Metropolis-Hastings part

Fangzhou Yang, Kim Ting Li, Adrian Diaz

11/3/2017

MAIN FUNCTION

Here we write a Metropolis-Hastings algorithm which is sampled from a Beta distribution. we should use the Detail Balance equation here:

$$\pi(\phi)T(\phi \to \phi') = \pi(\phi')T(\phi' \to \phi)$$

, which also can be written as

$$\frac{T(\phi \to \phi')}{T(\phi' \to \phi)} = \frac{\pi(\phi')}{\pi(\phi)}$$

- Firstly, we choose a start value: ϕ at random.
- Then, we separate the transition in two sub-steps:the proposal distribution ($Q(\phi \to \phi')$) and the acceptance probability ($A(\phi \to \phi')$).

  $$T(\phi \to \phi') = Q(\phi \to \phi')A(\phi \to \phi')$$

  in the part below, we write the proposal function of the form

  $$\phi_{prop}|\phi_{old} \sim Beta(c\phi_{old}, c(1-\phi_{old}))$$

  , which is the probability of proposing a state $\phi'|\phi$.

ANd we create the posterior funtion which is

$$\frac{P(\phi')}{P(\phi_{old})} = \frac{dbeta(\phi', a, b)}{dbeta(\phi_{old}, a, b)}$$

where P follows the Beta distribution, and (a,b) = (6,4).

```
proposalfunction <- function(c, phi_old){
  return (rbeta(1, c*phi_old, c*(1-phi_old)))
}

posterior <- function(c, phi_old,trueA,trueB){
  return(dbeta(proposalfunction(c, phi_old),trueA,trueB)/dbeta(phi_old,trueA,trueB))
}
```

- Next, we writhe the Metropolis-Hastings algorithm which is sampled from a Beta distribution. and set the acceptance probability as:

  $$A(\phi \to \phi') = min(1, \frac{\pi(\phi')Q(\phi' \to \phi)}{\pi(\phi)Q(\phi \to \phi')})$$

  If state is accepted, set the current value to ϕ'; otherwise, set it to the startvalue ϕ.

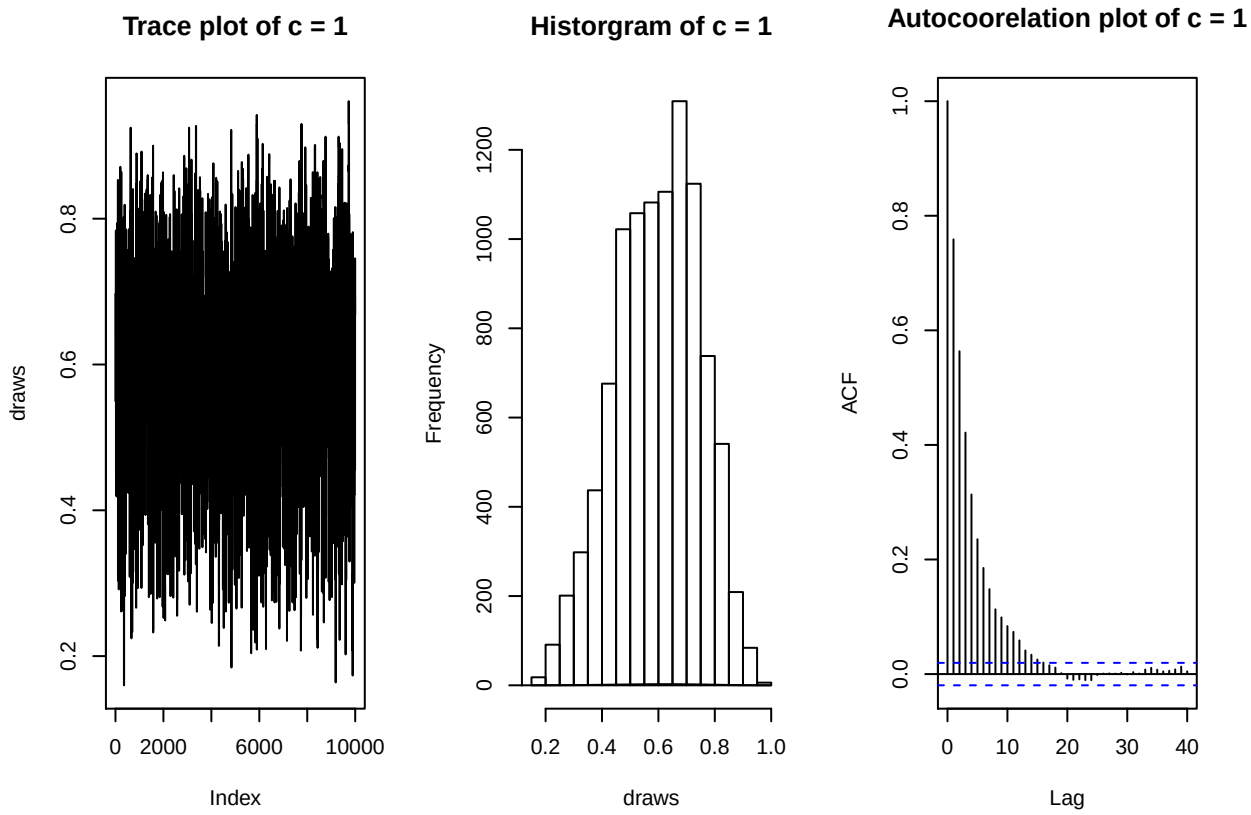
*Final, we repeat this process until generate the required number of samples we need.

```
beta_metropolis <- function(c,startvalue,iterations){
  trueA = 6
  trueB = 4
  chain <- rep(0,iterations)
  chain[1] = startvalue
```

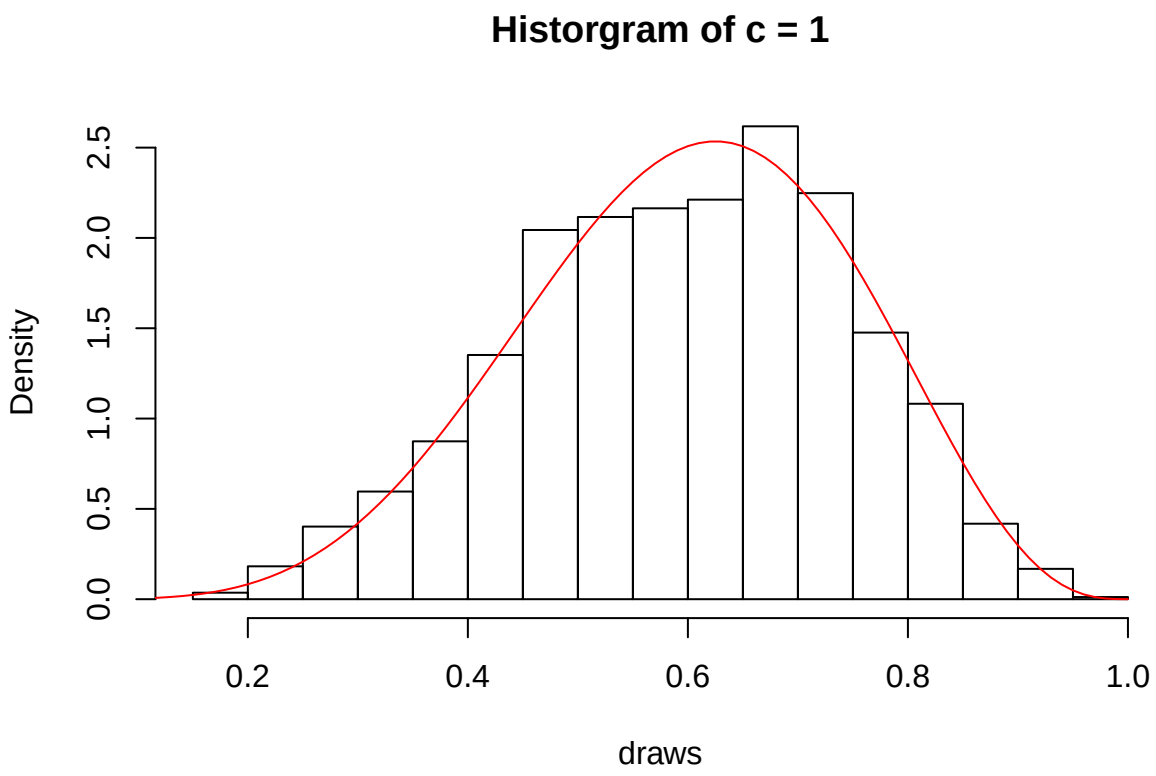
```
  for (i in 1:iterations){
    phi <- proposalfunction(c, startvalue)
    proposal <- dbeta(startvalue, c*phi, c*(1-phi))/dbeta(phi, c*startvalue, c*(1-startvalue))
    post <- dbeta(phi,trueA,trueB)/dbeta(startvalue,trueA,trueB)
    probab <- min(1,post*proposal)

    if(runif(1) < probab){
      startvalue <- phi
      chain[i] <- phi
    } else {
      chain[i]<-startvalue
      }
  }
  return(chain)
}
```

by test, we give a sample with c = 1, iterations = 10000 and start value = random uniform(1):

```
draws <- beta_metropolis(1,runif(1),10000)
par(mfrow = c(1,3))
plot(draws,type = "l",main = "Trace plot of c = 1")
hist(draws, main = "Historgram of c = 1")
x <- seq(0,1,0.01)
lines(x,dbeta(x,6,4))
acf(draws, main = "Autocoorelation plot of c = 1")
```



```r
par(mfrow = c(1,1))
x <- seq(0,1,0.01)
hist(draws, main = "Historgram of c = 1",freq = FALSE)
lines(x,dbeta(x,6,4),col = "red")
```

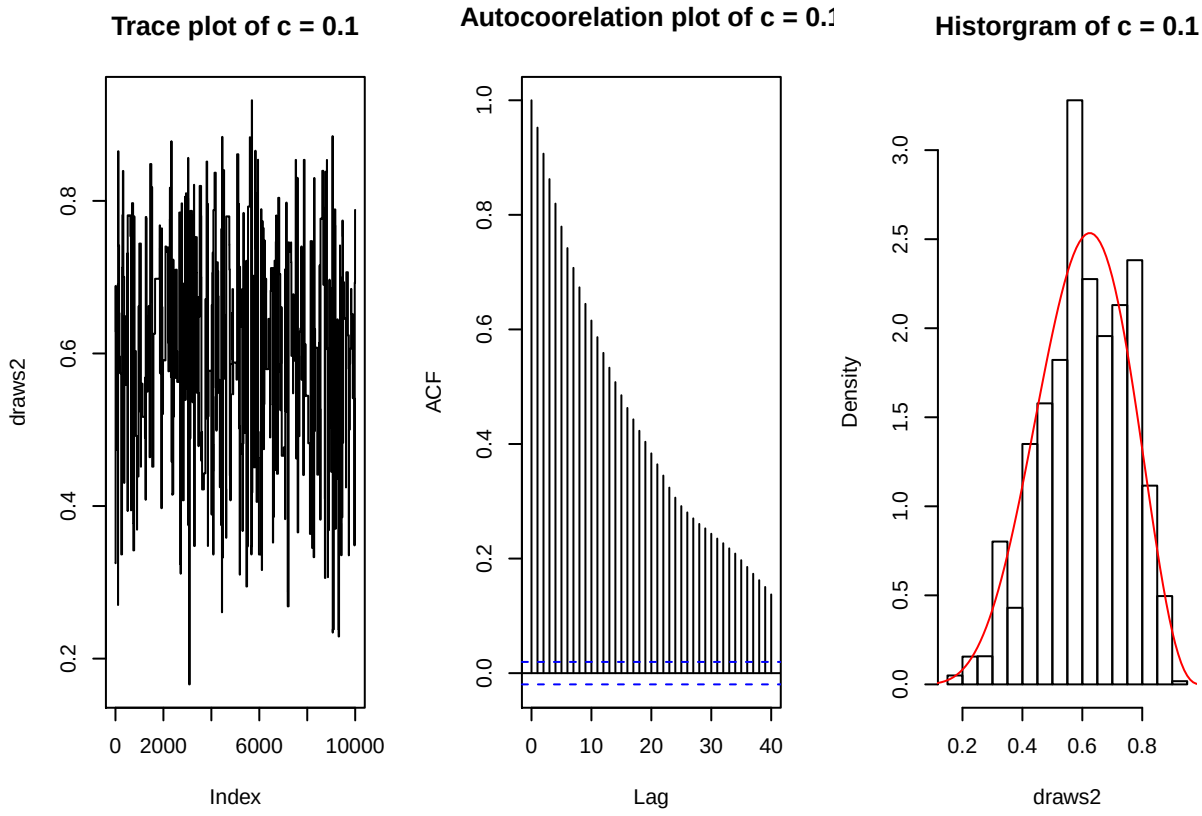



```r
ks.test(jitter(draws,0.0001),rbeta(10000,6,4))
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  jitter(draws, 1e-04) and rbeta(10000, 6, 4)
## D = 0.0241, p-value = 0.006006
## alternative hypothesis: two-sided
```

c = 0.1:

```r
draws2 <- beta_metropolis(0.1,runif(1),10000)
par(mfrow = c(1,3))
plot(draws2,type = "l",main = "Trace plot of c = 0.1")
acf(draws2, main = "Autocoorelation plot of c = 0.1")
hist(draws2, main = "Historgram of c = 0.1",freq = FALSE)
x <- seq(0,1,0.01)
lines(x,dbeta(x,6,4),col = "red")
```

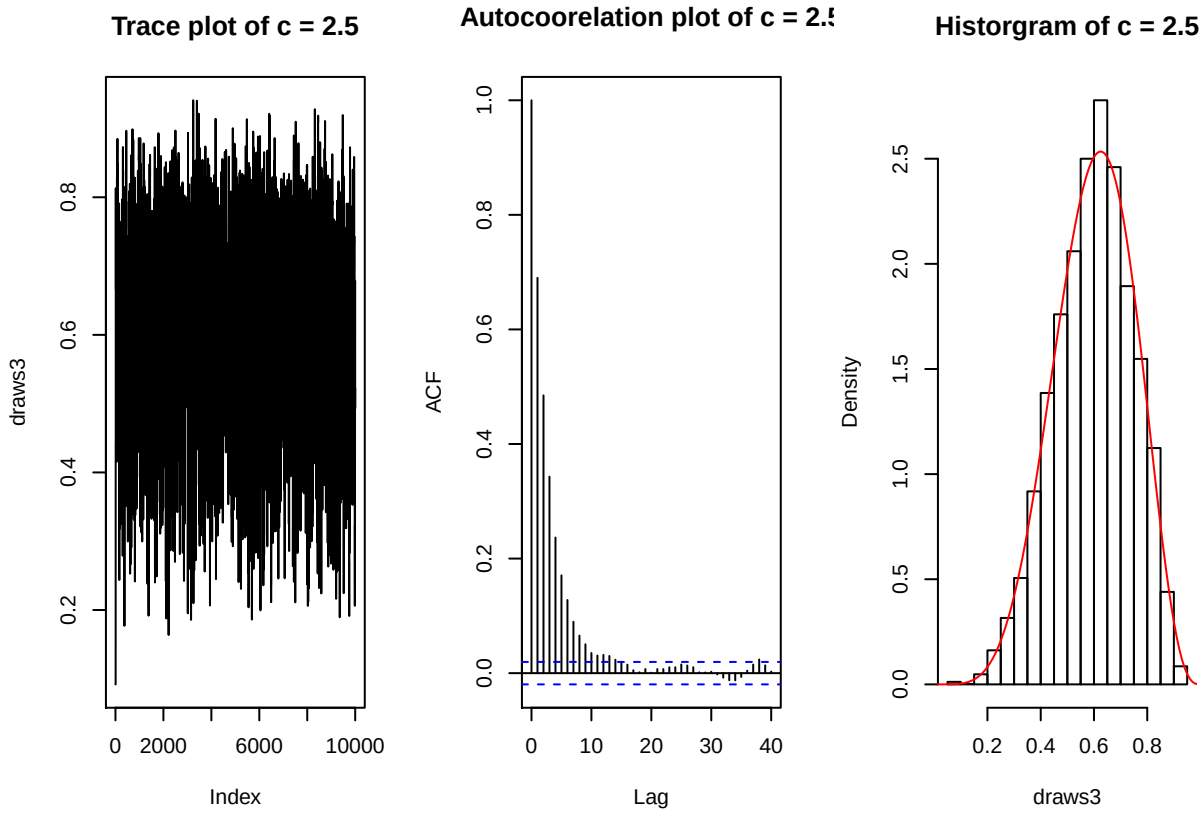



```
ks.test(jitter(draws2,0.0001),rbeta(10000,6,4))
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  jitter(draws2, 1e-04) and rbeta(10000, 6, 4)
## D = 0.0692, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

c = 2.5:

```
draws3 <- beta_metropolis(2.5,runif(1),10000)
par(mfrow = c(1,3))
plot(draws3,type = "l",main = "Trace plot of c = 2.5")
acf(draws3, main = "Autocoorelation plot of c = 2.5")
hist(draws3, main = "Historgram of c = 2.5",freq = FALSE)
x <- seq(0,1,0.01)
lines(x,dbeta(x,6,4),col = "red")
```

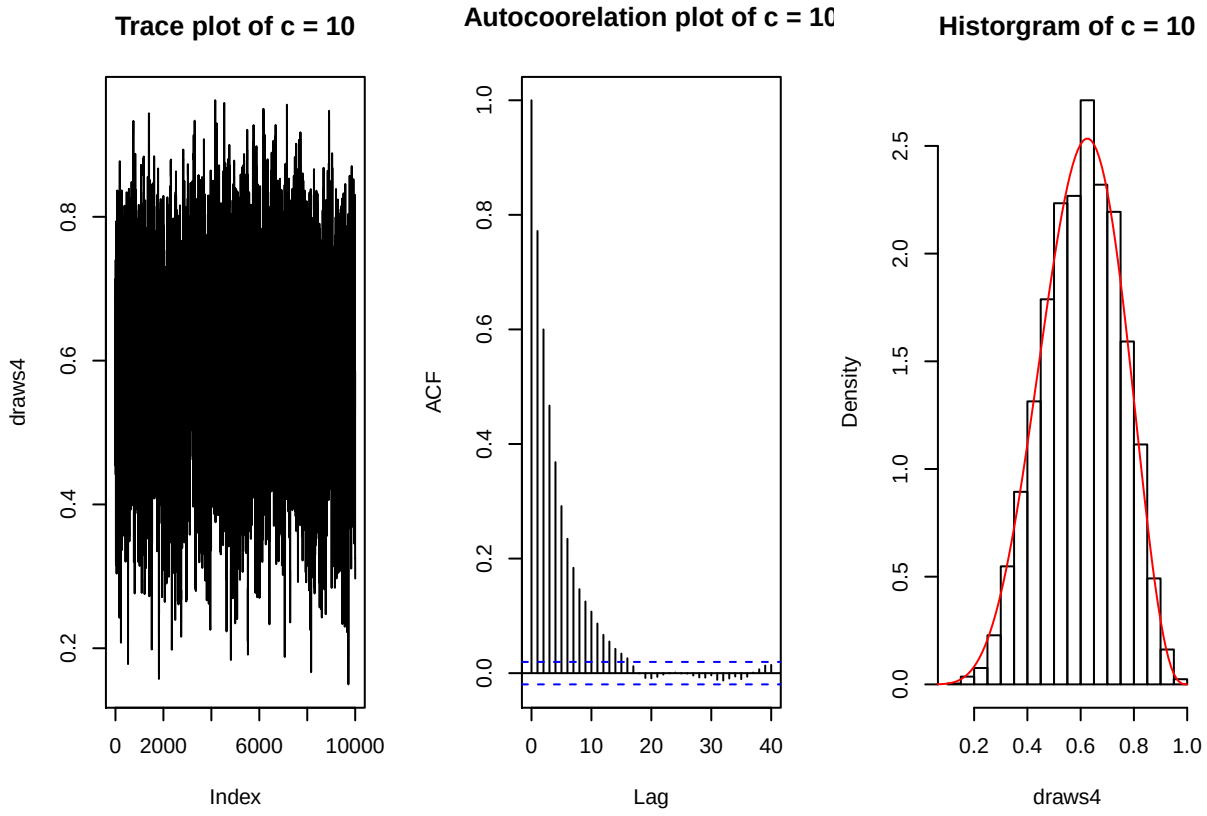



```
ks.test(jitter(draws3,0.0001),rbeta(10000,6,4))
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  jitter(draws3, 1e-04) and rbeta(10000, 6, 4)
## D = 0.0203, p-value = 0.03246
## alternative hypothesis: two-sided
```

c = 10:

```
draws4 <- beta_metropolis(10,runif(1),10000)
par(mfrow = c(1,3))
plot(draws4,type = "l",main = "Trace plot of c = 10")
acf(draws4, main = "Autocoorelation plot of c = 10")
hist(draws4, main = "Historgram of c = 10",freq = FALSE)
x <- seq(0,1,0.01)
lines(x,dbeta(x,6,4),col = "red")
```




```
ks.test(jitter(draws4,0.0001),rbeta(10000,6,4))
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  jitter(draws4, 1e-04) and rbeta(10000, 6, 4)
## D = 0.0159, p-value = 0.1595
## alternative hypothesis: two-sided
```

By re-run the sampler with c = 0.1, c = 2.5 and c = 1, we can see when c is getting larger, it meets the target distribution better and have larger p-value, but when c is too large, the autocorrelation would be worse.

1. Compare their plot, as c becomes larger, the more efficient it meets target distribution.
2. Through the acf plot, we can see that when c = 10, it is better than c= 0.1 but worse than c = 2.5. So we should choose a fitable c which cannot be too large or too small.
3. By comparing their historgram, we can see as c becomes larger, it meets the target distribution more.
4. Through the Kolmogorov¨CSmirnov statistic test, the p-value is getting larger as c gets larger.

So by comparing this values of c, c = 2.5 is most effective at drawing from the target distribution since it could meets the target distribution enough well and has a better autocorrelation plot than c= 10.

K-means

The algorithm for the k-means implementation is the following:

(1) Select n=3 random observations in the data, assign a label for each of one and regard them as the centroid for their respective classification.
(2) Compute the euclidean distance of all other observations to these initial centroids, and assign each to the label corresponding to the closest centroid.
(3) Calculate new centroids for each label based on this initial labeling.
(4) Compute the euclidean distance of all observations to these new centroids, and relabel them to the corresponding label to the closest centroid.
(5) Calculate new centroids again for each label.
(6) Repeat steps (4) and (5) until the labels do not change. The resulting labels are the final labels for each observation.

Note: Compared to the k-means implementation in R taking the same single realization for the random starting centroids (that is, setting the same seed), the only difference between both is that somehow R's k-means always assigns the labels in a very similar way as in the original data, while the above one arrives to the same clusters but with different labels each time.

The following code chunks below show the code for the algorithm:

```
library(rattle.data)
data(wine)
data(iris)
```

```
library(fpc)
```

```
newwine<-as.matrix(wine) #Transform data into numeric (necessary for the function to work)
class(newwine)<-"numeric"
newwine<-data.frame(newwine)
newwine<-scale(newwine)
newiris<-as.matrix(iris)
species<-newiris[,ncol(newiris)] #Assign numeric codes to species
for (i in 1:length(species)) {
  if (species[i]=="setosa") {
    species[i]<-1
  } else if (species[i]=="versicolor") {
    species[i]<-2
  } else {
    species[i]<-3
  }
}
class(species)<-"numeric"
newiris<-newiris[,-ncol(newiris)] #Transform the Iris DB so it is similar to the wine one
newiris<-cbind(species,newiris)
class(newiris)<-"numeric"
```

```
eucdist<-function(x,y) { #Euclidean distance
  x_y2<-(x-y)*(x-y)
  eucdist<-sqrt(sum(x_y2))
  return(eucdist)
}
```

```
truetypes<-function(x) { #Returns a vector with the true types
  trueclass<<-as.matrix(x)
```

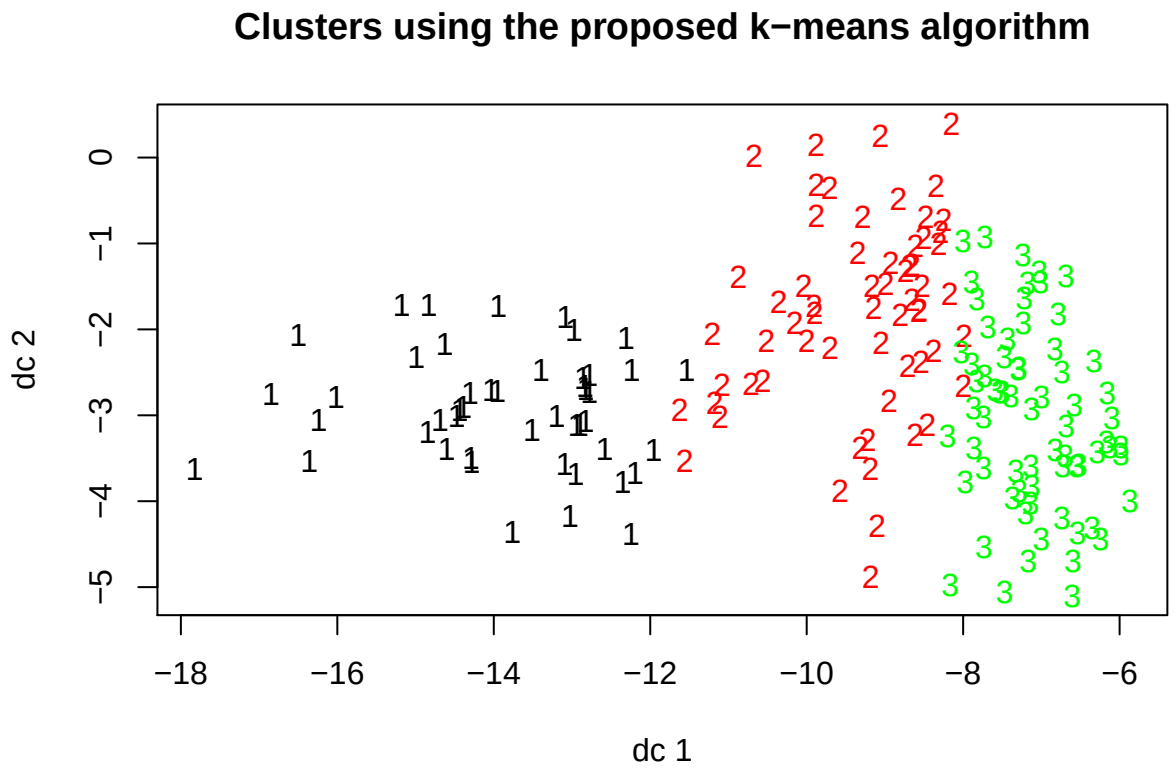
```
  trueclass<<-trueclass[,1]
}
randomcentertype<-function(x,n=3) { #Selects observations as random to act as centroids.
  ddb<<-x
  ddb<<-ddb[,-1]
  randomcentroidtype<<-ddb[sample(nrow(ddb),n),]
  randomcentroidtype<<-t(randomcentroidtype)
  class(randomcentroidtype)<<-"numeric"
  return(randomcentroidtype)
}
centertype<-function(x,currtypes) {
#Returns the centroids, based on the mean, corresponding to given labels
  dd<<-x
  dd<<-dd[,-1]
  dd<<-cbind(currtypes,dd)
  centroidtype<<-aggregate(dd,(by=list(Type=dd[,1])),FUN = function(x) mean(as.numeric(as.character(x))))
  centroidtype<<-centroidtype[,-1:-2]
  centroidtype<<-t(centroidtype)
  class(centroidtype)<<-"numeric"
  return(centroidtype)
}
selectype<-function(x,r1=randomcentertype(x)[,1],r2=randomcentertype(x)[,2],r3=randomcentertype(x)[,3])
#Represents a single iteration, relabels observations to the label corresponding to
#the closest centroid
#If no centroids are provided, the function randomcentertype picks 3 at random.
  bd<<-as.matrix(x)
  bd<<-bd[,-1]
  bd<<-t(bd)
  class(bd)<<-"numeric"
  types<<-c()
  for (i in 1:nrow(x)) {
    if (min(c(eucdist(bd[,i],r1),eucdist(bd[,i],r2),eucdist(bd[,i],r3)))==eucdist(bd[,i],r1)) {
      types<<-c(types,1)
    } else if (min(c(eucdist(bd[,i],r1),eucdist(bd[,i],r2),eucdist(bd[,i],r3)))==eucdist(bd[,i],r2)) {
      types<<-c(types,2)
    } else {
      types<<-c(types,3)
    }
  }
  return(types)
}
kavg<-function(x,n=25,m=3) {
#Actual k-means function. First part is an initialization.
#since the result depends on the starting point, an attempt was made to make it more similar to
#the native R kmeans function which allows the user to pick several random starting points
#and returns the best result. However, there are no details on how does the native R kmeans
#function evaluate which result is the "best" so it was not possible to try to include that
#as an option.
  tipos<<-cbind2(0,selectype(x)) #First iteration using random centroids for the labels.
  trueclass<<-truetypes(x)
  class(trueclass)<<-"numeric"
  i<-2
  while (eucdist(tipos[,i-1],tipos[,i])>0) {
```

```
#While loop which continues iterating and choosing labels until the labels do not change.
#Euclidean distance can be used since the labels are numeric.
    datos<<-as.matrix(x)
    datos<<-datos[,-1]
    datos<<-cbind2(tipos[,ncol(tipos)],datos)
    datos<<-data.frame(datos)
    currentcenter<<-centertype(datos,tipos[,ncol(tipos)])
    tipos<<-cbind2(tipos,selectype(datos,currentcenter[,1],currentcenter[,2],currentcenter[,3]))
    i<-i+1
  }
  finaltype<<-tipos[,ncol(tipos)] #Returns final labels, states how many iterations it took.
  class(finaltype)<<-"numeric"
  return(finaltype)
}
```

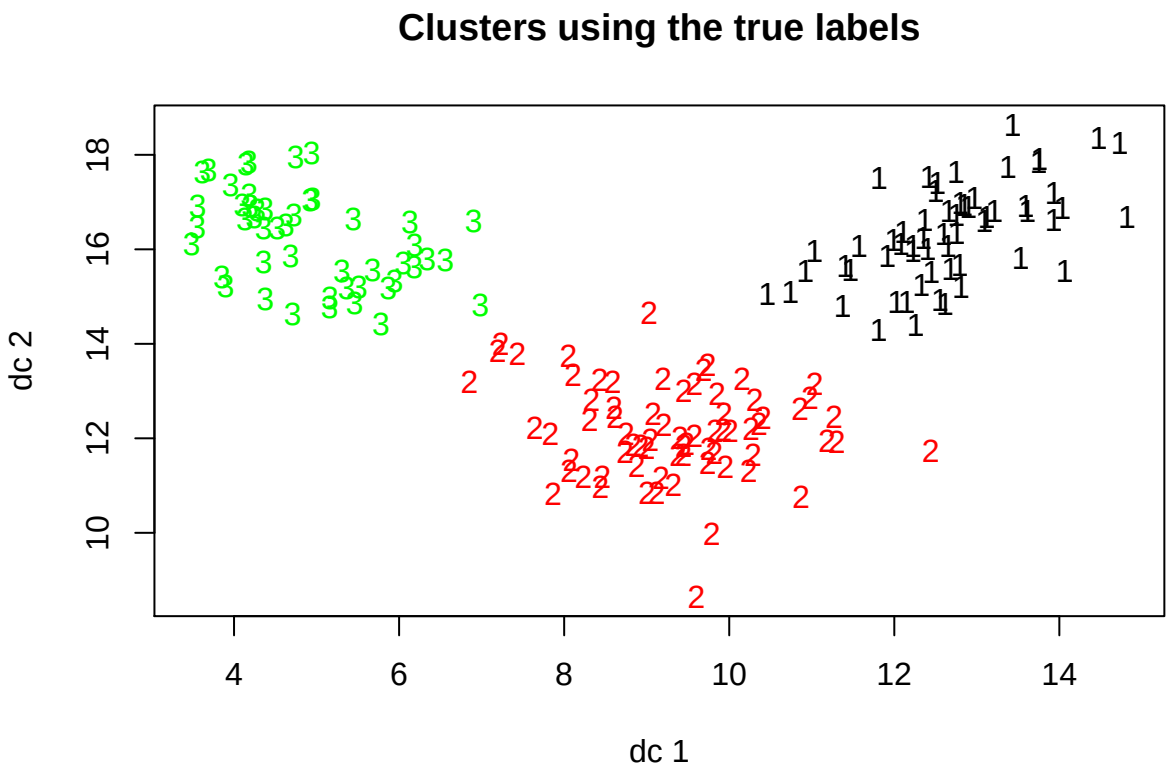
Part (a)

To compare with the true labels, since the algorithm assigns arbitrary labels, it is preferable to simply compare the resulting plots from using the algorithm with that resulting from using the true labels as the input over constructing 2-way tables, as is done in the plots below:

```
plotcluster(wine[,-1],kavg(wine),main="Clusters using the proposed k-means algorithm")
```




```
plotcluster(wine[,-1],truetypes(wine),main="Clusters using the true labels")
```



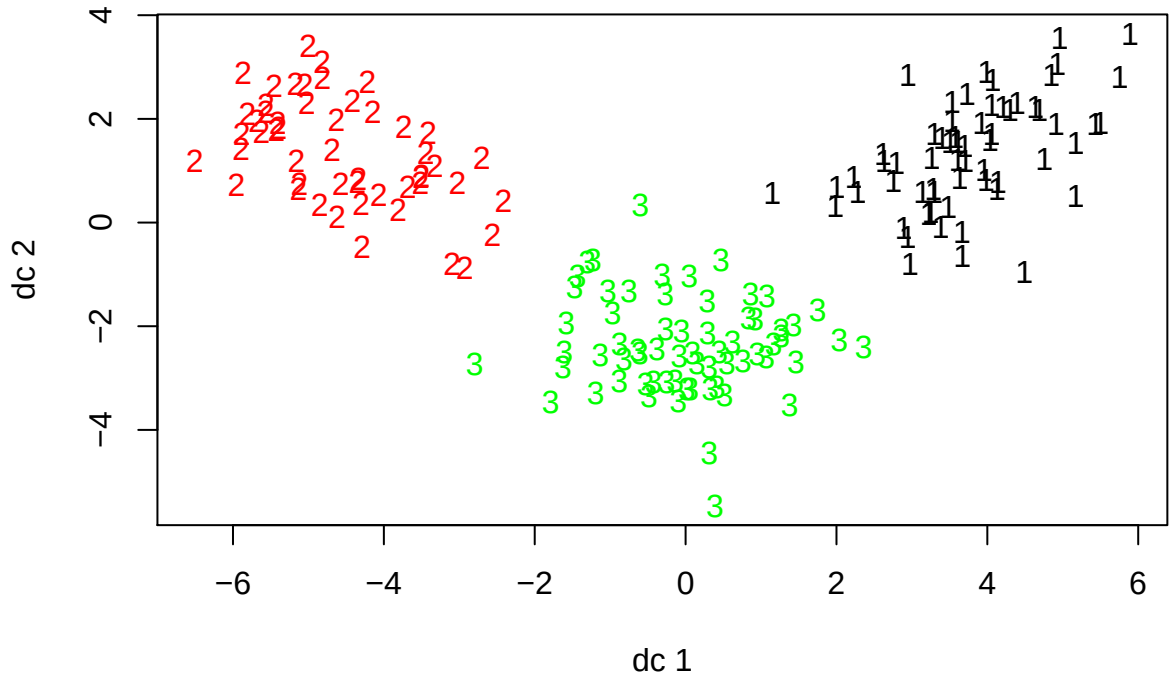
The clusters are not similar to those formed by the real labels, with the resulting clusters showing a higher overlap with each other than in the true ones.

Part (b)

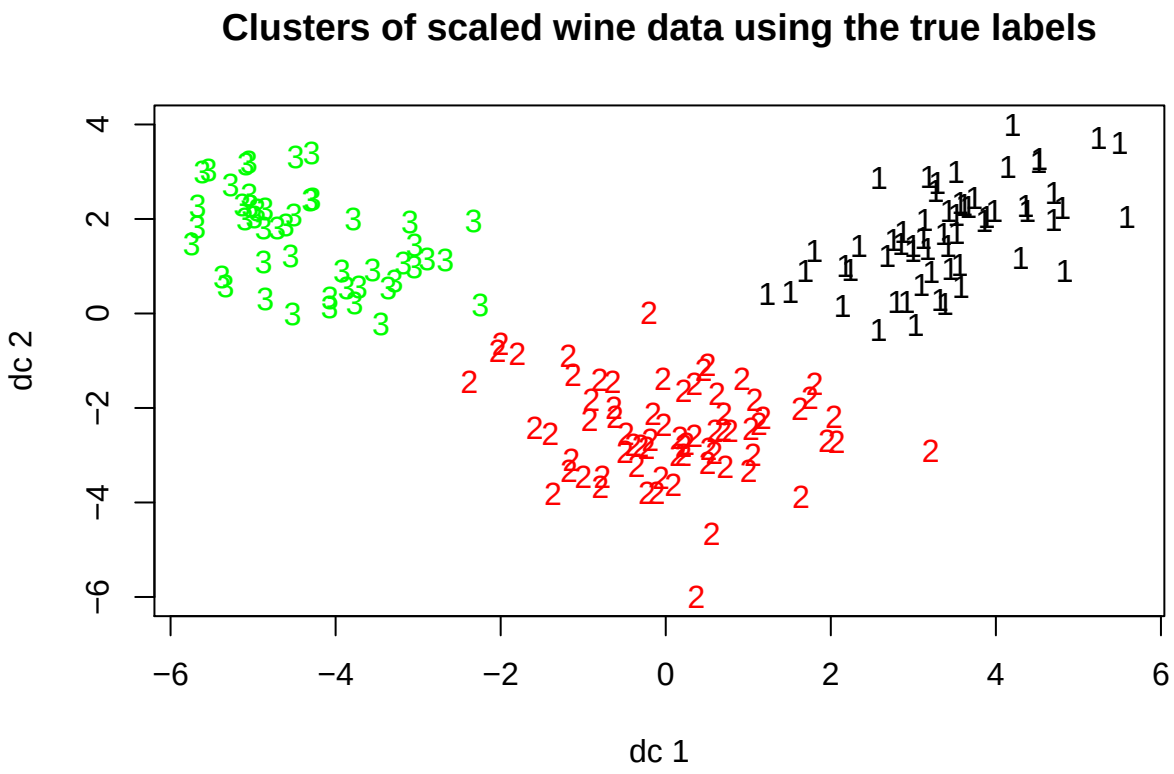
The scale() function scales the data numeric data so it is expressed as deviation from its means divided over the standard deviation for the corresponding variable.

When scaling the data, the resulting clusters are much closer to the clusters formed by the true labels, as shown in the plots below:

```
plotcluster(newwine[,-1],kavg(newwine),main="Clusters of scaled wine data using the proposed k-means al
```



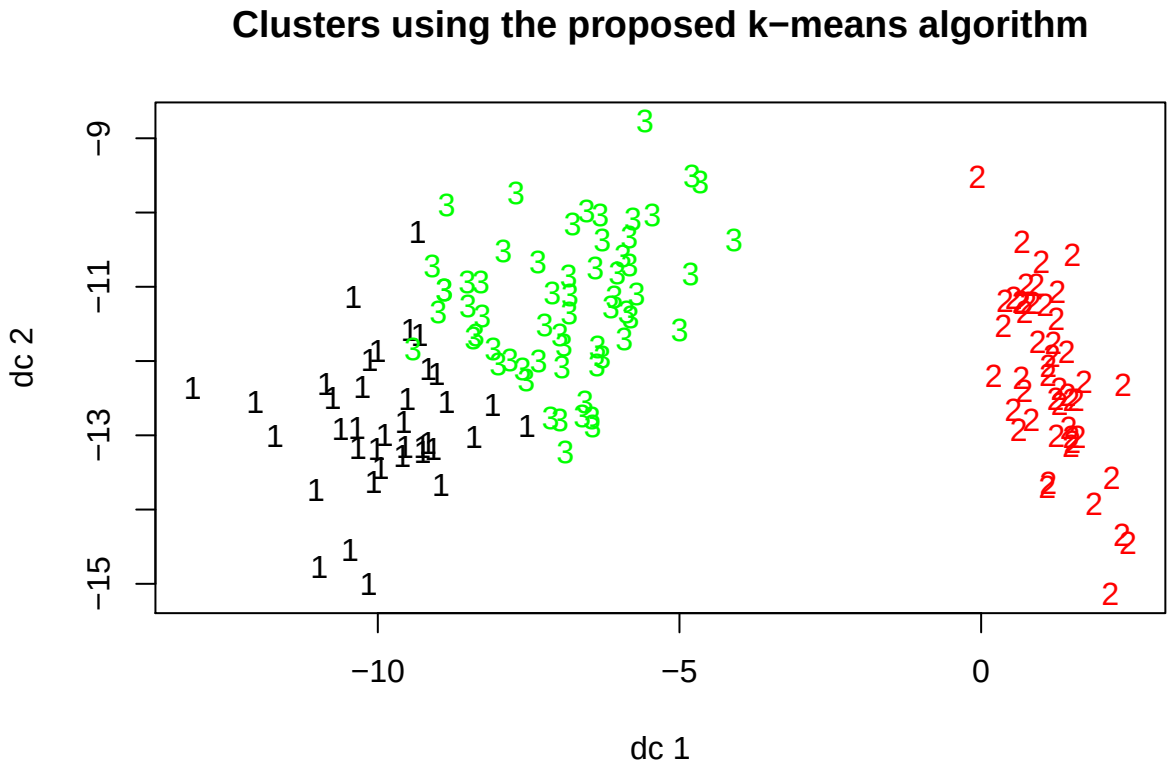

```
plotcluster(newwine[,-1],truetypes(newwine),main="Clusters of scaled wine data using the true labels")
```



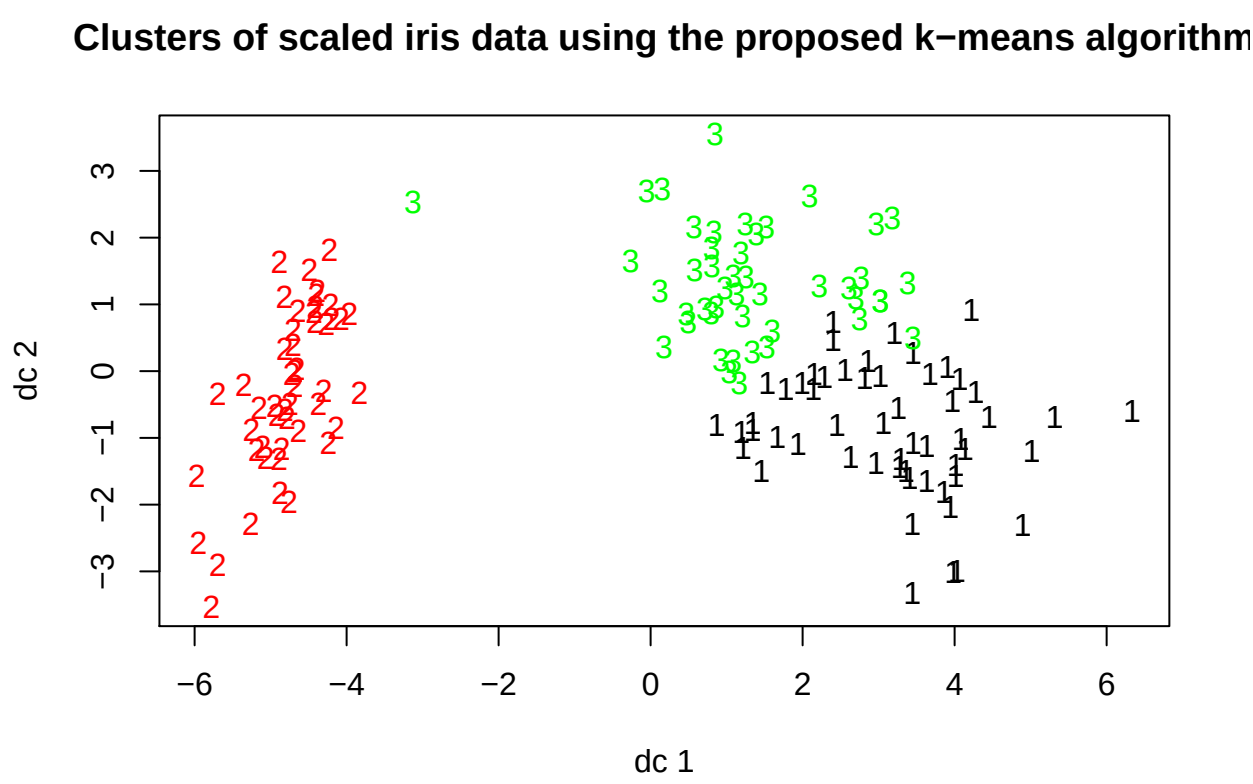
Note that even though the clustering is more similar, the resulting plot can be equivalent to a rotation of that using the true labels. This is because a random starting point is used.

Part (c)

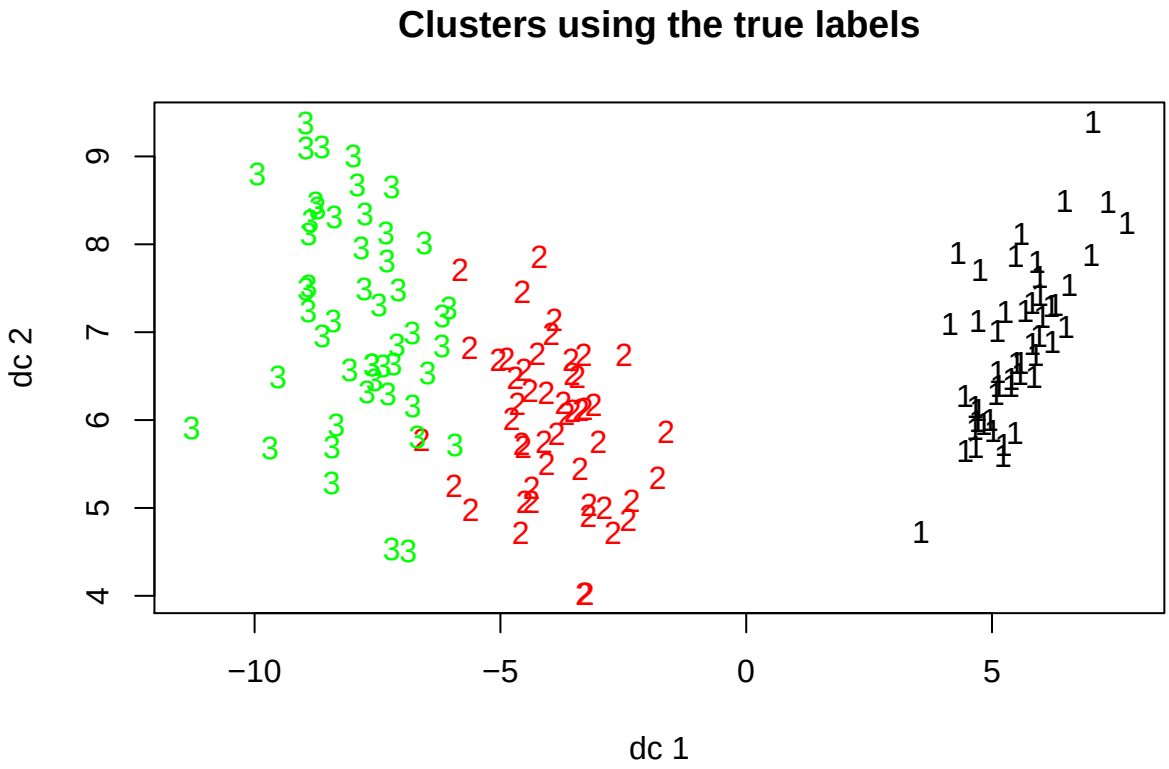
```
plotcluster(newiris[,-1],kavg(newiris),main="Clusters using the proposed k-means algorithm")
```




```
plotcluster(scale(newiris[,-1]),kavg(scale(newiris)),main="Clusters of scaled iris data using the propos
```




```
plotcluster(newiris[,-1],truetypes(newiris),main="Clusters using the true labels")
```



As shown in the plots above, the k-means algorithm seems to work much better than when using the wine dataset, as the clustering is similar to that from using the true labels.

When scaling the data, the clustering does not suffer any major changes, which suggests the original data was already scaled.

As in Part (b), note that even though the clustering is similar, the resulting plot can be equivalent to a rotation of that using the true labels. This is because a random starting point is used.