37810 Final Project report

Hanyang Peng, Yi Jin, Yuhan Li

Stat 378 Final Project Matropolis-Hasting Alagorithm

Report on Matropolis-Hasting Alagorithm

Problem 1

Introduction

In order to understand a certain distribution $p(x)$, the idea of MCMC is to simulate a Markov chain whose equilibrium distribution is $p(x)$, and metropolis-hasting algorithm provides a way to generate the Markov chain. Giving a start value, and with the help of proposal function, M-H produce a new proposed value based only on its previous value in the Markov chain and decide whether to accept the new value by the acceptance rate. If accepted, the chain moves to the new value, and if not, the chain stays with the old value. By repeating these steps, M-H could generate a chain to simulate the distribution.

code of algorithm

Below is the metropolis-hasting algorithm sample from a beta distribution ϕ with parameters (6,4). We would explain the algorithm and how we implemented it later.

```r
mh.beta <- function(iterations, start,shape1, shape2,c) {
  phi.old <- start
  #set the intitial value to be the start value
  draws <- c()
  #save the values of chain in a vector
  phi.update <- function(phi.old, shape1, shape2) {
    phi.new <- rbeta(1, c*phi.old, c*(1-phi.old))
    #generate the new value
    accept.prob <- dbeta(phi.new, shape1 = shape1, shape2=shape2)/
      dbeta(phi.old,shape1 = shape1, shape2 = shape2)*
      (dbeta(phi.old,c*phi.new,c*(1-phi.new))/dbeta(phi.new,c*phi.old,c*(1-phi.old)))
    #caculate the acceptance probability of the new value with a correction factor
    if (runif(1) <= accept.prob)
    {phi.update=phi.new}
    #if the acceptance probability is larger then accept the new value
    else
        {phi.update=phi.old}
  }
  # if the acceptance probability is lower then stay with the old value
  for (i in 1:iterations) {
    draws[i] <- phi.old <- phi.update(phi.old, shape1 = shape1, shape2 = shape2)
  }
  #give the new value to the chain
  return(draws[1:iterations])
}
```

code instructions

The algorithm is generally a function to describe the process of M-H. The input of the function would be the times of iterations, which refers to the length of chain we want, the start value, two shape parameters of the beta distribution, and the parameter "c" in the proposal function.

In the function, we use "phi.old" to save the old value of the chain, use "phi.new" to save the new value generated from the old value, the function we use to generate the new value is called "phi.update", and we save the result of the chain in the vector "draw".

When iterating for the first time, we set the "phi.old" to be the start value. Then, we generate a new value based on the start value by proposal function. As mentioned in the problem, the proposal function is of the form $\phi_{old}|\phi_{new} \sim Beta(c * \phi_{old}, c * (1 - \phi_{old}))$, which means the new value is generated by producing a random number from the given beta distribution. We notice that, in the beta distribution of the proposal function, the shape parameters are determined by a variable c and the old value of ϕ, that's just how old value would generate a new value in M-H algorithm.

proposal function and criterion for accepting

After generating the new value, we need to decide whether to accept the new value.

Towards a normally distributed proposal, the criterion for accepting or rejecting the new value is as follow: (we denote x as old value, and x^* as the new value)
1. If $p(x^*) \geq p(x)$, that means $p(x)$ has high density near x^*, and will be accepted as the new value in the chain.
2. If $p(x^*) < p(x)$, that indicates that $p(x)$ has low density near x^*. In that case, the new value may still be accepted, but only randomly, and with a probability $\frac{p(x^*)}{p(x)}$.

However, we must notice that beta distribution is not symmetric, which means the proposal function needs a correction factor to suit the reversibility condition of Markov chain, that is the transition probability from $x^{(t)}$ to $x^{(t-1)}$ should be equal to that from $x^{(t-1)}$ to $x^{(t)}$.

As a result, we have to introduce a correction factor, which is defined as $C = \frac{q(x^{(t-1)}|x^*)}{q(x^*|x^{(t-1)})}$, where q refers to the density function of each conditional distribution. Therefore, we could calculate the acceptance rate α by $\alpha = \min\left(1, \frac{p(x^*)}{p(x)} \times C\right)$.

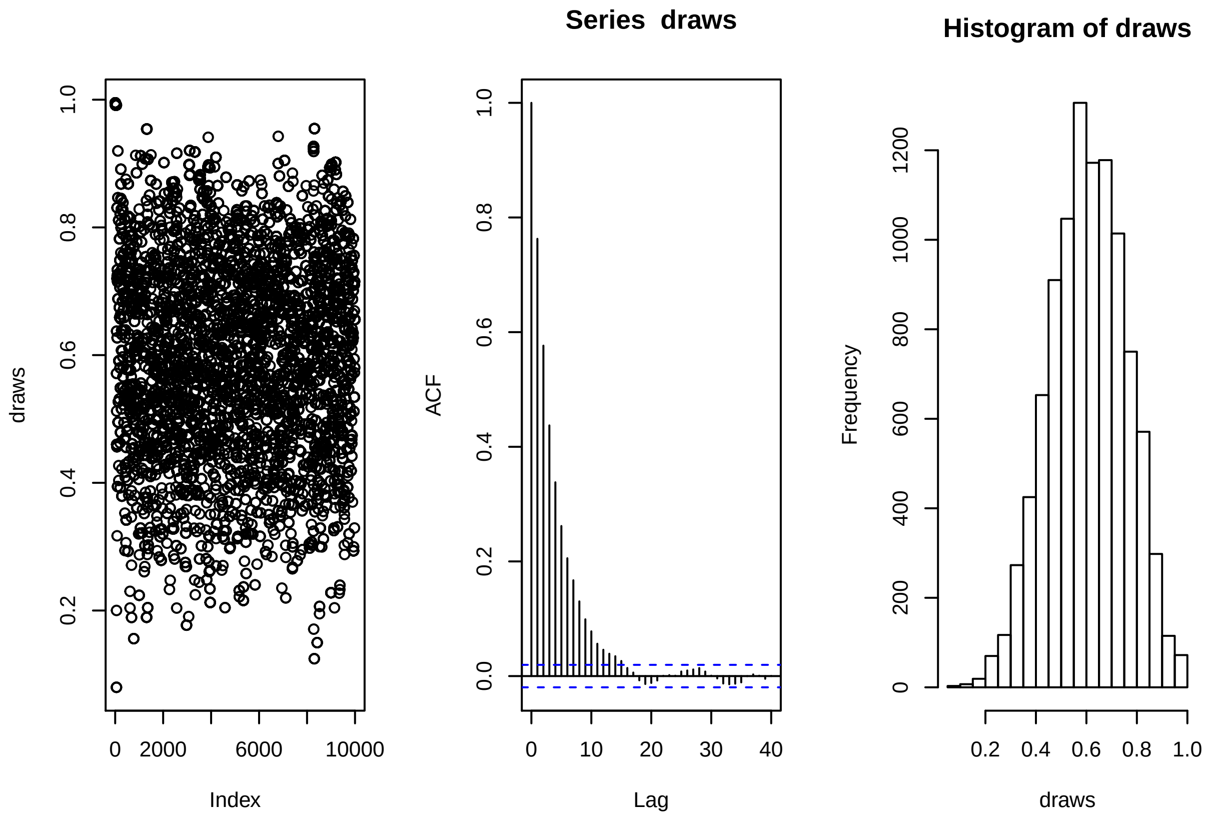
Taking the asymmetrical proposal function into consideration, we draw a random number u from distribution $Unif(0, 1)$ each time to implement such process, and the criterion for accepting or rejecting the new value is changed into the following criteria:
1. If $u \leq \alpha$, we accept the new value.
2. If $u > \alpha$ the new value would be rejected and the chain would stay with the old value.

Finally, we save the value into the chain vector, and update the value of ϕ. In further iterations, we repeat this process until we get a chain with desired length.

Problem 2

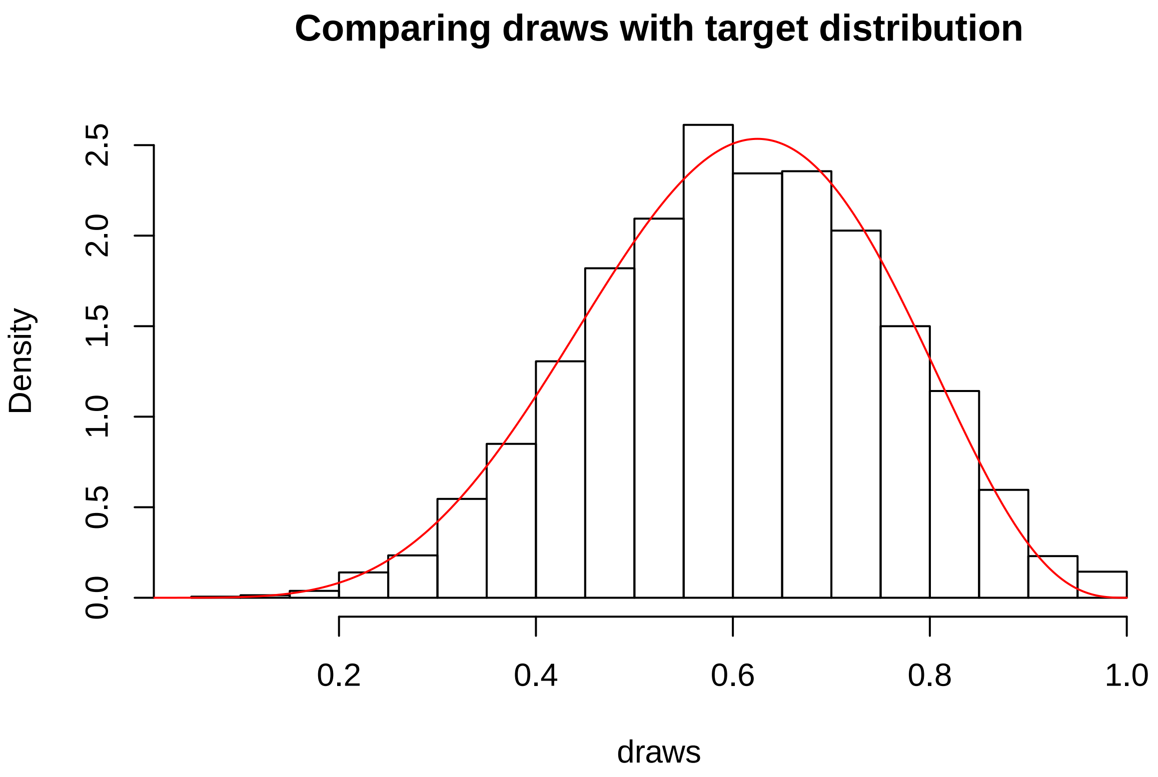
The trace plot, autocorrelation plot, and histogram of the draws are as below.

```r
draws<-mh.beta(10000, start = runif(1),shape1 = 6, shape2 = 4,c=1)
par(mfrow=c(1,3))  #1 row, 3 columns
 plot(draws)
 acf(draws)
 hist(draws)
```



The comparison between the histogram of draws and target distribution of Beta(6,4) is as below.

```
x=seq(0,1,0.001)
hist(draws,freq=F,main="Comparing draws with target distribution")
lines(x,dbeta(x,6,4),col="red")
```



```
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 = 0.025471, p-value = 4.631e-06
## alternative hypothesis: two-sided
```

According to the comparison of histogram and the result of K-S test (the p-value is quite small), the distribution of draws follows the target distribution Beta(6,4) quite well. The histogram shows that the sampler generally have the same distribution as beta distribution with shape parameters (6,4). The result of K-S test shows that we are quite confident that the sampler and the target distribution shares the same distribution, since both the distance and the p-value of K-S test are quite low.

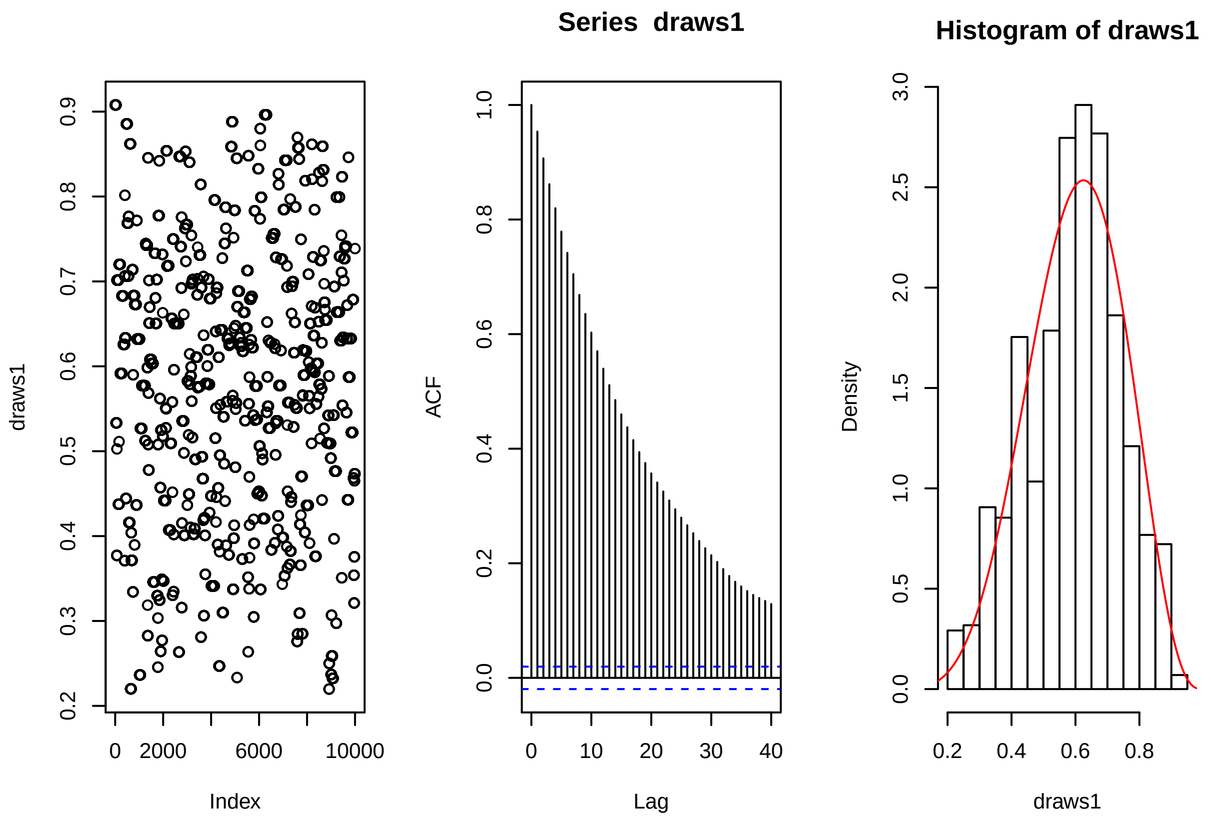
Therefore, we can conclude that the sampler could simulate the target distribution quite accurately.

Problem 3

Set $c = 0.1, 2.5, 10$ separately, we rerun the M-H algorithm, and have the trace plots, autocorrelation plots and histograms for these three samplers. The related plots are as below.

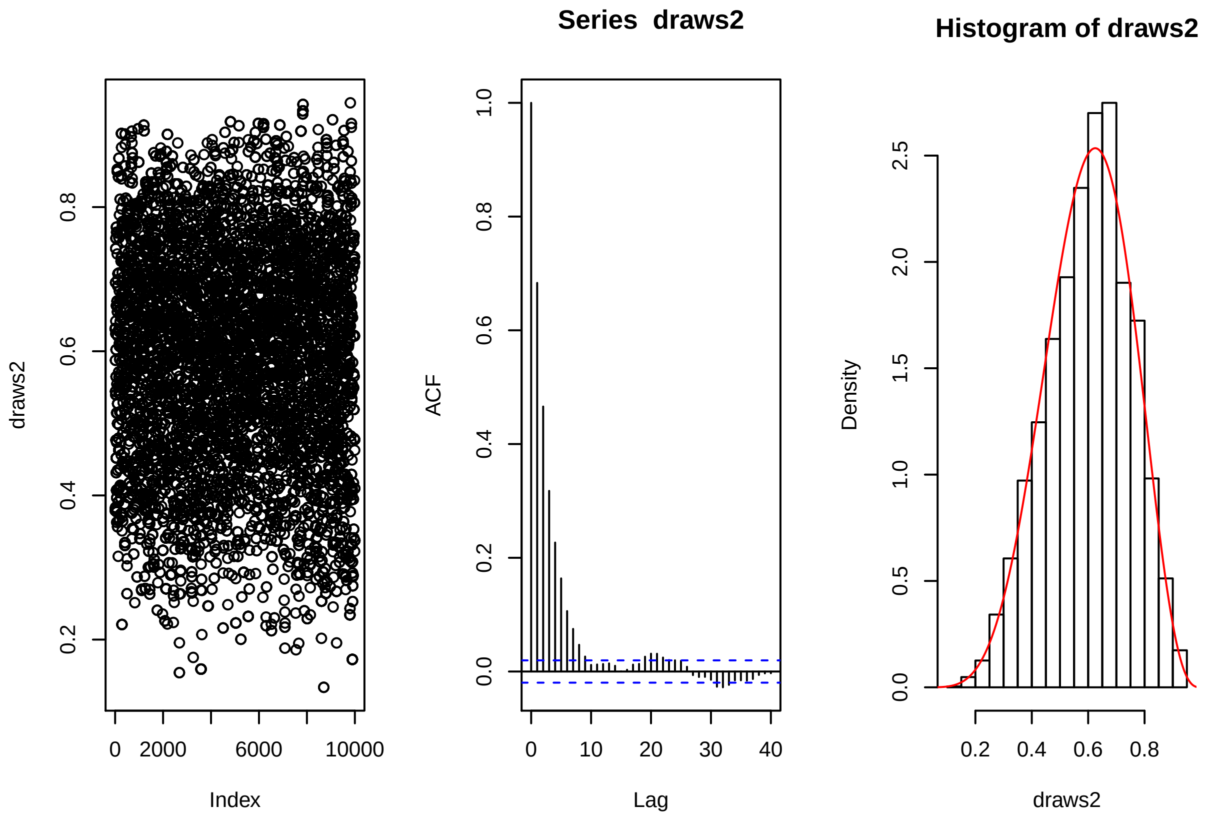
For $c = 0.1$:

```
draws1<-mh.beta(10000, start = runif(1),shape1 = 6, shape2 = 4,c=0.1)
par(mfrow=c(1,3))  #1 row, 3 columns
 plot(draws1)
 acf(draws1)
 hist(draws1,freq=F)
lines(x,dbeta(x,6,4),col="red")
```



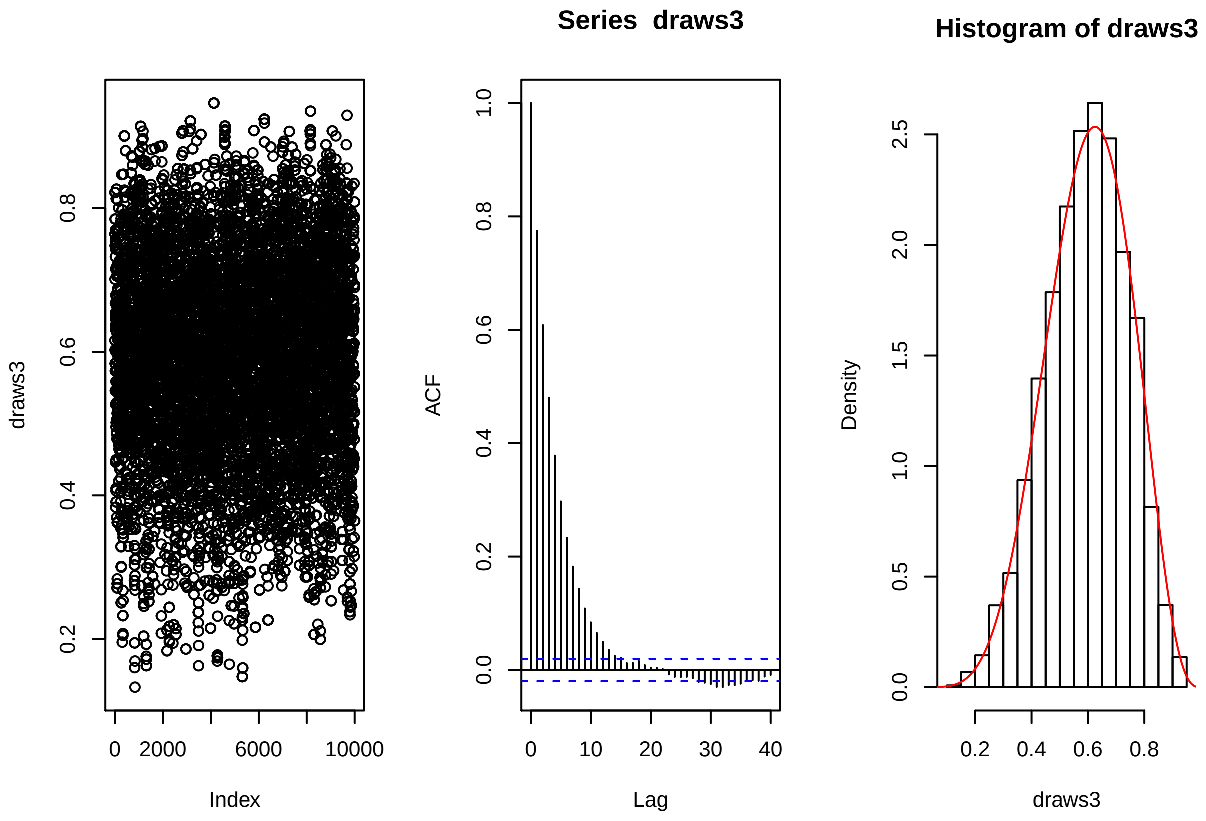
For $c = 2.5$:

```
draws2<-mh.beta(10000, start = runif(1),shape1 = 6, shape2 = 4,c=2.5)
par(mfrow=c(1,3))  #1 row, 3 columns
 plot(draws2)
 acf(draws2)
 hist(draws2,freq=F)
 lines(x,dbeta(x,6,4),col="red")
```



For $c = 10$:

```
draws3<-mh.beta(10000, start = runif(1),shape1 = 6, shape2 = 4,c=10)
par(mfrow=c(1,3))  #1 row, 3 columns
 plot(draws3)
 acf(draws3)
 hist(draws3,freq=F)
 lines(x,dbeta(x,6,4),col="red")
```



We compare these samplers by comparing their acceptance rates. Generally speaking, neither a too high or too low acceptance rates would benefit the M-H algorithm. If the acceptance rates is too high, it indicates that the proposal distribution is too narrow compared with the target distribution. In that case, the chain we get actually is just hanging around at a certain range, which would lead to bad simulation. Meanwhile, if the acceptance rates is too low, we would stay at the same point for most of the time, this would cause bad simulation as well. Thus, we usually take an acceptance rates between 20% and 30% to be optimal.

We define a function to calculate the acceptance rate with c set to different values, the related codes are as below.

```
accept.rate=function(draws){
  j=0
  for(i in 1:(length(draws)-1)){
    if (draws[i]==draws[i+1]){
      j=j
    }
    else{j=j+1}
  }
  return(j/length(draws))
}
accept.rate(draws) #c=1
```

```
## [1] 0.2585
```

```
accept.rate(draws1) #c=0.1
```

```
## [1] 0.0404
```

```
accept.rate(draws2) #c=2.5
```

```
## [1] 0.4223
```

```r
accept.rate(draws3) #C=10
```

```
## [1] 0.6707
```

We notice that as c becomes larger, the acceptance rates also get higher. This is because that for a beta distribution $Beta(\alpha, \beta)$, its variance would be $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$. Thus, when c becomes larger, the two shape parameters in the proposal function simultaneously get larger, which leads to the decrease in the variance of the proposal function. Therefore, larger c would lead to lower variance in the proposal distribution, which means the new value we generate each time would generally be closer to the old value, and the acceptance rates would go higher.

As we mentioned above, an acceptance rates between 20% and 30% would be optimal, thus $c = 1$ would be the most appropriate choice for the sampler, and would be most effective sampler drawing from the target distribution.

Stat 37810 Final Project - Gibbs Sampling

1.

Let $p(x|y) = C_1 y e^{-yx}, p(y|x) = C_2 x e^{-xy}$, then because they are conditional distributions, so we can get that
$\int_0^B C_1 y e^{-yx} dx = 1, \int_0^B C_2 x e^{-xy} dy = 1$
$\Rightarrow C_1 = \frac{1}{1-e^{-yB}}, C_2 = \frac{1}{1-e^{-xB}}$

So the CDF of these distributions are
$F_X(x) = \int_0^x \frac{1}{1-e^{-yB}} y e^{-yu} du = \frac{1-e^{-yx}}{1-e^{-yB}}$
Similar, we get
$F_Y(y) = \frac{1-e^{-yx}}{1-e^{-xB}}$
So the inverse of these CDF are:
$F_X^{-1}(u) = -\frac{1}{y} log(1-u(1-e^{-yB}))$
$F_Y^{-1}(u) = -\frac{1}{x} log(1-u(1-e^{-xB}))$

So in this case, we can change the example sampler like this:
First, we will check if the input value is correct, both B and T should be larger than 0, T's type should be int, and B's type should be int of float.
Then, we create a T*2 Numpy matrix as the output matrix, the first column is X, and the second column is Y.
After that, we set the start value of x and y (They can't be 0 because they are the denominators). For each loop, we generate different u's from Unif[0,1] and use Inverse Transform Sampling, which inverse function of CDF is calculated before, to generate the samples from the conditional distribution. We also set thin=1000 so we can get the sample more correctly.
Finally, we put the value of x and y in the output matrix. And we return the output matrix after the loop.
The code is showed below

In [1]:

```
import numpy as np
import random,math
import matplotlib.pyplot as plt
```

In [2]:

```
def gibbs(B=5.0,T=1000,thin=1000):
    if(type(B)==int):  #change the type of B to float
        B=float(B)
    if(type(T)!=int or type(B)!=float or type(thin)!=int):  #check if we input the correct type
        print("Wrong input type! B should be float(or int), T should be int, and thin should be
 int")
        return False
    if(T<0): #check if the input T is correct
        print("Wrong input! T should larger than 0")
        return False
    if(B<0): #check if the input B is correct
        print("Wrong input! B should larger than 0")
        return False
    output=np.zeros((T,2)) #create the output matrix T*2
    x=1
    y=1
    # Set the start value of x and y
    for i in range(T):
        for j in range(thin):
            u=random.uniform(0,1)
            x=-1/y*math.log(1-u*(1-math.exp(-y*B)))
            #u has to be sampled each time
            u=random.uniform(0,1)
            y=-1/x*math.log(1-u*(1-math.exp(-x*B)))
            # using Gibbs sampling to get the samples we need
        output[i,0]=x
        output[i,1]=y
        # save the value of x and y for each loop
    return output
```

Test for incorrect inputs:

In [3]:

```
gibbs(B='a',T=500)
gibbs(B=5.5,T=-100)
gibbs(B=-5,T=100)
```

```
Wrong input type! B should be float(or int), T should be int, and thin should be i
nt
Wrong input! T should larger than 0
Wrong input! B should larger than 0
```

Out[3]:

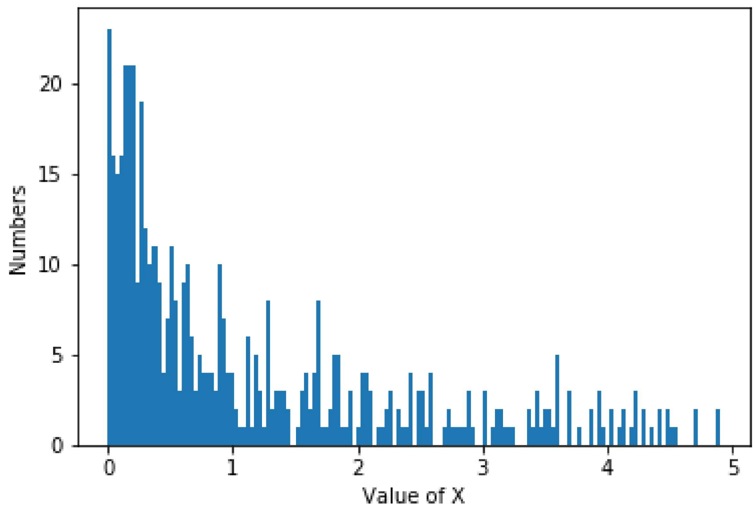
```
False
```

2.

According to the problem, let's set B=5 and T=500, 5000, 50000 seperately, and we use plt functions to plot the histogram of the values of X.

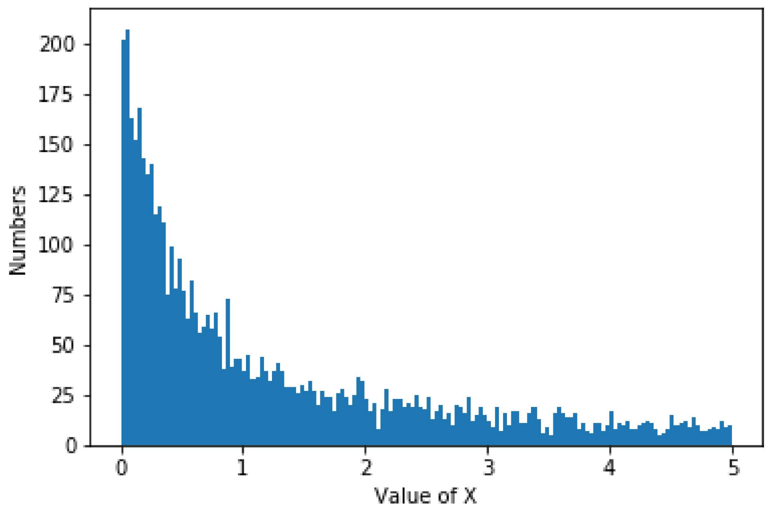
In [4]:

```
#get the sample of different sample size and plot the historgram of the values of x
output1=gibbs(B=5.0,T=500)
plt.hist(output1[:,0],150)
plt.xlabel("Value of X")
plt.ylabel("Numbers")
plt.show()
```



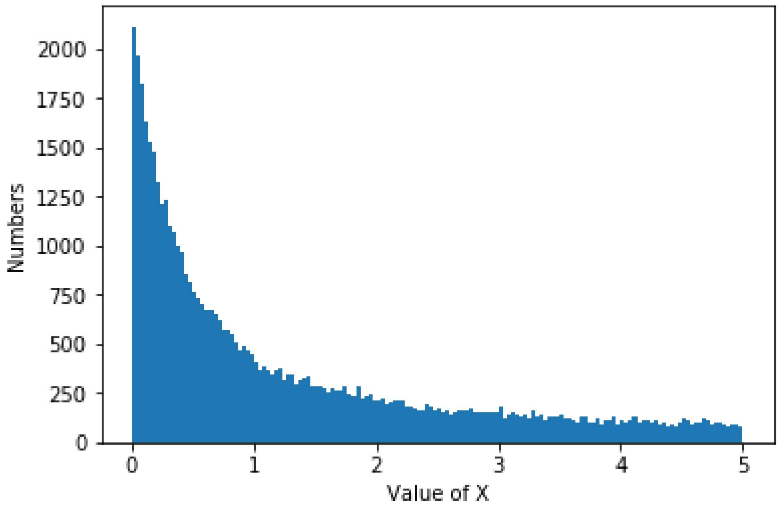
In [5]:

```
output2=gibbs(B=5.0,T=5000)
plt.hist(output2[:,0],150)
plt.xlabel("Value of X")
plt.ylabel("Numbers")
plt.show()
```



In [6]:

```
output3=gibbs(B=5.0,T=50000)
plt.hist(output3[:,0],150)
plt.xlabel("Value of X")
plt.ylabel("Numbers")
plt.show()
```



We can see that with the increase of sample size, the histogram of values of x will be more smooth and similar to the exponential distribution.

3.

According to the sampler, we know that
$$E_{p(x)}[X]=\int_0^B\int_0^B xf(x,y)dxdy=\int_0^B\int_0^B xf(x,y)dydx=\int_0^B xf_X(x)dx$$
Because in the sampler we get discrete values, so according to the Method of moments (or the Law of Large Numbers):
$$E_{p(x)}(X)=\bar{x}=\frac{1}{T}\sum_{i=1}^{T}x_i$$
So we can calculate the estimate of the expection of X with different sample sizes by calculate their means.

In [7]:

```
# compute the mean value of output matrix of different sample sizes, and according to the method
 of moments,
#they are the estimates of the expectation of X
EX1=np.mean(output1[:,0])
EX2=np.mean(output2[:,0])
EX3=np.mean(output3[:,0])
```

So we can see that the estimates of the expectation of X by using the 500, 5000, and 50000 samples from the samplers are

In [8]:

```
print(EX1,",",EX2,"and",EX3)
```

```
1.16141951207 , 1.26744417498 and 1.26260419724
```

According to the x and y's conditional distribution, we should think that their marginal distribution should be symmetric and probably alike. Let's use the cumfreq function in the scipy package to check whether they have similar cdf.

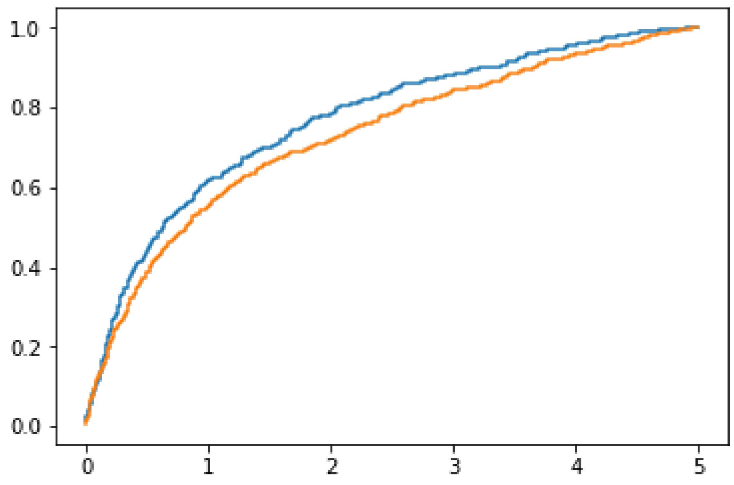
In [9]:

```
import scipy
from scipy import stats
from scipy.stats import cumfreq,ks_2samp

x=np.arange(0,5,.01)
```

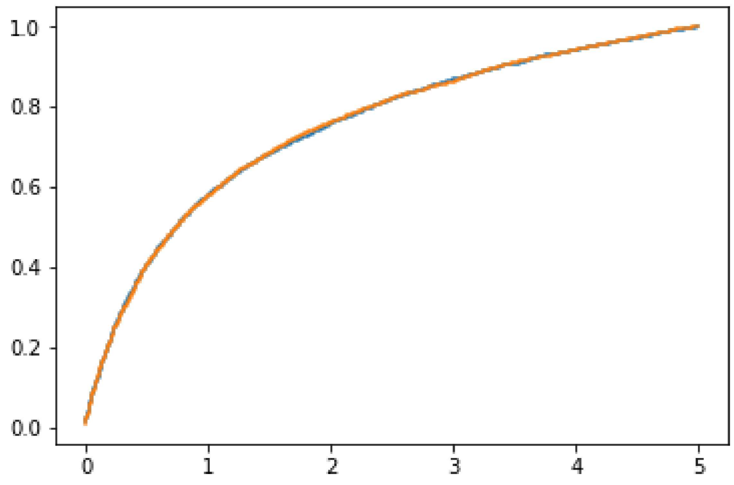
In [10]:

```
out1=cumfreq(output1[:,0],numbins=len(x),defaultreallimits=[min(x),max(x)]).cumcount
out1/=max(out1)
plt.step(x,out1)
out2=cumfreq(output1[:,1],numbins=len(x),defaultreallimits=[min(x),max(x)]).cumcount
out2/=max(out2)
plt.step(x,out2)
plt.show()
```



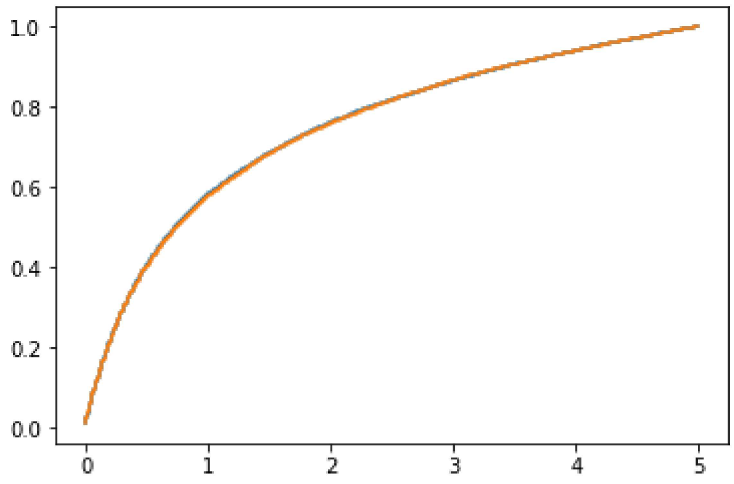
In [11]:

```
out1=cumfreq(output2[:,0],numbins=len(x),defaultreallimits=[min(x),max(x)]).cumcount
out1/=max(out1)
plt.step(x,out1)
out2=cumfreq(output2[:,1],numbins=len(x),defaultreallimits=[min(x),max(x)]).cumcount
out2/=max(out2)
plt.step(x,out2)
plt.show()
```



In [12]:

```
out1=cumfreq(output3[:,0],numbins=len(x),defaultreallimits=[min(x),max(x)]).cumcount
out1/=max(out1)
plt.step(x,out1)
out2=cumfreq(output3[:,1],numbins=len(x),defaultreallimits=[min(x),max(x)]).cumcount
out2/=max(out2)
plt.step(x,out2)
plt.show()
```



This time we can finally see that the cdf of x and y are similar. So the code may be finally right.

Also, we can use the ks_2samp() function to check whether x and y's distribution are similar.

In [14]:

```
ks_2samp(output1[:,0],output1[:,1])
```

Out[14]:

```
Ks_2sampResult(statistic=0.080000000000000071, pvalue=0.077420340647895214)
```

In [15]:

```
ks_2samp(output2[:,0],output2[:,1])
```

Out[15]:

```
Ks_2sampResult(statistic=0.014199999999999999, pvalue=0.69162117217812036)
```

In [16]:

```
ks_2samp(output3[:,0],output3[:,1])
```

Out[16]:

```
Ks_2sampResult(statistic=0.0056399999999999784, pvalue=0.40324893559989317)
```

We can see that with different values of T, the p-values are all larger than 0.05, so we can conclude that x and y's distribution are similar, so our code is correct.

Stat 37810 final project k-means

Algorithm of K-means classifier

(1) Randomly choose n samples from the dataset, and set them as initial points. Here, n is the number of types which we want to group into (In this question, $n = 3$) by k-means classifier.

(2) Calculate the Euclidean distance from each data point to this three points, find the samllest Euclidean distance and divide data point into different types according to the samllest Euclidean distance.

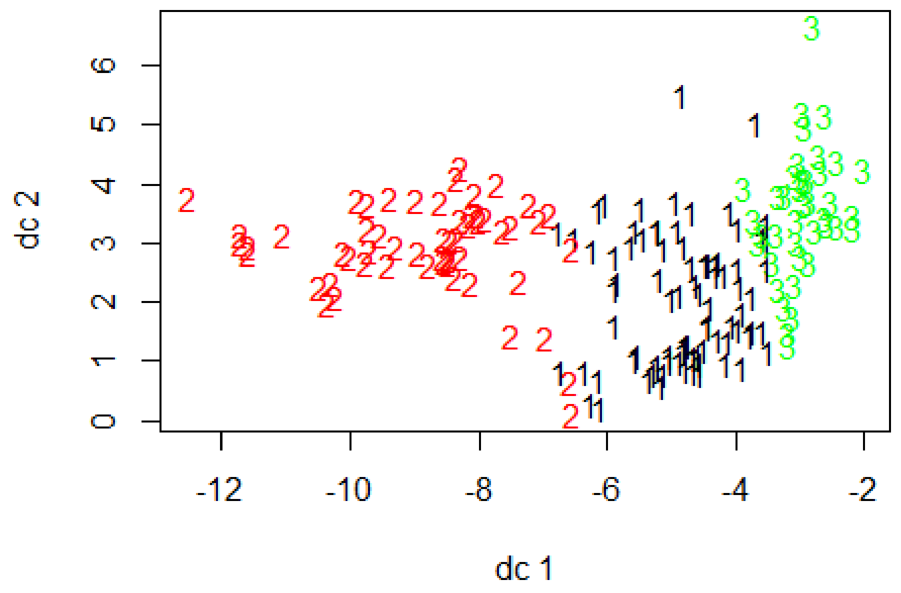
(3) Calculate the mean value of data points of each new types, and set them as new initial points.

(4) Repeat the steps 2 and 3, until the outcome of data points of each type converges.

K-means classifier of data “wine”

```r
data(wine, package="rattle")
library('fpc')
data.train = wine[,-1]
set.seed(10)
kmean_cluster=function(data,kinds){
  start=sample(nrow(data),kinds)
  start_value=data[start,]
  distances=matrix(0,nrow=nrow(data),ncol=kinds+1)
  index=matrix(1,nrow=nrow(data),1)
  while(all(distances[,kinds+1]!=index)){
    index=distances[,kinds+1]
    for(j in 1:nrow(data)){
      for(k in 1:kinds){
        distances[j,k]=sum((data[j,]-start_value[k,])^2)
      }
    }
    distances[,kinds+1]=apply(distances[,1:kinds],1,which.min)
    for(k in 1:kinds){
      start_value[k,]=apply(data[which(distances[,kinds+1]==k),],2,mean)
    }
  }
  return(list(distances[,kinds+1],start_value))
}

fit.km=kmean_cluster(data.train,3)[1]
plotcluster(data.train, unlist(fit.km))
```



Then, we calculate the accuracy of the K-means classifier. The specific procedures are as follows:

(1) Calculate the number of matched data between the three types of original data “wine” and the three types of the results of K-means cluster.

(2) Find the maximum matching number of each type of wine.

(3) Calculate the total matching number and accuracy.

```
fit=unlist(fit.km)
test1=matrix(0,3,3)
for(i in 1:3){
  for(j in 1:3){
    test1[i,j]=sum(wine[which(fit==i),1]==j)/sum(fit==i)
  }
}
test1

##            [,1]       [,2]       [,3]
## [1,] 0.1097561 0.39024390 0.50000000
## [2,] 0.8928571 0.07142857 0.03571429
## [3,] 0.0000000 0.87500000 0.12500000
```

According to the outcome, we can see the wine types 1,2,3 are corresponding to k-means cluster types 2,3,1 respectively. Then, we can calculate the match rate

```r
a1=sum(fit[which(wine[,1]==1)]==2)
a2=sum(fit[which(wine[,1]==2)]==3)
a3=sum(fit[which(wine[,1]==3)]==1)
accuracy=(a1+a2+a3)/nrow(wine)
```

The number of original type 1 in dataset wine is 59.

The number of original type 2 in dataset wine is 71.

The number of original type 3 in dataset wine is 48.

The number of K-means cluster successfully matches original type 1 in dataset wine is 50.

The number of K-means cluster successfully matches original type 2 in dataset wine is 35.

The number of K-means cluster successfully matches original type 3 in dataset wine is 41.

The matched rate of original type 1 in wine is 0.8474576.

The matched rate of original type 2 in wine is 0.4929577.

The matched rate of original type 3 in wine is 0.8541667.

The total accuracy of K-means cluster is 0.7078652.

K-means cluster analysis 1

(1) According to the plot, one can observe that the majority of the samples are separated into three different clusters. However, some samples which are separated into different clusters are very close to each other, and some of them are even enclosed by samples in other clusters. As a result, the outcome of clusters make sense to some degree but maybe not quite well.

(2) It can be seen from our algorithm that the total accuracy is 70.78%, among which the match rates of types 1 and 3 of wine are very high, but the type 2 of wine does not match very well. To sum up, most of the data are matched well, but there are also some data that were not matched quite well.

K-means classifier of data “scale(wine)”

Function of scale()

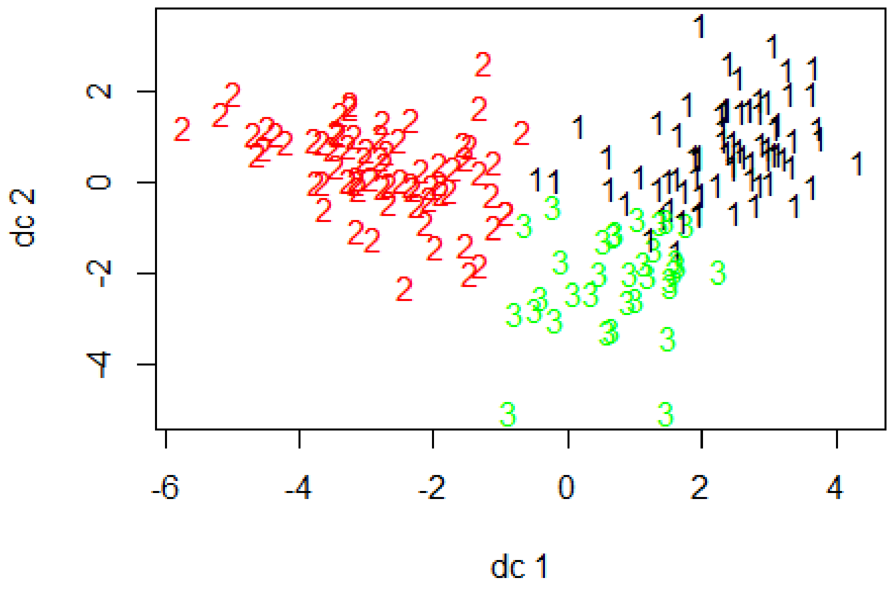
Scale command can centralize and standardize an array. The numeric centering and scalings are returned as attributes: “scaled:center” and “scaled:scale”. Towards each number x in the array, scale command change it into $x_{scale} = \frac{x - scaled:cente}{scaled:scale}$.

To be special, scaled:center is the mean value of the array, which is used to make the array centralization. scaled:scale is the sample standard deviation of the centralized array.

```r
data(wine, package="rattle")
library('fpc')
data.train = scale(wine[,-1])
```

```
set.seed(10)
kmean_cluster=function(data,kinds){
  start=sample(nrow(data),kinds)
  start_value=data[start,]
  distances=matrix(0,nrow=nrow(data),ncol=kinds+1)
  index=matrix(1,nrow=nrow(data),1)
  while(all(distances[,kinds+1]!=index)){
    index=distances[,kinds+1]
    for(j in 1:nrow(data)){
      for(k in 1:kinds){
        distances[j,k]=sum((data[j,]-start_value[k,])^2)
      }
    }
    distances[,kinds+1]=apply(distances[,1:kinds],1,which.min)
    for(k in 1:kinds){
      start_value[k,]=apply(data[which(distances[,kinds+1]==k),],2,mean)
    }
  }
  return(list(distances[,kinds+1],start_value))
}

fit.km=kmean_cluster(data.train,3)[1]
plotcluster(data.train, unlist(fit.km))
```



```
fit=unlist(fit.km)
test1=matrix(0,3,3)
for(i in 1:3){
```

```r
  for(j in 1:3){
    test1[i,j]=sum(wine[which(fit==i),1]==j)/sum(fit==i)
  }
}
test1
```

```
##            [,1]      [,2]       [,3]
## [1,] 0.0000000 0.3611111 0.63888889
## [2,] 0.8194444 0.1805556 0.00000000
## [3,] 0.0000000 0.9411765 0.05882353
```

According to the outcome, we can see the wine types 1,2,3 are corresponding to k-means cluster types 2,3,1 respectively. Then, we can calculate the match rate

```r
a1=sum(fit[which(wine[,1]==1)]==2)
a2=sum(fit[which(wine[,1]==2)]==3)
a3=sum(fit[which(wine[,1]==3)]==1)
accuracy=(a1+a2+a3)/nrow(wine)
```

The number of original type 1 in dataset wine is 59.

The number of original type 2 in dataset wine is 71.

The number of original type 3 in dataset wine is 48.

The number of K-means cluster successfully matches original type 1 in dataset wine is 59.

The number of K-means cluster successfully matches original type 2 in dataset wine is 32.

The number of K-means cluster successfully matches original type 3 in dataset wine is 46.

The matched rate of original type 1 in wine is 1.

The matched rate of original type 2 in wine is 0.4507042.

The matched rate of original type 3 in wine is 0.9583333.

The total accuracy of K-means cluster is 0.7696629.

K-means cluster analysis 2

(1) According to the plot, we can see that the majority of the samples are separated into three different clusters. However, there still exist some samples which are enclosed by samples in other clusters. In other words, a minority of might be classified into the other clusters. As a result, the outcome of clusters make sense to some degree but still not quite well.

(2) It can be seen from our algorithm that the total accuracy is 76.96%, which is higher than the result of K-means cluster based on the original dataset last time. It is worth mentioning that the match rate of types 1 and 3 of wine is quite high, which is more than 95%, but the match rate of type 2 is still not very well. To sum up, the data are clustered better than before.

How scale() command affects the results

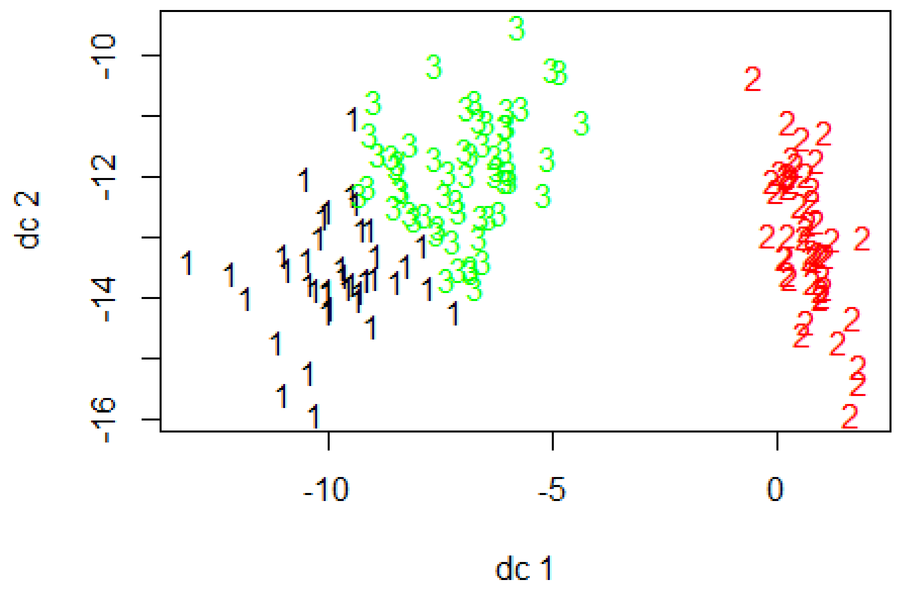
(1)Through using the scale() command towards to original dataset, the accuracy rate of K-meas cluster increases in this experiment.

(2) However, whether the scale() command can always improve the result of cluster still need to be tested. In fact, our group have tried plenty of initial points of K-means, and we found that in some cases, scale() command would decrease the final cluster accuracy. The reason might be that scale() command makes the original data become much closer to each other, which would potentially make it more difficult to cluster. As a result, we cannot make the conclusion that scale() command can always improve the accuracy.

K-means classifier of data "iris"

```
data(iris)
library('fpc')
data.train = iris[,-5]
set.seed(10)
kmean_cluster=function(data,kinds){
  start=sample(nrow(data),kinds)
  start_value=data[start,]
  distances=matrix(0,nrow=nrow(data),ncol=kinds+1)
  index=matrix(1,nrow=nrow(data),1)
  while(all(distances[,kinds+1]!=index)){
    index=distances[,kinds+1]
    for(j in 1:nrow(data)){
      for(k in 1:kinds){
        distances[j,k]=sum((data[j,]-start_value[k,])^2)
      }
    }
    distances[,kinds+1]=apply(distances[,1:kinds],1,which.min)
    for(k in 1:kinds){
      start_value[k,]=apply(data[which(distances[,kinds+1]==k),],2,mean)
    }
  }
  return(list(distances[,kinds+1],start_value))
}

fit.km=kmean_cluster(data.train,3)[1]
plotcluster(data.train, unlist(fit.km))
```



```
fit=unlist(fit.km)
test1=matrix(0,3,3)
types=c("setosa","versicolor","virginica")
for(i in 1:3){
  for(j in 1:3){
    test1[i,j]=sum(iris[which(fit==i),5]==types[j])/sum(fit==i)
  }
}
test1

##      [,1]      [,2]      [,3]
## [1,]    0 0.1000000 0.9000000
## [2,]    1 0.0000000 0.0000000
## [3,]    0 0.7666667 0.2333333
```

According to the outcome, we can see the iris types “setosa”,“versicolor”,“virginica” are corresponding to k-means cluster types 2,3,1 respectively. Then, we can calculate the match rate

```
a1=sum(fit[which(iris[,5]=="setosa")]==2)
a2=sum(fit[which(iris[,5]=="versicolor")]==3)
a3=sum(fit[which(iris[,5]=="virginica")]==1)
accuracy=(a1+a2+a3)/nrow(iris)
```

The number of original type ‘setosa’ in dataset iris is 50.

The number of original type ‘versicolor’ in dataset iris is 50.

The number of original type ‘virginica’ in dataset iris is 50.

The number of K-means cluster successfully matches original type ‘setosa’ in dataset iris is 50.

The number of K-means cluster successfully matches original type ‘versicolor’ in dataset iris is 46.

The number of K-means cluster successfully matches original type ‘virginica’ in dataset iris is 36.

The matched rate of original type ‘setosa’ in iris is 1.

The matched rate of original type ‘versicolor’ in iris is 0.92.

The matched rate of original type ‘virginica’ in iris is 0.72.

The total accuracy of K-means cluster is 0.88.

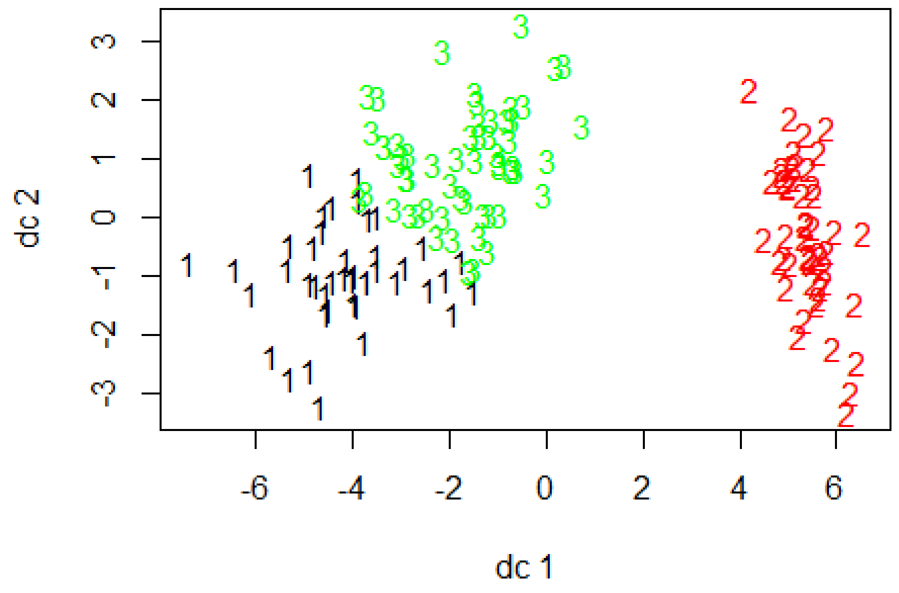
K-means cluster analysis 3

(1) According to the plot, one can observe that the majority of the samples are separated into three different clusters. In addition, the cluster 2 is classified quite well. Although some points are very close to points in other clusters, the result of clusters is quite well as a whole.

(2) It can be seen from our algorithm that the total accuracy is 88%, among which the match rates of types 1 and 2 of wine are very high. It should be pointed out that the match rate of type ‘setosa’ is 100%, which means it classified quite well.

```r
data(iris)
library('fpc')
data.train = scale(iris[,-5])
set.seed(10)
kmean_cluster=function(data,kinds){
  start=sample(nrow(data),kinds)
  start_value=data[start,]
  distances=matrix(0,nrow=nrow(data),ncol=kinds+1)
  index=matrix(1,nrow=nrow(data),1)
  while(all(distances[,kinds+1]!=index)){
    index=distances[,kinds+1]
    for(j in 1:nrow(data)){
      for(k in 1:kinds){
        distances[j,k]=sum((data[j,]-start_value[k,])^2)
      }
    }
    distances[,kinds+1]=apply(distances[,1:kinds],1,which.min)
    for(k in 1:kinds){
      start_value[k,]=apply(data[which(distances[,kinds+1]==k),],2,mean)
    }
  }
  return(list(distances[,kinds+1],start_value))
}

fit.km=kmean_cluster(data.train,3)[1]
plotcluster(data.train, unlist(fit.km))
```



```
fit=unlist(fit.km)
test1=matrix(0,3,3)
types=c("setosa","versicolor","virginica")
for(i in 1:3){
  for(j in 1:3){
    test1[i,j]=sum(iris[which(fit==i),5]==types[j])/sum(fit==i)
  }
}
test1

##      [,1]      [,2]      [,3]
## [1,]    0 0.1666667 0.8333333
## [2,]    1 0.0000000 0.0000000
## [3,]    0 0.7413793 0.2586207
```

According to the outcome, we can see the iris types “setosa”,“versicolor”,“virginica” are corresponding to k-means cluster types 2,3,1 respectively. Then, we can calculate the match rate

```
a1=sum(fit[which(iris[,5]=="setosa")]==2)
a2=sum(fit[which(iris[,5]=="versicolor")]==3)
```

```
a3=sum(fit[which(iris[,5]=="virginica")]==1)
accuracy=(a1+a2+a3)/nrow(iris)
```

The number of original type ‘setosa’ in dataset iris is 50.

The number of original type ‘versicolor’ in dataset iris is 50.

The number of original type ‘virginica’ in dataset iris is 50.

The number of K-means cluster successfully matches original type ‘setosa’ in dataset iris is 50.

The number of K-means cluster successfully matches original type ‘versicolor’ in dataset iris is 43.

The number of K-means cluster successfully matches original type ‘virginica’ in dataset iris is 35.

The matched rate of original type ‘setosa’ in iris is 1.

The matched rate of original type ‘versicolor’ in iris is 0.86.

The matched rate of original type ‘virginica’ in iris is 0.7.

The total accuracy of K-means cluster is 0.8533333.

K-means cluster analysis 4

(1) According to the plot, we can see that the majority of the samples are separated into three different clusters. However, compared with the plot of the cluster result of original data, we can see that type 1 and 3 become much more closer, and some of them are even classified into other clusters, which means that the result of cluster might not be better than last time.

(2) It can be seen from our algorithm that the total accuracy is 85.33%. Although this accuracy is acceptable, it is lower than the result of K-means cluster based on the original dataset last time.

How scale() command affects the results

(1) Through using the scale() command towards to original dataset, the accuracy rate of K-meas cluster decreases in this experiment.

(2) As we have discussed when analyzing the “wine” dataset, scale() command cannot improve the accuracy all the time. To verify this, our group also carried out quite a lot of experiments by using the “iris” dataset, and we found that scale() command could both increase or decrease the final cluster accuracy.

(3) On one hand, the result of accuracy might be influenced by the initial points of K-means, which is also tested by us through many experiments. On the other hand, as we discussed before, scale() command makes the original data become much closer to each other, which would make it difficult to cluster. In fact, the original dataset would

affect the result of transformed data by using scale(), which would further influence the accuracy.