STAT 37810 final project

Hsiang Wang

11/1/2020

1

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

1.1

The algorithm starts from create an empty vector that stores ϕ. For each draw, generates proposal phi by proposal function. Then, computes acceptance probability

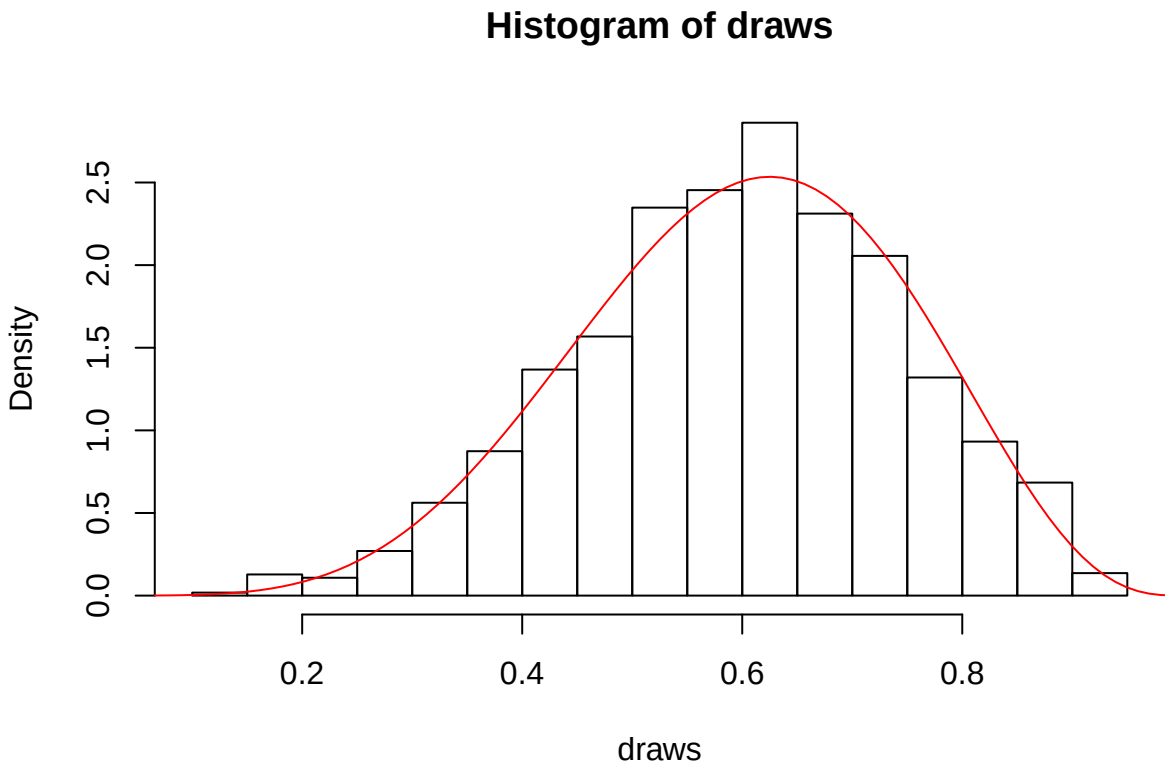
$$r = \frac{p(\phi_{prop})/J_t(\phi_{prop}|\phi_{old})}{p(\phi_{old})/J_t(\phi_{old}|\phi_{phi})}$$

where p follows $beta(6, 4)$ and J follows proposal distribution. Then new phi is proposed phi if r is smaller than random uniform or else it is current phi.

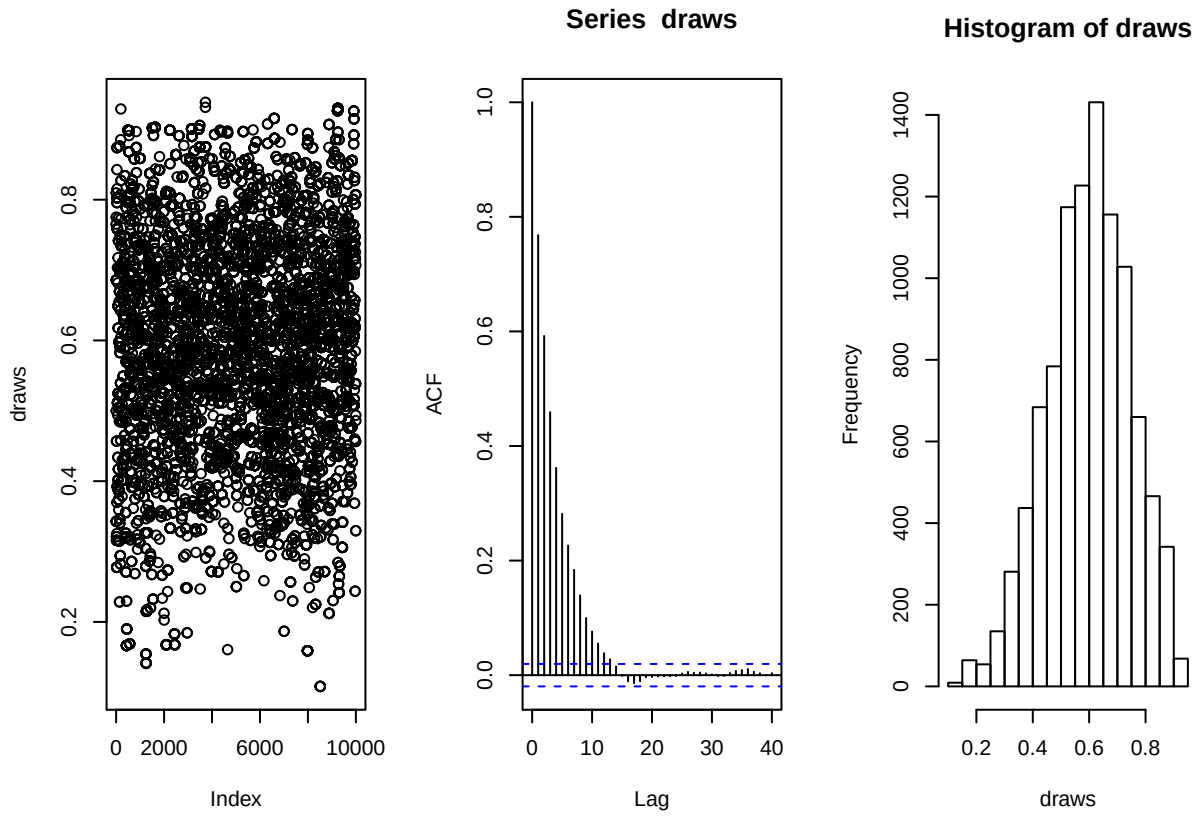
1.2

```
proposal <- function(prop,old,c){
  #This function computes probability from proposal functino
  dbeta(prop,c*old,c*(1-old))
}

MHbeta <- function(nsample,initial,alpha,beta,c){
  # This function computes Metropolis-Hastings algorithm for beta
  phi=c()
  phi[1]=initial
  for (i in 1:nsample){
    current_phi=phi[i]
    proposal_phi=rbeta(1,c*current_phi,c*(1-current_phi))
    A=(dbeta(proposal_phi,alpha,beta)/proposal(proposal_phi,current_phi,c))/(dbeta(current_phi,alpha,be
    if(runif(1)<A){
      phi[i+1]=proposal_phi
    }
    else{
      phi[i+1]=current_phi
    }
```

```
  }
  phi[2:(nsample+1)]
}


draws=MHbeta(10000,0.5,6,4,1)
hist(draws,freq =F)
xx = seq(0,1,length=100)
lines(xx,dbeta(xx,6,4),col="red")
```




```
par(mfrow=c(1,3)) #1 row, 3 columns
plot(draws); acf(draws); hist(draws) #plot commands
```




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

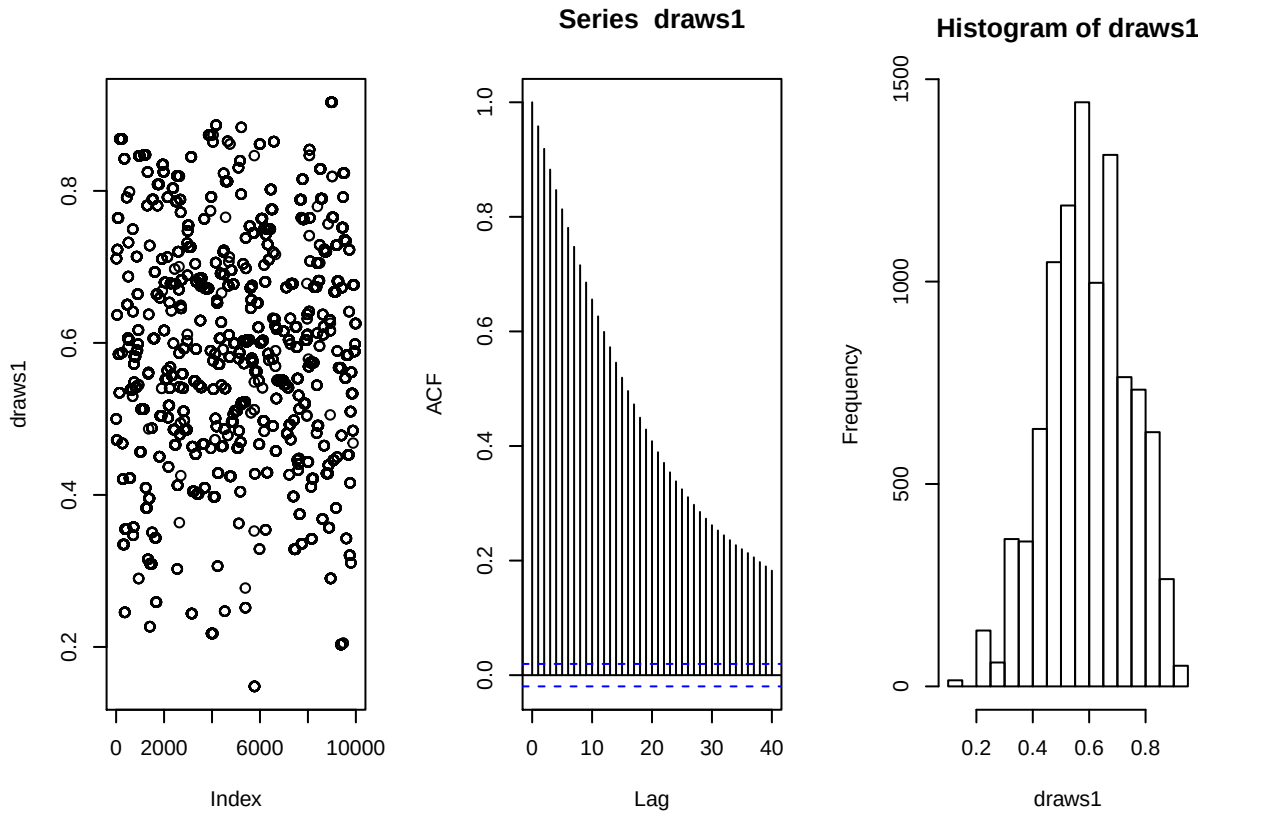
```
## Warning in ks.test(draws, "pbeta", 6, 4): ties should not be present for the
## Kolmogorov-Smirnov test
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  draws
## D = 0.023768, p-value = 2.479e-05
## alternative hypothesis: two-sided
```

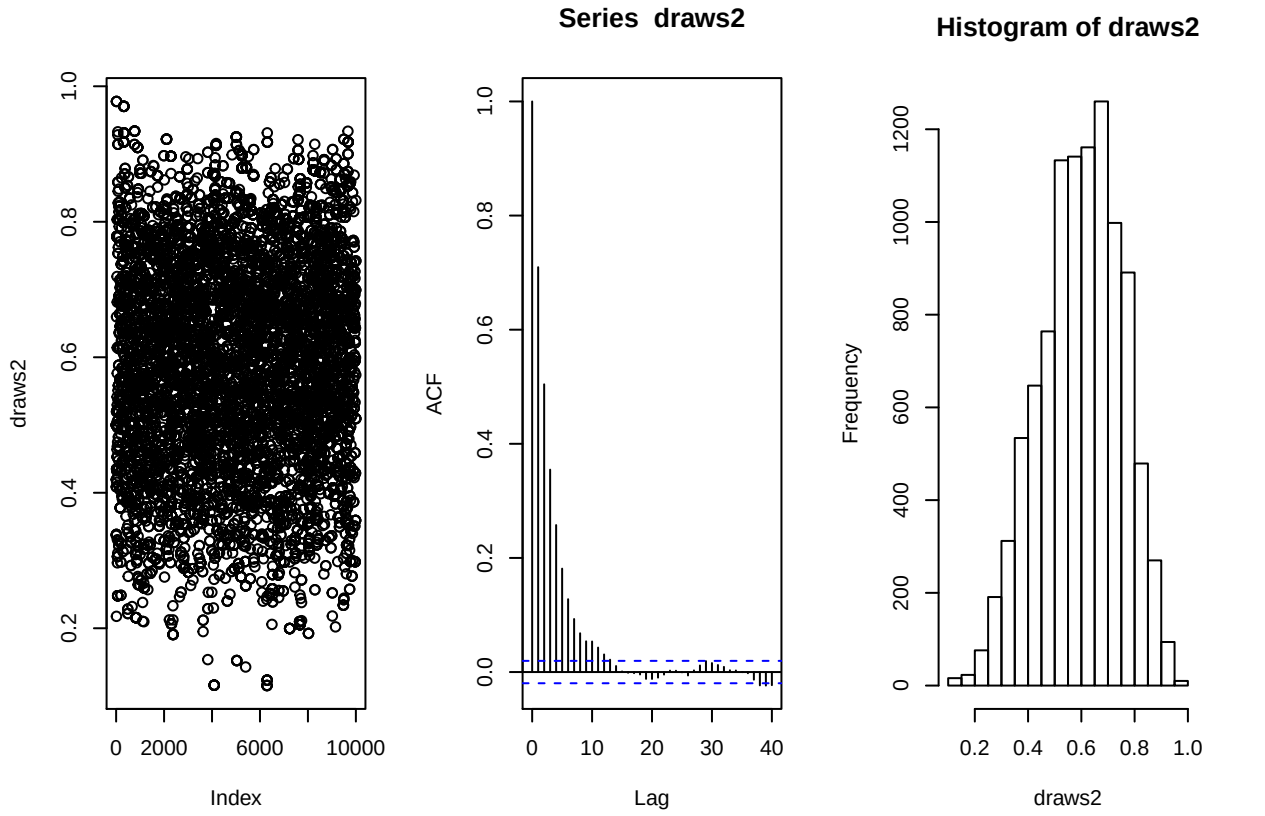
Based on the histogram, MH algorithms has simulated $Beta(6,4)$ well.
Based on Kolmogorov-Smirnov statistics, it is $Beta(6,4)$.

1.3

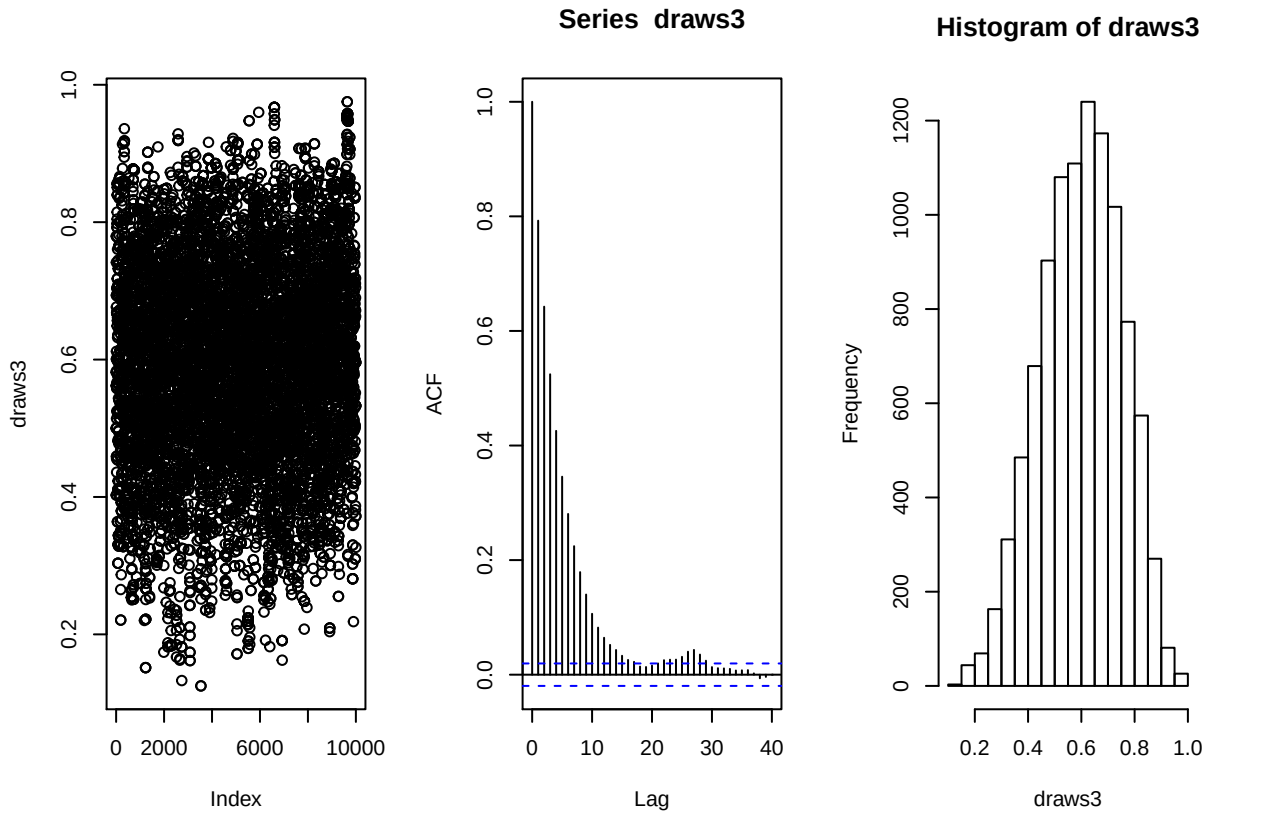
```
par(mfrow=c(1,3)) #1 row, 3 columns
draws1=MHbeta(10000,0.5,6,4,0.1) #c=0.1
draws2=MHbeta(10000,0.5,6,4,2.5) #c=2.5
draws3=MHbeta(10000,0.5,6,4,10)  #c=10
plot(draws1); acf(draws1); hist(draws1) #plot commands
```



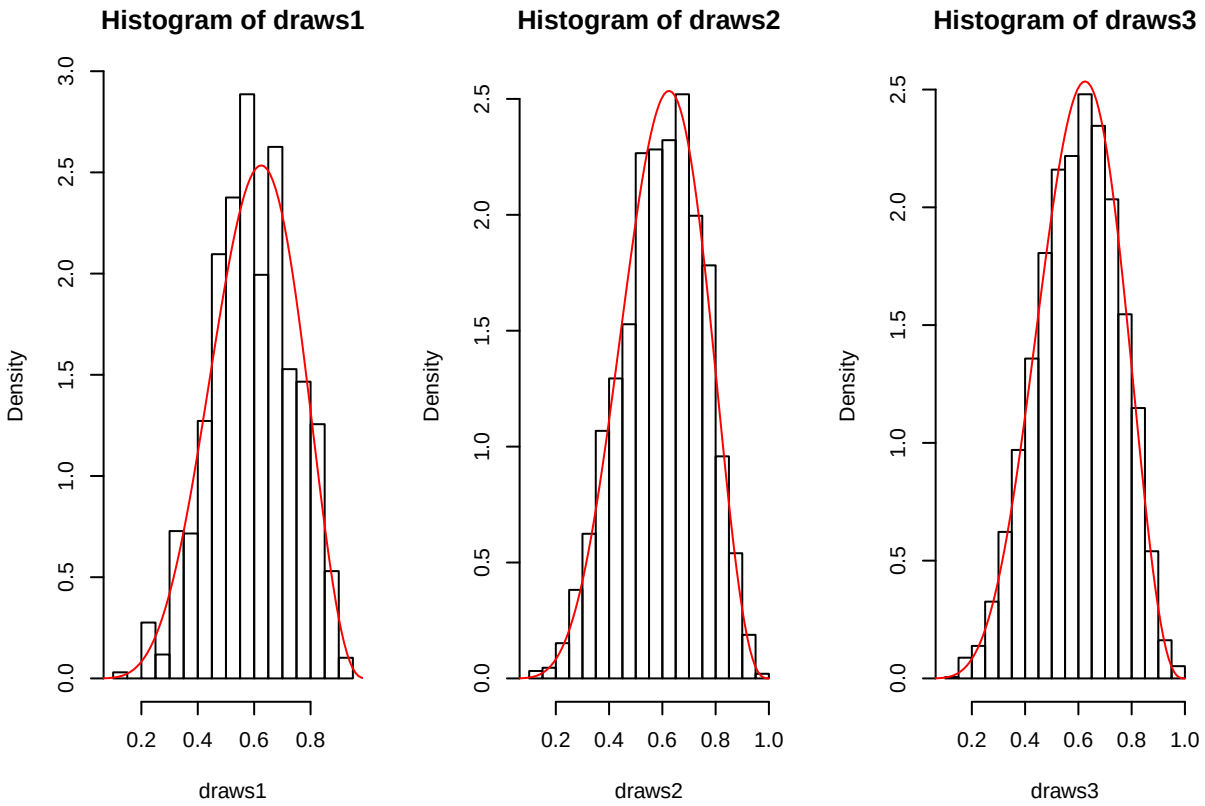

```
plot(draws2); acf(draws2); hist(draws2) #plot commands
```



```r
plot(draws3); acf(draws3); hist(draws3) #plot commands
```




```r
hist(draws1,freq =F)
xx = seq(0,1,length=100)
lines(xx,dbeta(xx,6,4),col="red")

hist(draws2,freq =F)
xx = seq(0,1,length=100)
lines(xx,dbeta(xx,6,4),col="red")

hist(draws3,freq =F)
xx = seq(0,1,length=100)
lines(xx,dbeta(xx,6,4),col="red")
```



Based on the auto correlation plot, c=0.1 has highest auto correlation, c=10 comes next and c=2.5 has the least. This is because large or samll c will lead to slow change in phi. Therefore, it takes longer time to reach stationary distribution and thus requires higher burn-in rate.

1.4

```
#not pre-allocate
MHbeta <- function(nsample,initial,alpha,beta,c){
  # This function computes Metropolis-Hastings algorithm for beta
  phi=c()
  phi[1]=initial
  for (i in 1:nsample){
    current_phi=phi[i]
    proposal_phi=rbeta(1,c*current_phi,c(1-current_phi))
    A=(dbeta(proposal_phi,alpha,beta)/proposal(proposal_phi,current_phi,c))/(dbeta(current_phi,alpha,be
    if(runif(1)<A){
      phi[i+1]=proposal_phi
    }
    else{
      phi[i+1]=current_phi
    }
  }
  phi[2:(nsample+1)]
}

t0=Sys.time()
draws=MHbeta(10000,0.5,6,4,1)
t1=Sys.time()
t1-t0 #not pre allocate
```

```
## Time difference of 0.1932991 secs
```

```
#pre-allocate
MHbeta <- function(nsample,initial,alpha,beta,c){
  # This function computes Metropolis-Hastings algorithm for beta
  phi=rep(0,1+nsample)
  phi[1]=initial
  for (i in 1:nsample){
    current_phi=phi[i]
    proposal_phi=rbeta(1,c*current_phi,c(1-current_phi))
    A=(dbeta(proposal_phi,alpha,beta)/proposal(proposal_phi,current_phi,c))/(dbeta(current_phi,alpha,be
    if(runif(1)<A){
      phi[i+1]=proposal_phi
    }
    else{
      phi[i+1]=current_phi
    }
  }
  phi[2:(nsample+1)]
}

t0=Sys.time()
draws=MHbeta(10000,0.5,6,4,1)
t1=Sys.time()
t1-t0 #not pre allocate
```

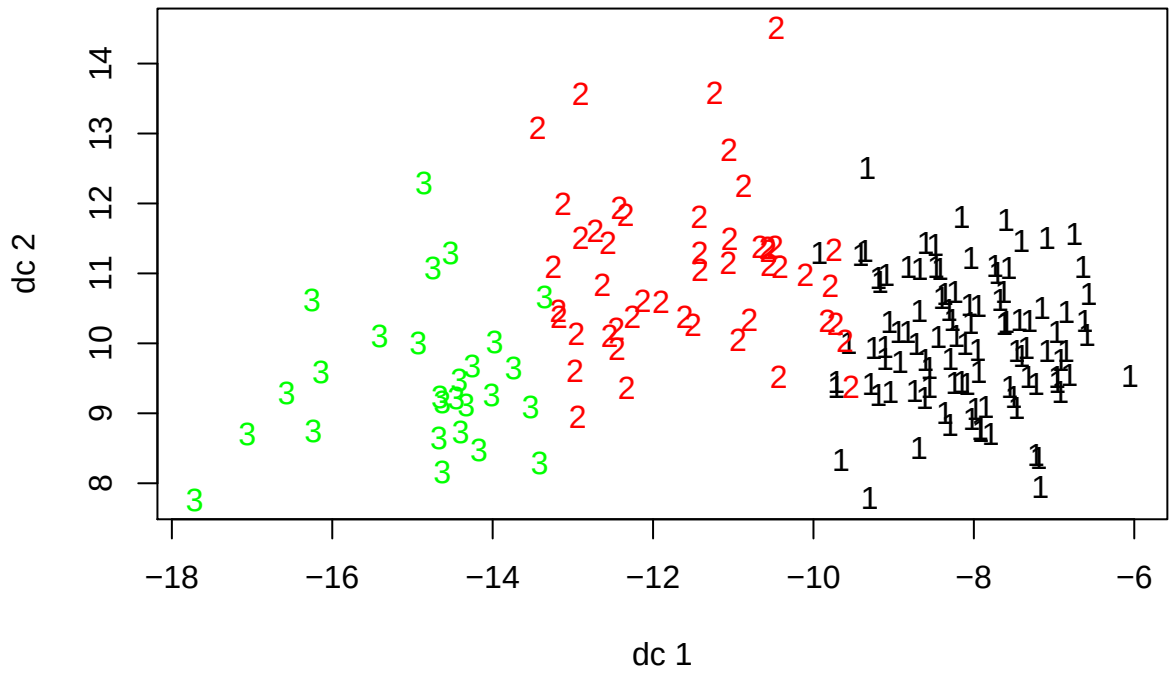
```
## Time difference of 0.1362331 secs
```

Pre-allocation is faster

3

```
data(wine, package="rattle")
df=wine[,-1]
k=3
kmean <- function(df,k,tol=1){
  #This function computes k mean of dataframe df
  len=ncol(df) #number of features in data frame
  mat=matrix(0,nrow = len,ncol=k) #create empty matrix to storage k means

  for (i in 1: len){
    #computes initial value of means
    mat[i,]=runif(3,min(df[,i]),max(df[,i]))
  }

  computelabel <- function(df,mat){
    #This functions label data point by computing and choosing minimum euclidean
    #distance between data points and k means
    df_nl <- df[,1:len]# remove label from df
    computeenclideandistance <- function(df,mat,k){
      #This sub function computes enclidean distance between mean and data points
      dist(rbind(df, t(mat)[k,]))
    }
    meandf <- apply(df_nl,1,computeenclideandistance,mat=mat,k=1) %>% as.data.frame
    for (i in 2:k){
      meandf <- meandf %>% cbind(apply(df_nl,1,computeenclideandistance,mat=mat,k=i))
```

```
    }
    label <- meandf%>%apply(1,which.min) # find minimum distance
    dfwithlabel <- cbind(df_nl,label)
    dfwithlabel
  }

  computekmean <- function(df,mat){
    #This functions updates mean for each iteration in k mean
    for (i in 1:k){
      newmean <- df%>%as.data.frame()%>%filter(label==i)%>%apply(2,mean)
      mat[,i] <- newmean[1:len]
    }
    mat
  }

  while(TRUE){ #iteration
    df <- computelabel(df,mat) #compute label
    updated_mat=computekmean(df,mat) #update mean
    if (norm(updated_mat-mat)<tol){ #if smaller than tolerance
      return(as.data.frame(df))
    }
    mat=updated_mat
  }
}


test <- kmean(df,3,1) # run kmean with k=3, tol=1
#test

library('fpc')
plotcluster(df, test$label)
```



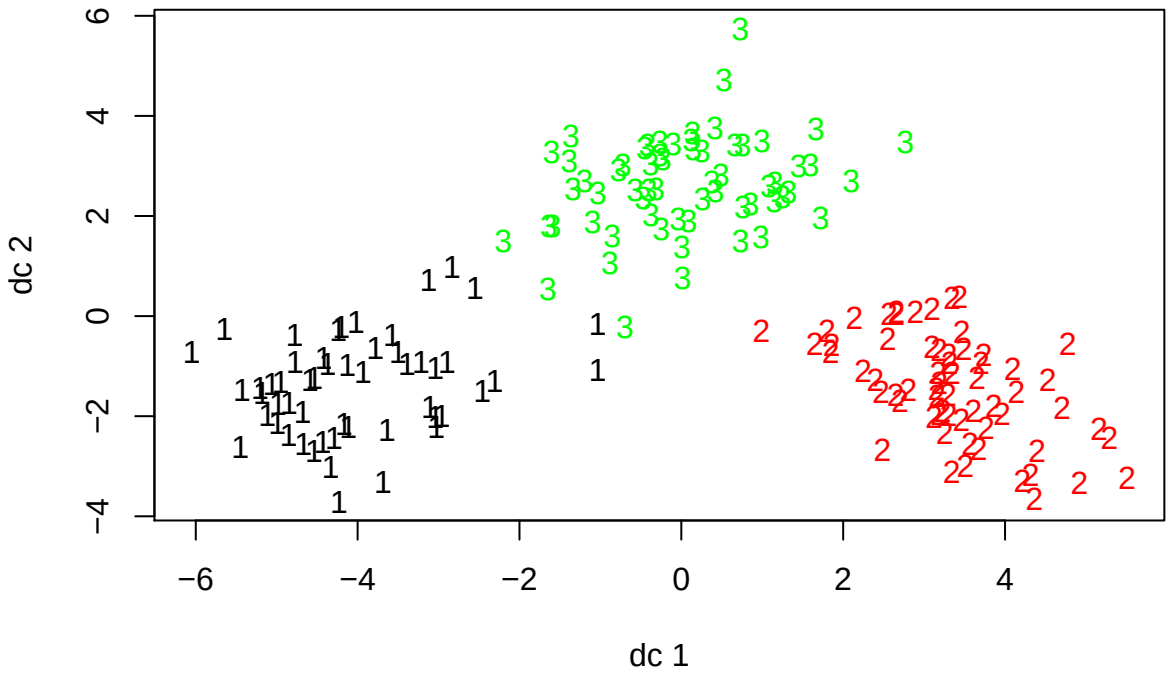
```
#df=data%>%mutate_all(scale)
```

The data is well seperated.

report

The algorithm starts with initialize matrix for recording k means. There are two sub functions in the algorithms: computelabel and computekmean. computelabel returns label based on the minimum euclidean distance between data points and k means. computekmean returns new mean matrix based on the current label. For each iteration, these two subfucnctions are executed. New label and mean matrix is updated. The algorithm stops running when the norm of new mean matrix and previous mean matrix is below tolerance.

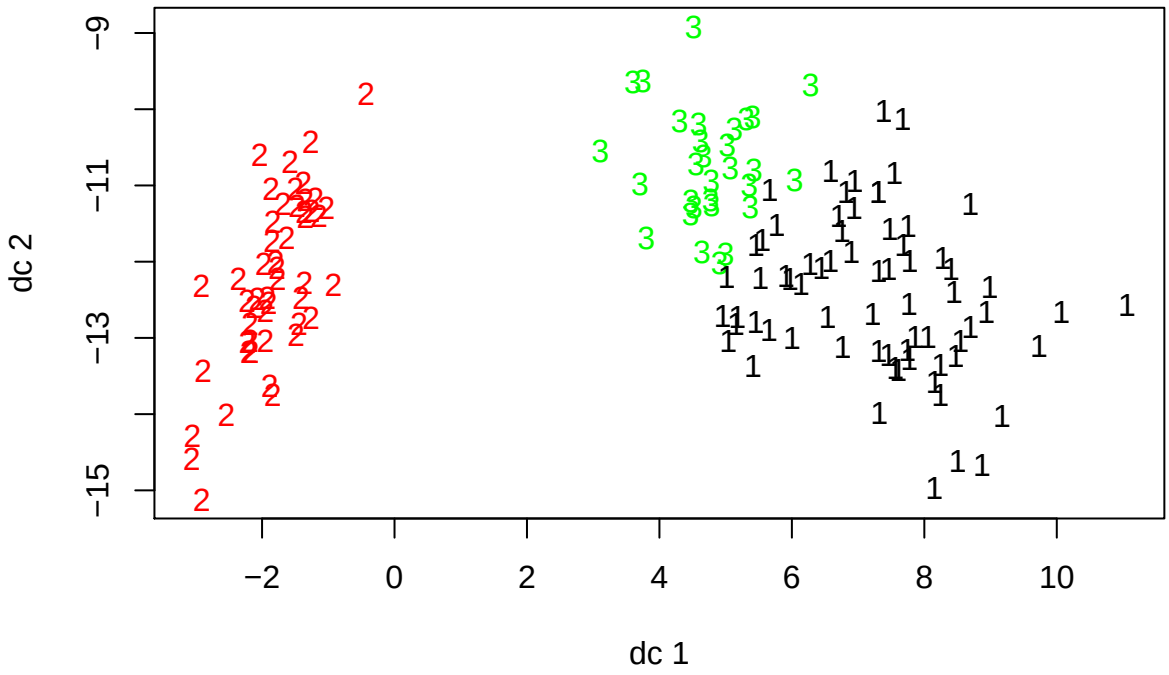
```
data.train <- scale(wine[-1])
test2 <- kmean(data.train,3,1)
plotcluster(data.train, test2$label)
```



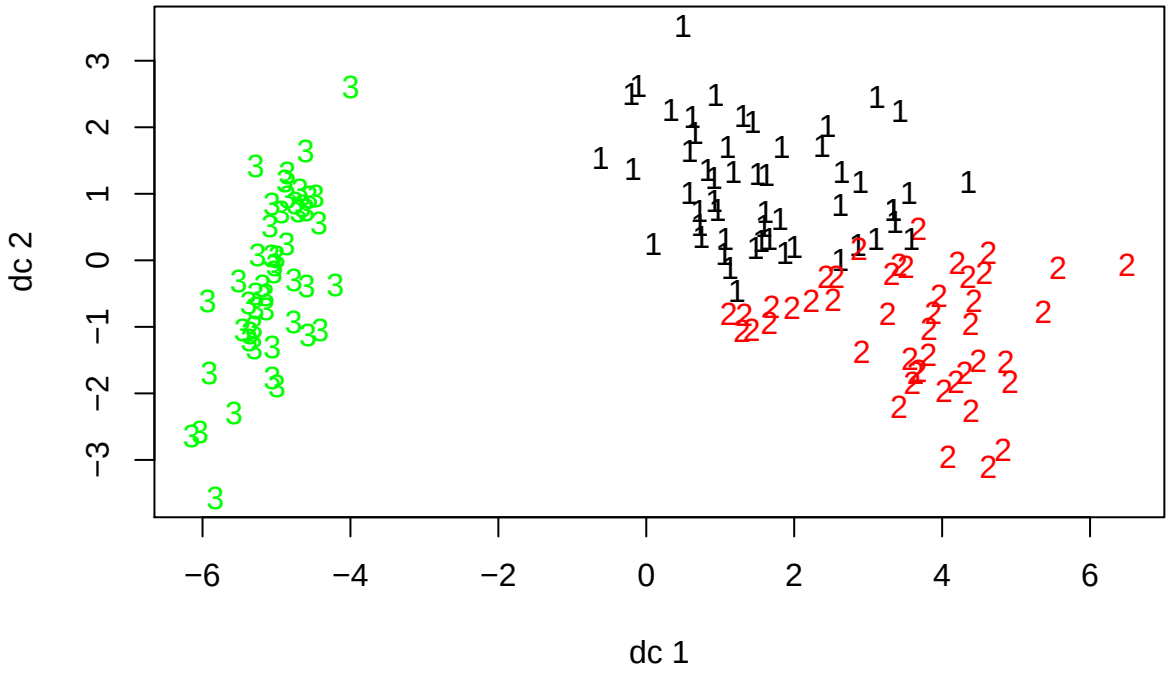
The function scale normalize the dataframe by first subtract the mean and then divided it by its standard deviation.

The results are different. This is because for unnormalized version, k mean weights more for features with larger value, which would lead to higher euclidean distance.

```
data(iris)

df=iris[,-5]
test <- kmean(df,3,1)
plotcluster(df, test$label)
```




```
data.train <- scale(iris[,-5])
test2 <- kmean(data.train,3,1)
plotcluster(data.train, test2$label)
```



The visualization looks well in both unnormalize and normalize version.