly resembles the target distribution in terms of the left-skewness. The KS-statistic is 0.0288185 with a p-value of 1.2227675×10^{-7}, suggesting that at a significance level of 5% we shall drop the null hypothesis that our MH draws come from the target distribution.

3. Here are the plots for c=0.1.

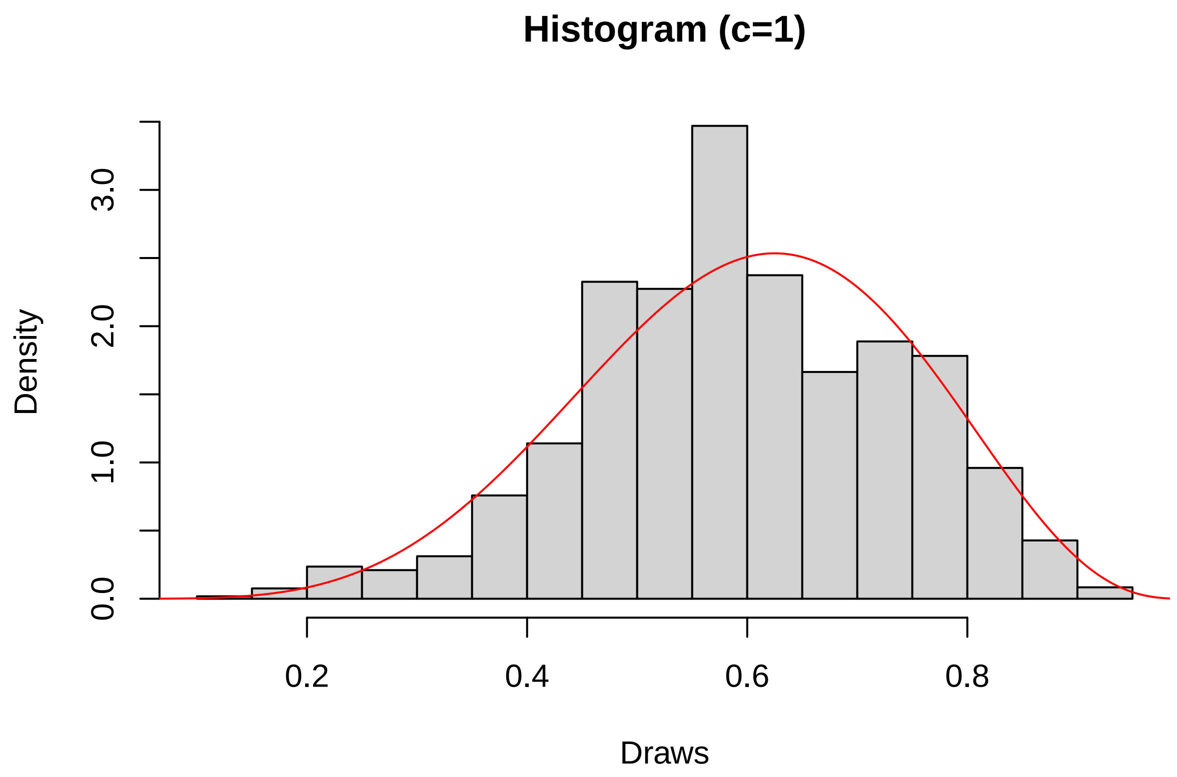
```r
d2=mh.beta(10000,0.1)
plot(d2,main="Plot (c=0.1)",ylab="Draws")
```



```
acf(d2,main="ACF (c=0.1)")
```

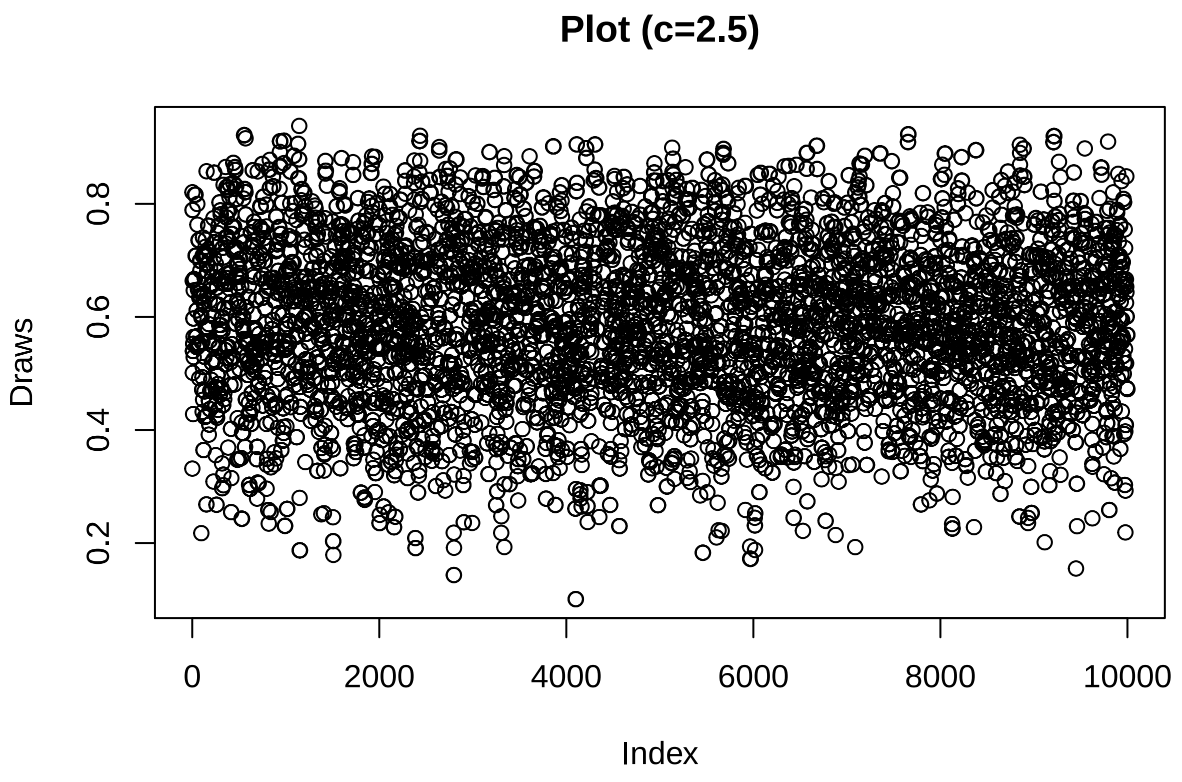


```r
hist(d2,main="Histogram (c=1)",xlab="Draws",freq=F)
lines(x,dbeta(x,6,4),col="red")
```

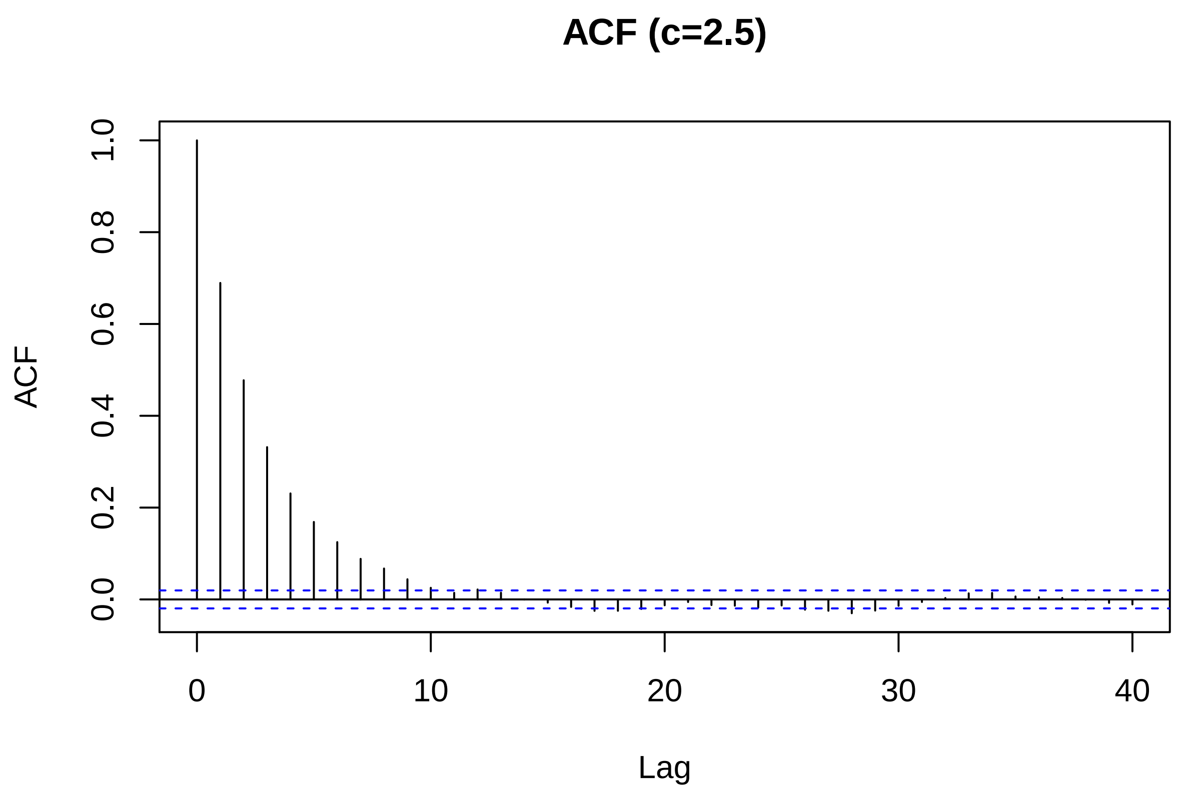


Here are the plots for c=2.5.

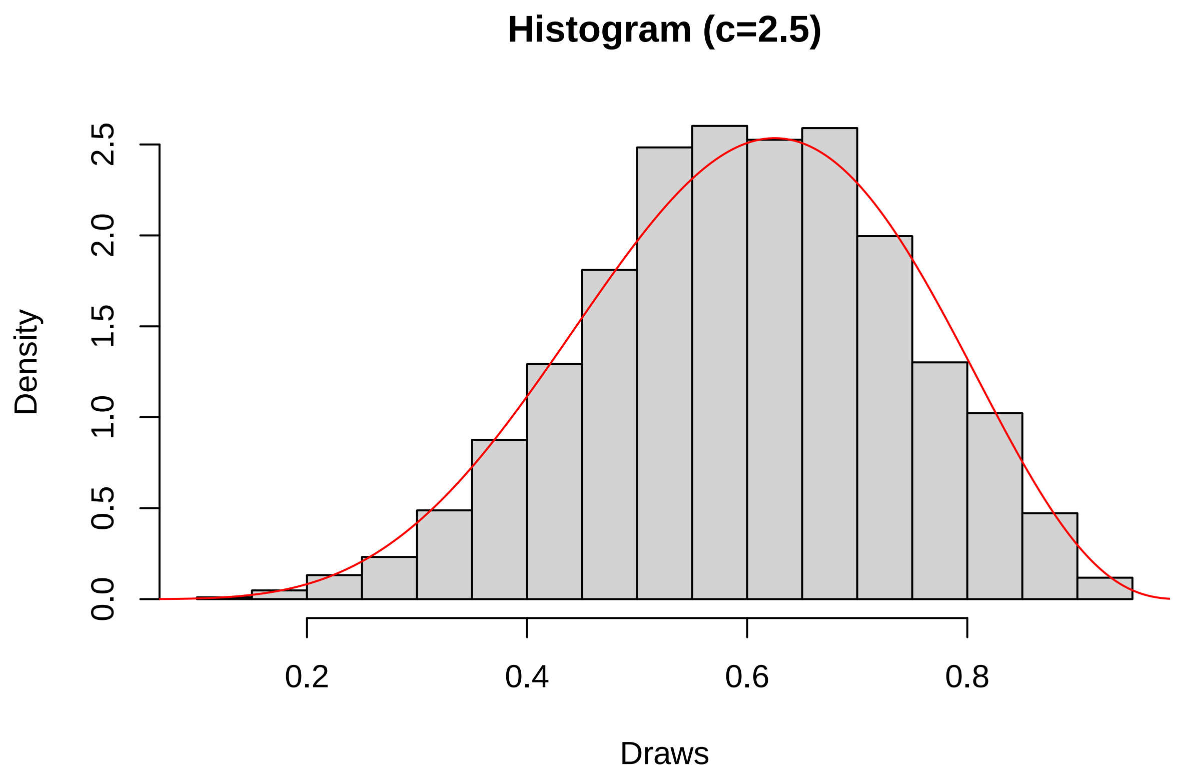
```
d3=mh.beta(10000,2.5)
plot(d3,main="Plot (c=2.5)",ylab="Draws")
```



```r
acf(d3,main="ACF (c=2.5)")
```

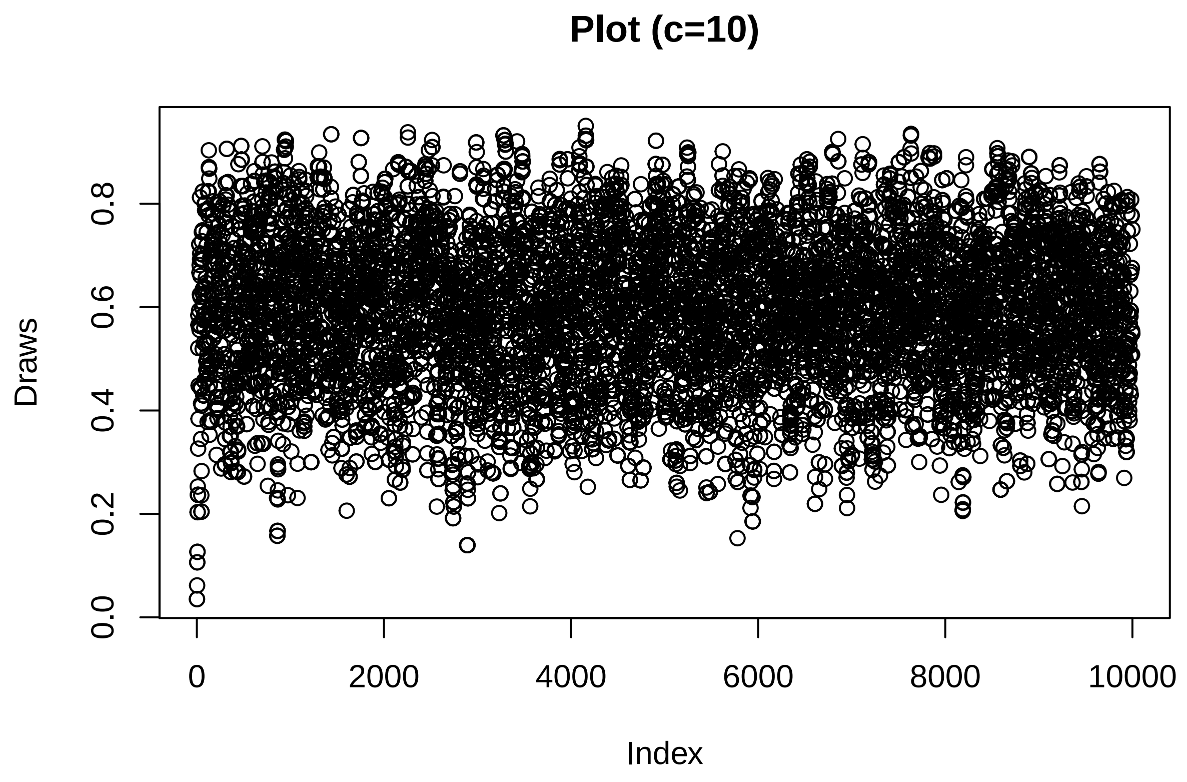


```r
hist(d3,main="Histogram (c=2.5)",xlab="Draws",freq=F)
lines(x,dbeta(x,6,4),col="red")
```

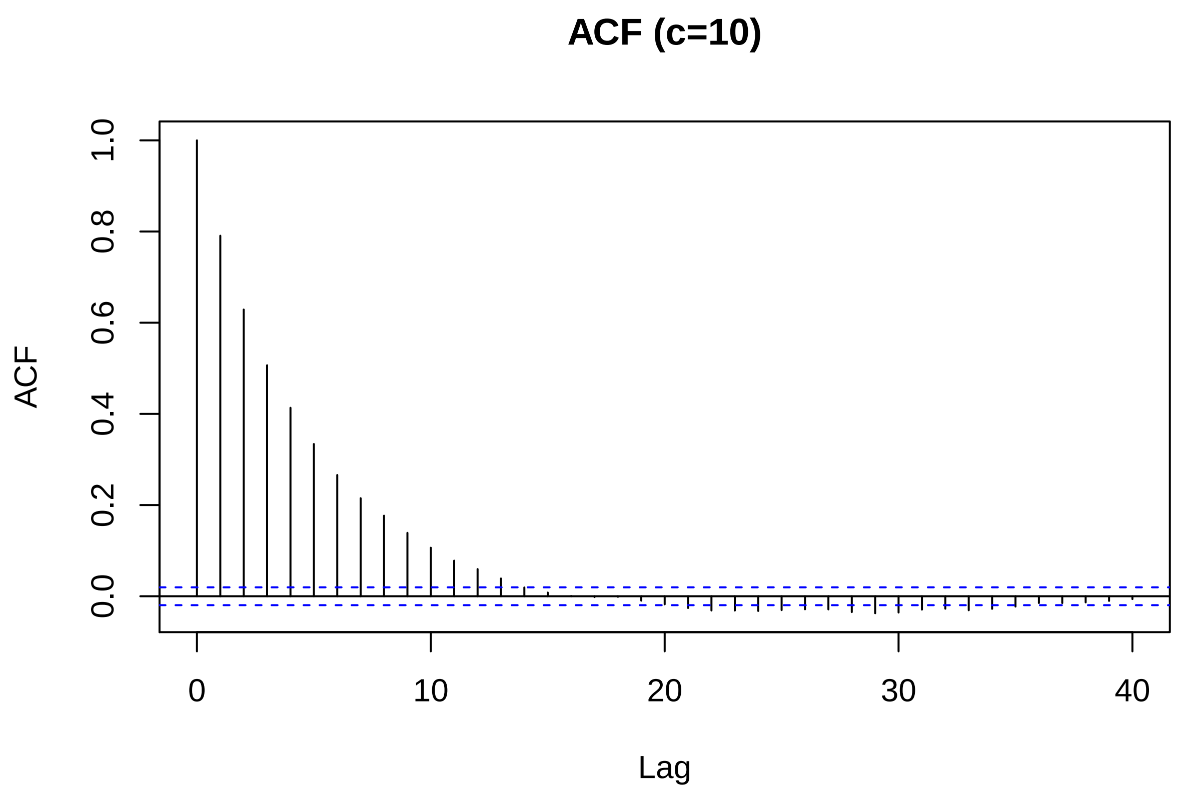


Here are the plots for c=10.

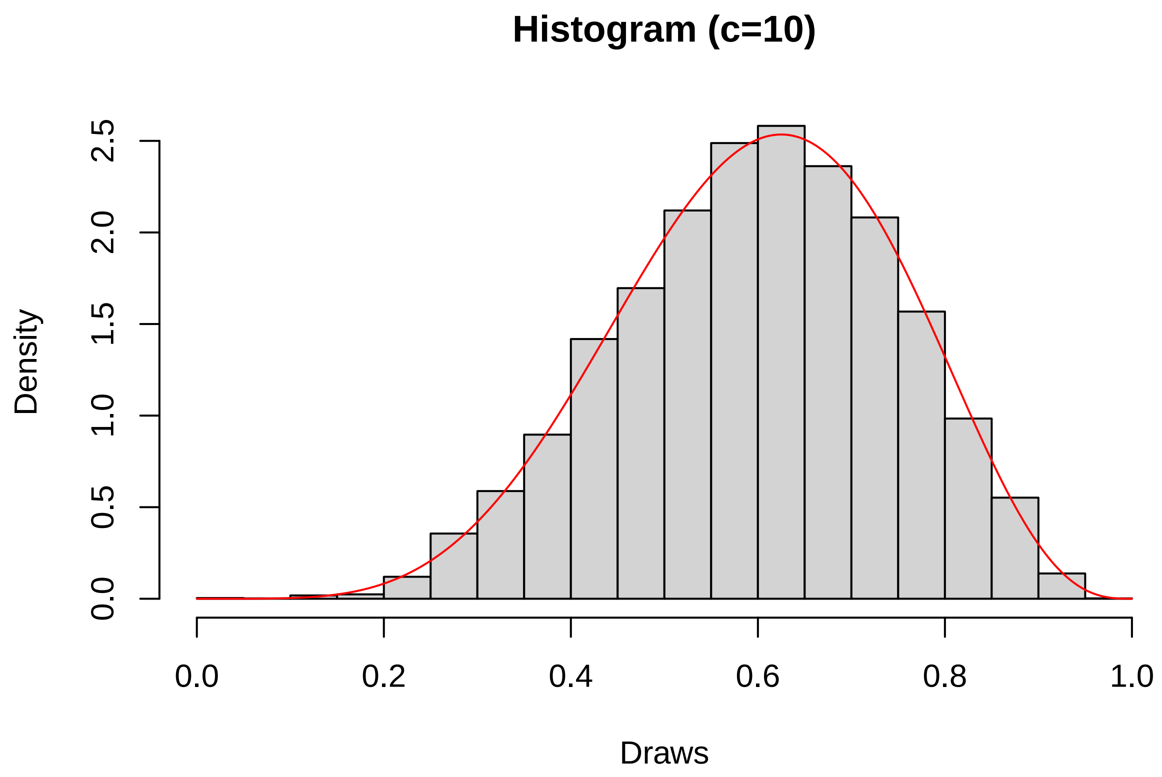
```
d4=mh.beta(10000,10)
plot(d4,main="Plot (c=10)",ylab="Draws")
```



```r
acf(d4,main="ACF (c=10)")
```



```
hist(d4,main="Histogram (c=10)",xlab="Draws",freq=F)
lines(x,dbeta(x,6,4),col="red")
```



I would say the sampler corresponding to c=2.5 is most effective at drawing from the target distribution since its histogram resembles the histogram of the target distribution quite well and its autocorrelation plot features the fastest decay to insignicance (that means the least number of draws is needed).

4. Below we've created a new function mh1.beta with memory pre-allocation.

```r
mh.beta1 <- function(n, c) {
  phi=runif(1,0,1)
  draws <- numeric(n) #memory pre-allocation
  phi.update <- function(phi, c) {
    phi.p <- rbeta(1,c*phi,c*(1-phi))
    r <- dbeta(phi.p, 6,4)/dbeta(phi,6,4)/dbeta(phi.p,c*phi,c*(1-phi))*
      dbeta(phi,c*phi.p,c*(1-phi.p))
    if (runif(1) <= r){
      phi.p
    }else{
      phi
    }
  }
  for (i in 1:n) {
    draws[i] <- phi <- phi.update(phi, c)
  }
  return(draws)
}
```

We've just run each of the two functions 100 times with the arguments n=10000 and c=1 and computed the mean runtime for each function. It seems the function with memory pre-allocation is generally faster.

```r
t<-numeric(100)
for(i in 1:100){
  t_s=Sys.time()
  dd<-mh.beta(10000,1)
  t_e<-Sys.time()
  t[i]=t_e-t_s

}


mean(t)
```

```
## [1] 0.1350651
```

```r
t1<-numeric(100)
for(i in 1:100){
  t_s1=Sys.time()
  ee<-mh.beta1(10000,1)
  t_e1<-Sys.time()
  t1[i]=t_e1-t_s1

}

mean(t1)
```

```
## [1] 0.1236293
```

K-Means:

We have put the k-means function below. The function just takes a dataset of numeric columns (data) and the number of clusters as arguments. Initially, it picks k points from the dataset as centroids. For each iteration, it assigns the points to their nearest centroids and then re-calculates the centroids of the clusters (a centroid of a cluster is just an average of all the points in that cluster). The function breaks the loop whenever there are no more label changes and returns the labels of the points as the output.

```r
library(rattle)
```

```
## Warning: package 'rattle' was built under R version 4.0.3

## Loading required package: tibble

## Loading required package: bitops

## Rattle: A free graphical interface for data science with R.
## Version 5.4.0 Copyright (c) 2006-2020 Togaware Pty Ltd.
## Type 'rattle()' to shake, rattle, and roll your data.
```

```r
data(wine)
true_label=wine[,1]
data=wine[,-1]
data1=scale(data)
```

```r
kmeans=function(data,k){
  n=nrow(data)
  label=numeric(n)
  label1=numeric(n)
  indices=1:n
  center_i=sample(1:n,k)
  center=list()

  #Randomly find k centroids from the n observations
  for (i in 1:k)
    center[[i]]=data[center_i[i],]

  while (TRUE){

    #Assign each point to its closest centroid
    for (i in indices){
      dist=numeric(k)
      for (j in 1:k)
        dist[j]=sum((data[i,]-center[[j]])^2)
      label1[i]=which(dist==min(dist))
    }

    #Break if the label is no longer changed
    if (norm(label1-label,"2")==0)
      break
    label=label1

    #Calculate the new centroid for each cluster
    for(i in 1:k){
      indexk=which(label==i)

      center[[i]]=colMeans(data[indexk,])
    }



  }
  return(label)

}
```

As you can see from the plot of the clusters for the wine data below, the clusters do seem to be well-separated. There are some overlaps over the boundary, though.

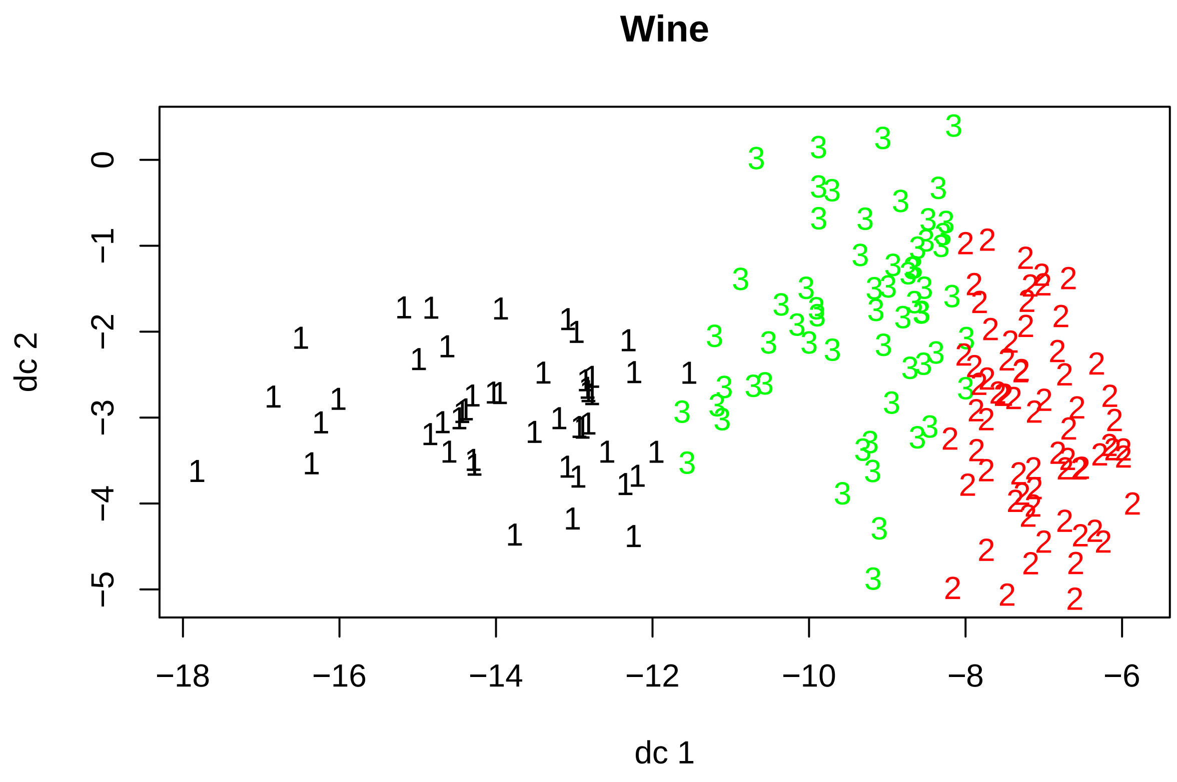
```r
library(fpc)
```

```
## Warning: package 'fpc' was built under R version 4.0.3
```

```r
library(modeest)
```

```
## Warning: package 'modeest' was built under R version 4.0.3
## Registered S3 method overwritten by 'statip':
##   method         from
##   predict.kmeans rattle
```

```
label=kmeans(data,3)
plotcluster(data,label,main="Wine")
```



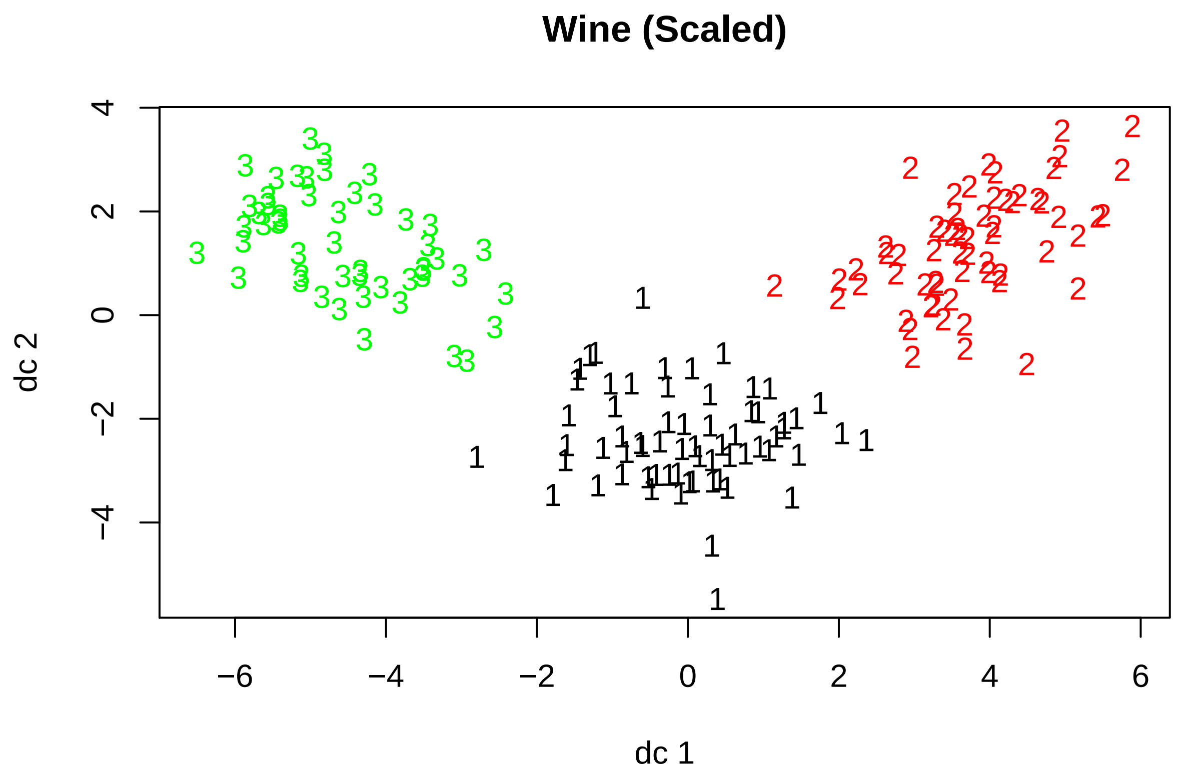
```
label_help=true_label
for(i in 1:length(label_help)){
  if (true_label[i]==1){
    label_help[i]=mfv(label[true_label==1])
  }else if(true_label[i]==2){
    label_help[i]=mfv(label[true_label==2])
  }else if(true_label[i]==3){
    label_help[i]=mfv(label[true_label==3])
  }
}
mean(label_help==label)
```

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

By comparing our learned labels with the true labels in the dataset, it seems 0.7022472 of the labels have been assigned correctly.

We have scaled the data below and redone the clustering. For each column, the scaling just subtracts the column mean from each entry and then divides it by the column standard deviation. As you can see from the plot below, the scaling does lead to clearer boundaries of the clusters. This is because for unscaled data columns with much larger scales affect the clustering results much more and clsutering basically takes away such effects.

```r
data1=scale(data)
label1=kmeans(data1,3)
plotcluster(data1,label1,main="Wine (Scaled)")
```



```r
label1_help=true_label
for(i in 1:length(label1_help)){
  if (true_label[i]==1){
    label1_help[i]=mfv(label1[true_label==1])
  }else if(true_label[i]==2){
    label1_help[i]=mfv(label1[true_label==2])
  }else if(true_label[i]==3){
    label1_help[i]=mfv(label1[true_label==3])
  }
}
mean(label1_help==label1)
```

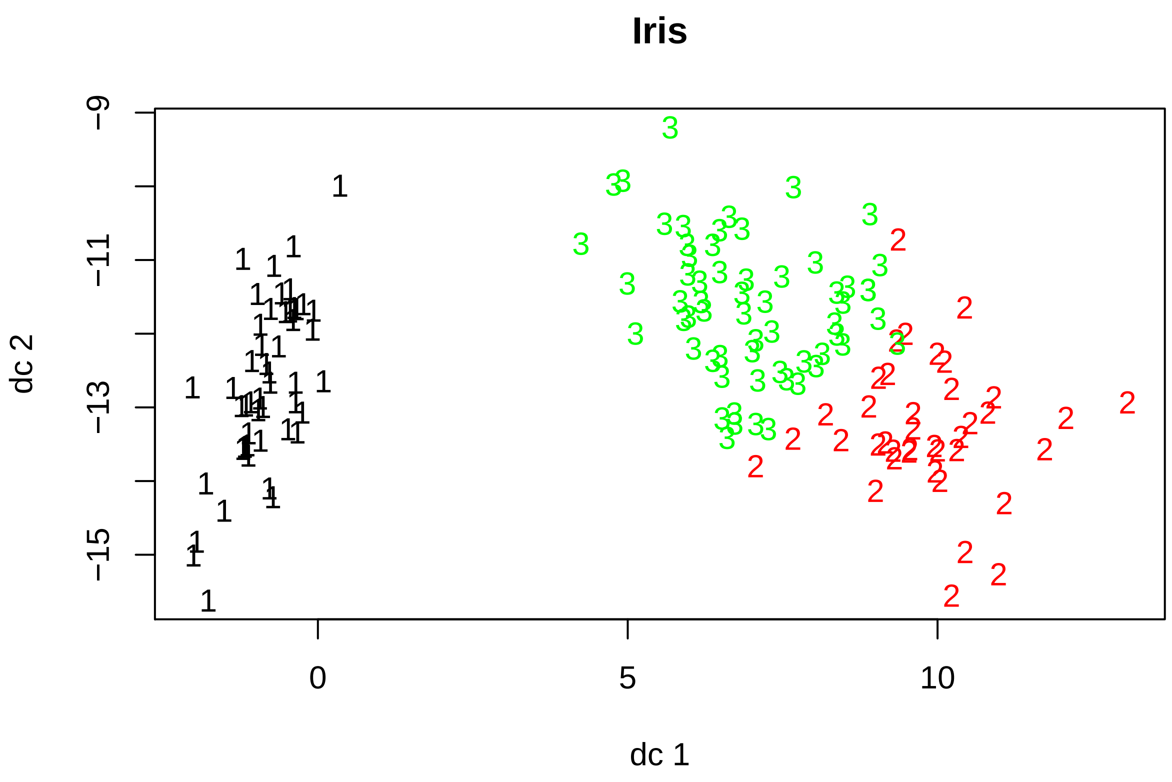
```
## [1] 0.9662921
```

For the scaled data, 0.9662921 of the labels have been assigned correctly, showing an significant improvement.

For the iris dataset, without scaling there are still overlaps over the boundaries in the plot. The resulting clusters look separated enough from each other, though.

```r
data(iris)
l_true=iris$Species

d<-iris[,-5]
l<-kmeans(d,3)
plotcluster(d,l,main="Iris")
```



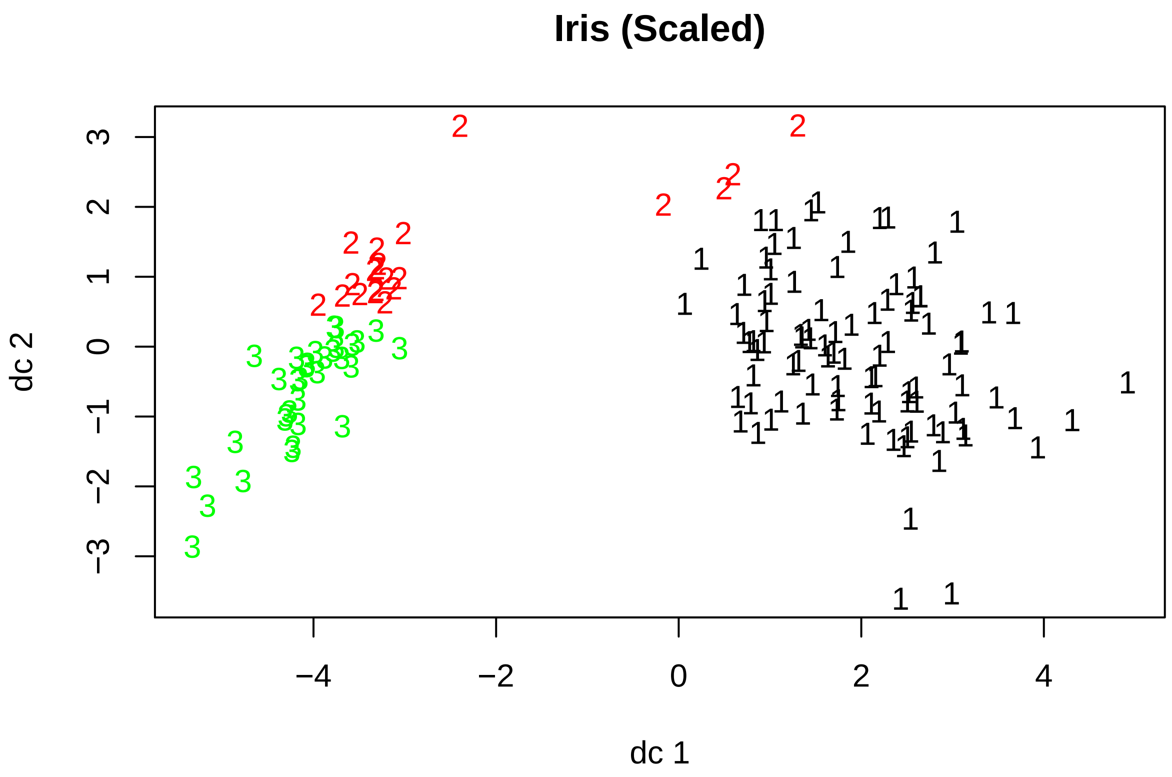
```r
l_help=l
for(i in 1:length(l_help)){
  if (l_true[i]=="setosa"){
    l_help[i]=mfv(l[l_true=="setosa"])
  }else if(l_true[i]=="versicolor"){
    l_help[i]=mfv(l[l_true=="versicolor"])
  }else if(l_true[i]=="virginica"){
    l_help[i]=mfv(l[l_true=="virginica"])
  }
}
mean(l_help==l)
```

```
## [1] 0.8866667
```

It seems 0.8866667 of the labels have been assigned correctly.

This time scaling does not help with the overlaps probably because the original columns in the dataset do not vary that much in terms of scales.

```r
d1<-scale(d)
l1<-kmeans(d1,3)
plotcluster(d1,l1,main="Iris (Scaled)")
```



```r
l1_help=l1
for(i in 1:length(l1_help)){
  if (l_true[i]=="setosa"){
    l1_help[i]=mfv(l1[l_true=="setosa"])
  }else if(l_true[i]=="versicolor"){
    l1_help[i]=mfv(l1[l_true=="versicolor"])
  }else if(l_true[i]=="virginica"){
    l1_help[i]=mfv(l1[l_true=="virginica"])
  }
}
mean(l1_help==l1)
```

```
## [1] 0.86
```

With scaling 0.86 of the labels have been assigned correctly. There has not been an improvement.