homework 4

Matt Best

October 26, 2015

Metropolis Hastings

part 1

From our class notes, we know that the Metropolis-Hastings (MH) algorithm can be decomposed into steps:

0. Choose an initial starting point, x_0.
1. Pick a new candidate point, x^*, from the proposal (jumping) distribution, $Q(x^*|x_n)$ where x_n is the current observation.
2. Calculate the acceptance probability,

$$A(x_{n+1} \rightarrow x^*) = \min\left[1, \frac{P(x^*)}{P(x_n)}\frac{Q(x_n|x^*)}{Q(x^*|x_n)}\right].$$

   Here $P(x)$ is proportional to the probability that an observation x came from the stationary (target) distribution.
3. Sample a number, r from a uniform(0,1) distribution. If r is less than $A(x_{n+1} \rightarrow x^*)$, then $x_{n+1} = x^*$, otherwise $x_{n+1} = x_n$.

Because our proposal distribution is not symmetric, the second term in calculating the acceptance probability, $Q(x_n|x^*)/Q(x^*|x_n)$, will not be equal to 1, and so we must actually calculate it.

Now that we have provided a high-level overview of the algorithm, we will begin to describe our implementation. First, we wrote individual functions to compute $P(x)$ and $Q(x^*|x_n)$ detailed below:

```
stationary_prob <- function(x, a = 6, b = 4) {
  # Returns the probability that an observation came from a beta distribution
  # with scale parameters a and b
  return(dbeta(x, a, b))
}

proposal_prob <- function(x, phi, c = 1) {
  # Returns the probability that a proposal, x, came from a beta distribution
  # with scale parameters c*phi, and c * (1-phi)
  return(dbeta(x, c * phi, c * (1 - phi)))
}
```

where `stationary_prob` computes $P(x)$ and `proposal_prob` computes $Q(x^*|x_n)$. Next, we specified a function to make random draws from the proposal distribution:

```
proposal_function <- function(phi, c = 1) {
  # proposal_function(phi, c) will return a random draw from the proposal
  # distribution given curren observation, phi, and constant, c.
  prop <- 0
  reps <- 1
  maxReps <- 10000
```

```
  # The function `rbeta` can return either 0 or 1, but this shouldn't happen
  # so if either a 0 or 1 is generated, we will discard it and draw again.
  while ((prop == 0 | prop == 1) & reps < maxReps) {
    prop <- rbeta(1, c * phi, c * (1 - phi))
    reps <- reps + 1
  }
  if (reps == maxReps) {
    stop("Sampler is jammed.")
  }
  return(prop)
}
```

Now, we have all the helper functions that we need to implement the main MH algorithm as detailed in the function `run_mh`

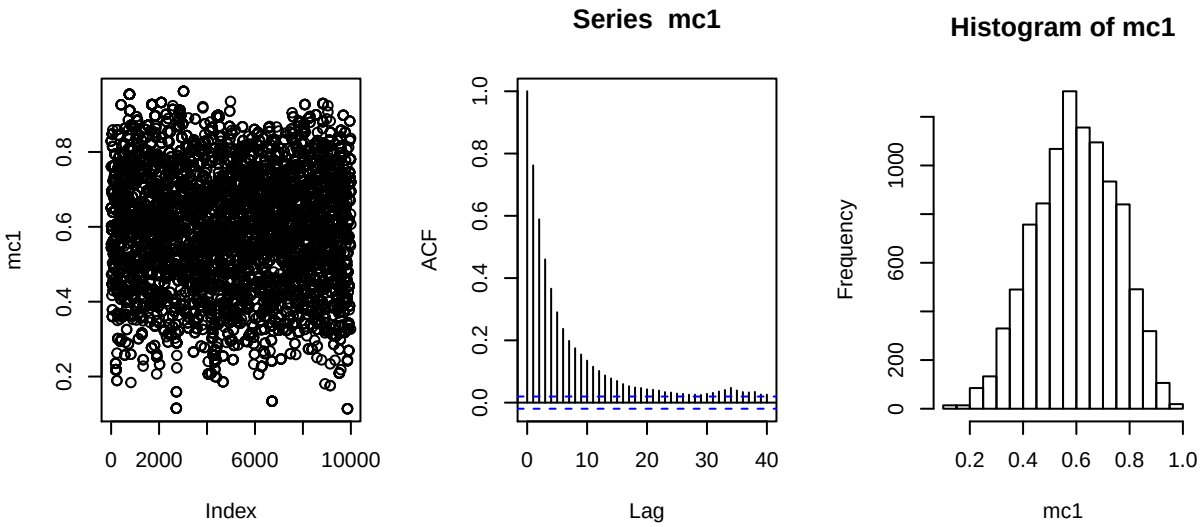
```
run_mh <- function(c = 1, startingPoint = runif(1), numIters = 10000, burnIn = 0, thinning = 0) {
  # run_mh(c, startingPoint, numIters, burnIn, thinning) will use the
  # Metropolis-Hastings algorithm to sample from a beta distribution with
  # scale parameters a = 6 and b = 4.  The width parameter, c, adjusts the
  # width of the proposal distribution.  Optional arguments include the
  # starting position of the Markov chain, startingPoint; the number of
  # iterations in the Markov chain, numIters; the number of samples to
  # discard for burn in, burnIn; and the number of samples that should be
  # discarded between consecutive draws of the chain, thinning.
  # The default value of thinning does not discard any samples.
  # run_mh returns the chain after discarding elements due to burn in and
  # thinning.
  m <- matrix( , nrow = numIters)
  xprev <- startingPoint

  for (iter in 1:numIters) {
    proposal <- proposal_function(xprev)

    a1 <- stationary_prob(proposal) / stationary_prob(xprev)
    a2 <- proposal_prob(xprev, proposal, c) / proposal_prob(proposal, xprev, c)

    a <- a1 * a2

    if (a > 1) {
      xprev <- proposal
    } else {
      if (runif(1) < a) {
        xprev <- proposal
      }
    }
    m[iter] <- xprev
  }
  m <- m[seq(burnIn+1, numIters, by=(thinning+1))]
  return(m)
}
```

part 2

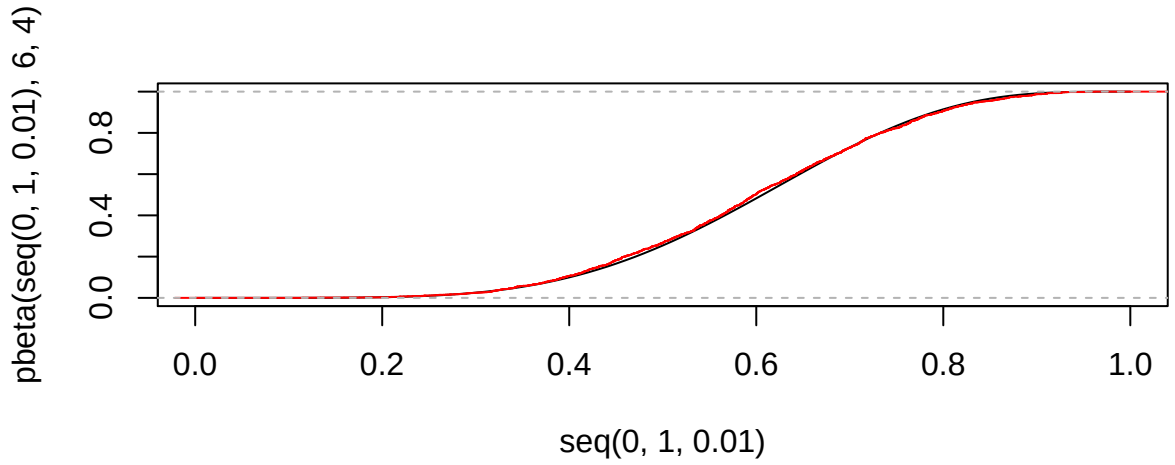
Using the code we developed in the previous section, we can compactly run our chain by calling:

```
mc1 <- run_mh(1)
```

With corresponding trace (left), autocorrelation (center), and histogram (right) plots:



Finally, we can compare the goodness of fit visually by comparing the empircal CDF with the actual CDF of a beta(6,4) distribution, and statistically using the KS-stat.




```
## Warning in ks.test(mc1, "pbeta", 6, 4): ties should not be present for the
## Kolmogorov-Smirnov test
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  mc1
## D = 0.0251, p-value = 6.525e-06
## alternative hypothesis: two-sided
```

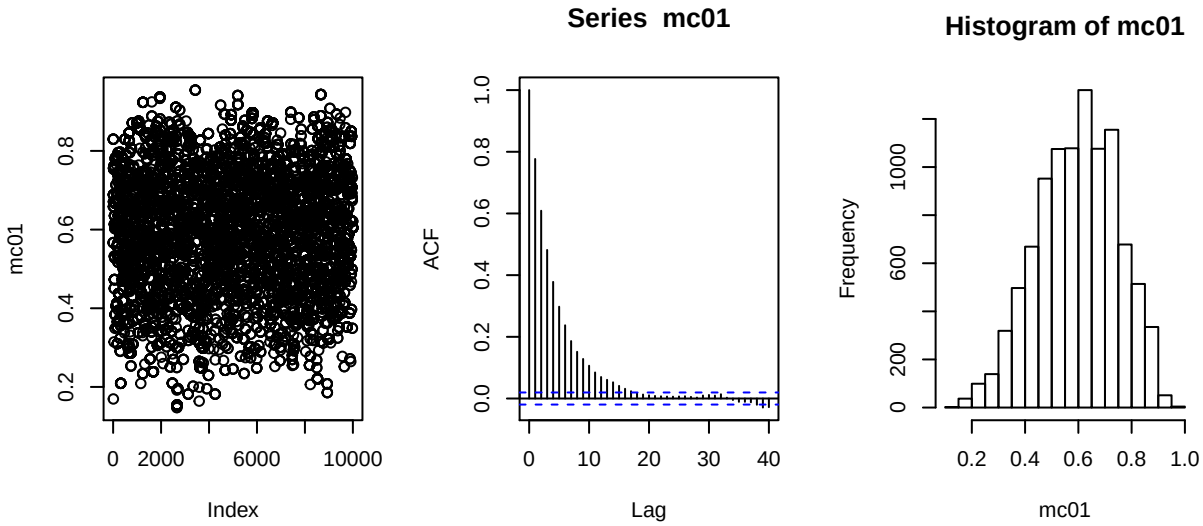
part 3

The sampler was re-run using different values of c:

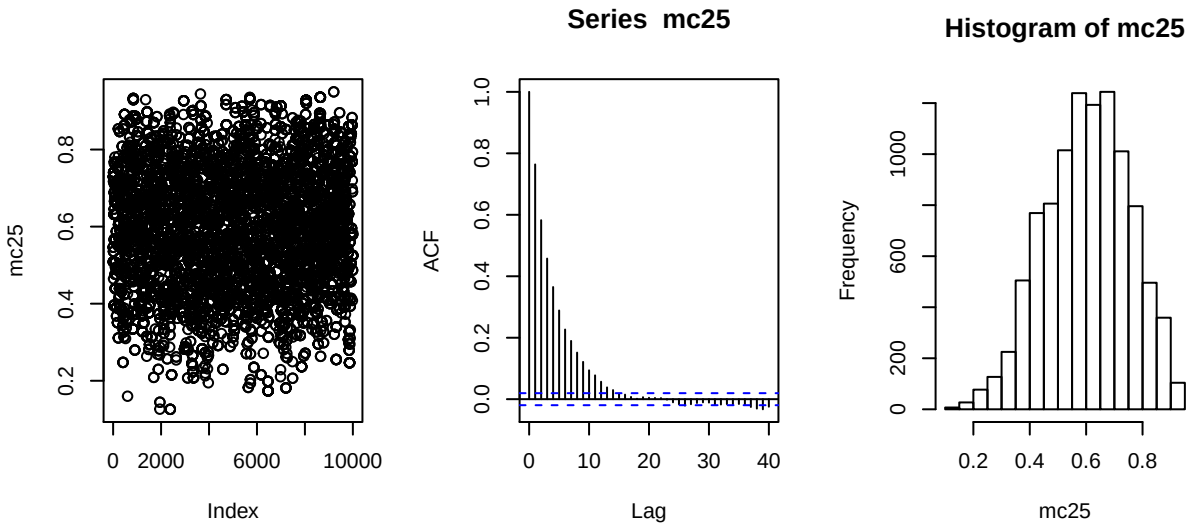
```
mc01 <- run_mh(.1)
mc25 <- run_mh(2.5)
mc10 <- run_mh(10)
```

Following the same conventions as part (b), we plotted the raw trace (left), autocorrelation (center), and histogram (right) of each sample.

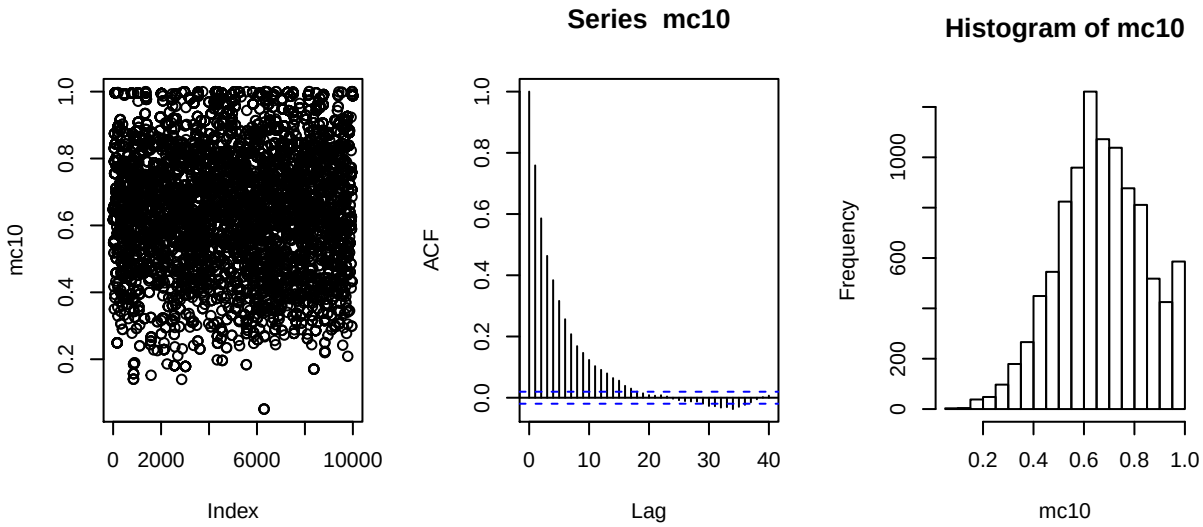
$c = 0.1$



$c = 2.5$



$c = 10$



Visually, the raw trace looks fairly stationary suggesting there is not an appreciable burn-in period for any values of c. However, we can see that the ACF function trails off slower for higher values of c suggesting that we should use thinning for larger values of c.

Quantitatively, again, we can use the KS-stat to compare the different c values.

```
ks.test(mc01, "pbeta", 6, 4)
```

```
## Warning in ks.test(mc01, "pbeta", 6, 4): ties should not be present for the
## Kolmogorov-Smirnov test
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  mc01
## D = 0.0193, p-value = 0.001202
## alternative hypothesis: two-sided
```

```
ks.test(mc25, "pbeta", 6, 4)
```

```
## Warning in ks.test(mc25, "pbeta", 6, 4): ties should not be present for the
## Kolmogorov-Smirnov test
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  mc25
## D = 0.0165, p-value = 0.008843
## alternative hypothesis: two-sided
```

```
ks.test(mc10, "pbeta", 6, 4)
```

```
## Warning in ks.test(mc10, "pbeta", 6, 4): ties should not be present for the
## Kolmogorov-Smirnov test
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  mc10
## D = 0.1663, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

We see that smaller c values have smaller KS statistics suggesting that they fit the theoretical beta(6,4) distribution better than larger c values which is consistent with our qualitative observations.

Gibbs

The Gibbs sampler algorithm is similar to the MH algorithm and can be written down in steps:

0. Pick an initial n-dimensional state, $S^{(0)}$ where the superscript denotes the ith iteration.
1. Update the first element in the state, $s_0^{(0)}$ according to the full conditional
$$P(s_0^{(1)}|s_1^{(0)}, s_2^{(0)}, \ldots, s_n^{(0)})$$
2. Generally, update the jth element in the state by drawing from the full conditional
$$P(s_j^{(i+1)}|s_0^{(i+1)}, \ldots, s_{j-1}^{(i+1)}, s_{j+1}^{(i)}, \ldots, s_n^{(i)})$$
3. Repeat k times

Based on this algorithmic description, we'll need to implement a function to draw from the full conditional distribution. The conditional distribution of y given x is the same as the conditional distribution of x given y up to a constant of proportionality, so we'll only need to implement one conditional function. Also, we note that $P(y|x) = P(x|y)$ are proportional to an exponential distribution on the interval $[0, B]$. In class, we talked about inverse transform sampling to use uniform random numbers to sample from any arbitrary distribution. We will now explain how we used this concept to draw random numbers from the specified conditional distribution. Generally, we would plug Unif(0,1) random numbers into the inverse CDF of an arbitrary distribution to sample from it. Here, we instead draw random numbers from the interval Unif(0, Y) where Y is the value of the CDF of an exponential distribution with rate λ evaluated at point B. We then plug those uniform random draws back into the inverse CDF of the exponential distribution with rate λ to sample from the conditional distribution. In code, this can be expressed in a single line as follows:

```
sample_conditional_distribution <- function(lambda, B = 5){
  # Draws a random sample from the conditional distribution of x given y or
  # y given x
  return(qexp(runif(1, 0, pexp(B, lambda)), lambda))
}
```

Now, to show the full Gibbs sampling algorithm:

```
run_gibbs <- function(numIter = 500) {
  # run_gibbs(numIter) draws numIter random samples using Gibbs sampling
  m <- matrix( , nrow = numIter, ncol = 2)
  m[1,] <- runif(2)
  for (iter in 2:numIter) {
    m[iter,1] <- sample_conditional_distribution(m[iter-1, 2])
    m[iter,2] <- sample_conditional_distribution(m[iter, 1])
  }
  return(m)
}
```

To calculate the first moment of the marginal distribution of X, we can simply average over observations of the X component from our random sample, e.g.:

```
s500 <- run_gibbs(500)
s5000 <- run_gibbs(5000)
s50000 <- run_gibbs(50000)

print(mean(s500[,1]))
```

```
## [1] 1.200397
```

```
print(mean(s5000[,1]))
```

```
## [1] 1.217387
```

```
print(mean(s50000[,1]))
```

```
## [1] 1.28474
```

K-Means

We begin by loading the data and also scaling the data. Scaling the data involves subtracting the mean of each feature and then dividing by the standard deviation so that each feature has its mean equal to 0 and standard deviation equal to 1.

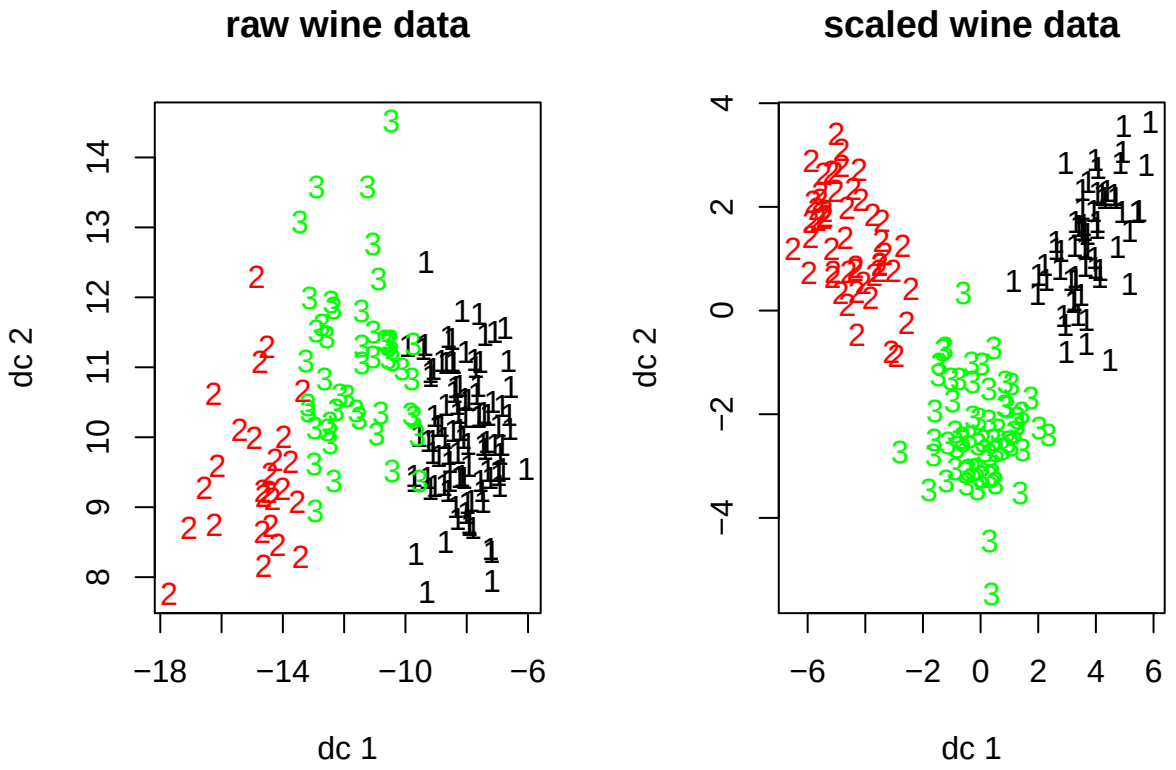
```
trainData <- wine[,2:14]
trainDataScaled <- data.frame(scale(trainData))

clustUnScaled <- km(trainData, 3)
```

```
## Algorithm converged in  13  iterations.
```

```
clustScaled <- km(trainDataScaled, 3)
```

```
## Algorithm converged in  8  iterations.
```

```
par(mfrow=c(1,2))
plotcluster(trainData, clustUnScaled, main="raw wine data")
plotcluster(trainDataScaled, clustScaled, main="scaled wine data")
```



From visual inspection of the plots, both raw and scaled data are separated by the 13 features. To quantify just how well the data were classified into each of the categories, we used the intersection over union metric described in the computer vision literature. We also use this time to mention that there is a model identifiability issue, that is, there is no way to be sure that what we defined to be cluster 1 is also cluster 1 in the wine data. Accordingly, we check all 6 possible reorderings of cluster labels and choose the one with the highest IOU.

```
table(wine[,1], clustUnScaled)
```

```
##    clustUnScaled
##      1  2  3
##   1  1 27 31
##   2 64  0  7
##   3 37  0 11
```

```
table(wine[,1], clustScaled)
```

```
##    clustScaled
##      1  2  3
##   1 59  0  0
##   2  3  3 65
##   3  0 48  0
```

```
findBestIOU(wine[,1], clustUnScaled)
```

```
## [1] 0.4203798
```

```
findBestIOU(wine[,1], clustScaled)
```

```
## [1] 0.9343912
```

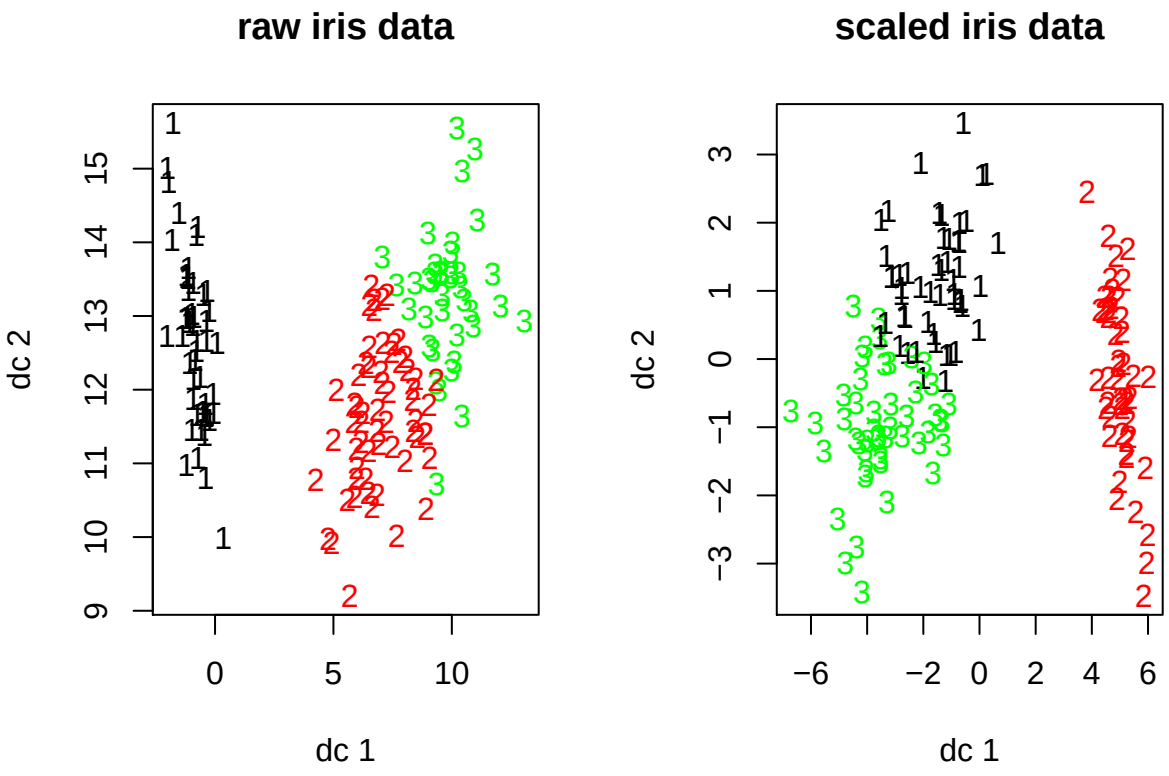
Quantitatively, we see that using the scaled data leads to much better classification accuracy. Now, for the iris data:

```
data(iris)
irisData <- iris[,1:4]
irisDataScaled <- data.frame(scale(irisData))

irisClustUnScaled <- km(irisData, 3)
```

```
## Algorithm converged in  13  iterations.
```

```
irisClustScaled <- km(irisDataScaled, 3)
```

```
## Algorithm converged in  5  iterations.
```




```
## [1] 0.8045401
```

```
##              irisClustUnScaled
##                1  2  3
##   setosa      50  0  0
##   versicolor   0 47  3
##   virginica    0 14 36
```

```
## [1] 0.7247984
```

```
##             irisClustScaled
##               1  2  3
##   setosa      0 50  0
##   versicolor 38  0 12
##   virginica  14  0 36
```

In the iris dataset, scaling does not improve separability of the clusters as much as in the wine data, although K-means on both scaled and unscaled iris data leads to fairly good classification performance.