Summative report

Iro Chalastani Patsioura

17/11/2021

Question 1

a)

Before starting the questions we first need to read in the data, by running the commands below:

```
library(nclSLR)
```

```
## Loading required package: plyr
```

```
##
## Attaching package: 'nclSLR'
```

```
## The following object is masked from 'package:datasets':
##
##     USArrests
```

```
library(readr)
gexpr = read.csv("/Volumes/KINGSTON/Ch10Ex11.csv", header = FALSE)
dim(gexpr)
```

```
## [1] 1000   40
```

We need to note that this gene expression data set is stored the "wrong" way around with rows representing genes and columns representing tissue samples. Therefore we need to transpose gexpr to get our data matrix, using the command below:

```
gexpr_t= t(gexpr)
dim(gexpr_t)
```

```
## [1]   40 1000
```

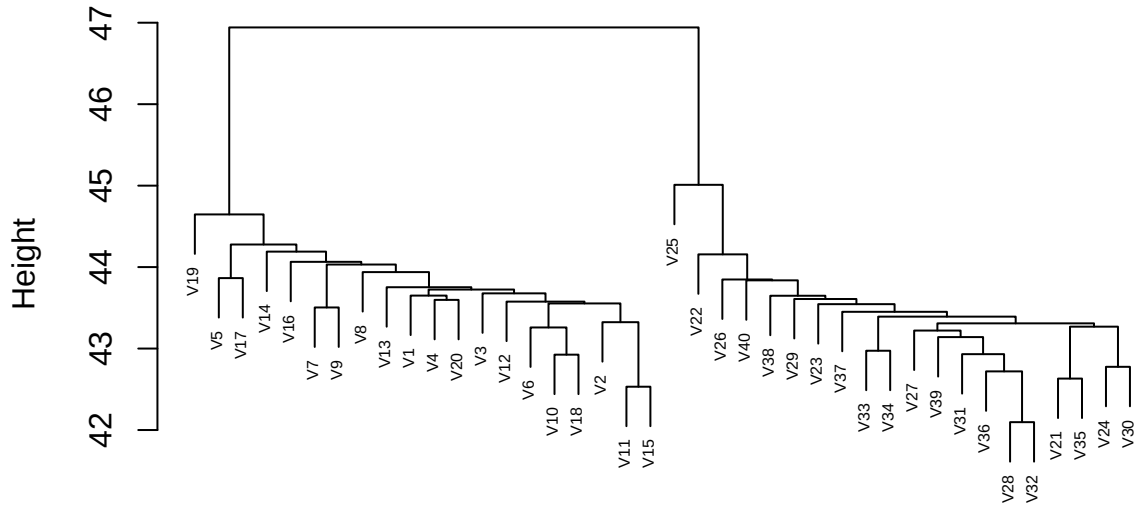
Looking at the dimensions of the transposed matrix we see that we are now ready to start our analysis.

Now we would like to apply hierarchical clustering with single linkage. In order to do that e first need to compute the distance matrix using the command below:

```
d = dist(gexpr_t)
```

Then we want to apply hierarchical clustering and plot the dentogram. This can be done by using the code bellow:

```
hc_c = hclust(d, method="single")
plot(hc_c, cex=0.5, main="", sub="", xlab="")
```

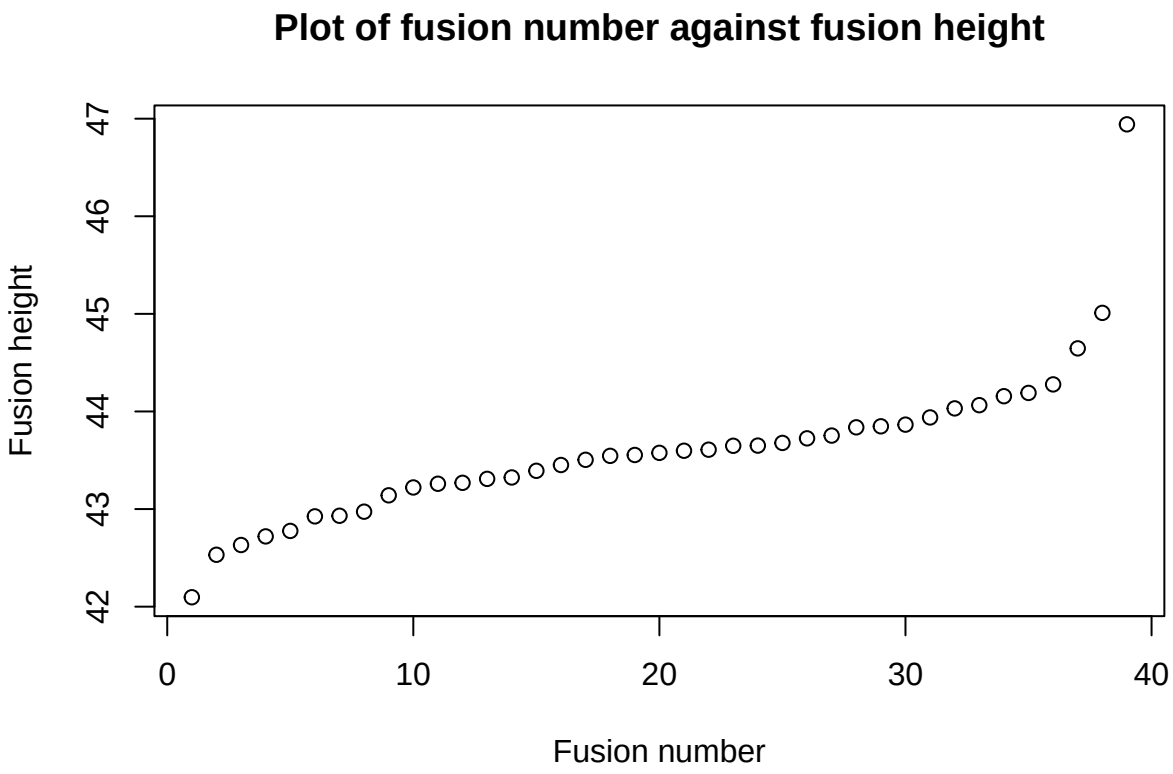


In general we observe two different clusters. In order to decide where to cut the tree we take a look at the heights of consecutive fusions and we search for a big jump. It is easier to observe that if we plot the heights. We first need to extract the heights using the command below and then we just generate the plot:

```
hc_c$height
```

```
##  [1] 42.09653 42.53182 42.63166 42.72007 42.77586 42.92528 42.93143 42.97320
##  [9] 43.14073 43.22124 43.25919 43.26878 43.30933 43.32386 43.39200 43.45128
## [17] 43.50457 43.54520 43.55411 43.57611 43.59758 43.60848 43.64836 43.65024
## [25] 43.67756 43.72466 43.75299 43.83746 43.84798 43.86501 43.93872 44.03160
## [33] 44.06459 44.15642 44.19001 44.27753 44.64663 45.01009 46.94188
```

```
plot(hc_c$height, xlab="Fusion number", ylab="Fusion height",
     main = "Plot of fusion number against fusion height")
```



We observe a bigger jump from around 45 to 47, so we can decide to cut the tree there, by using the following command:

```
cutree(hc_c, h=45.5)
```

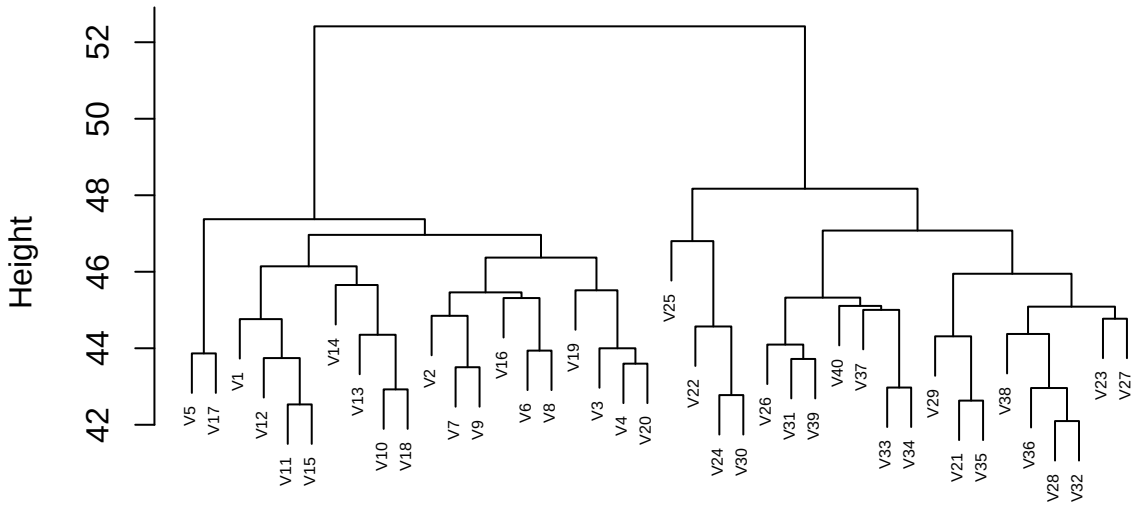
```
##  V1  V2  V3  V4  V5  V6  V7  V8  V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
## V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38 V39 V40
##   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
```

By examining the output of the command above, we observe that the genes separate the two groups of healthy and diseased patients. This is expected, as we would expect that the healthy patients would have more similar gene expressions among them and quite different gene expressions with the ones that are diseased

b)

Now we will repeat part a using complete and average linkage in order to see if the result changes. So we apply complete linkage and plot the corresponding dentogram, using the commands below:

```
hc_c_complete = hclust(d, method="complete")
plot(hc_c_complete, cex=0.5, main="", sub="", xlab="")
```

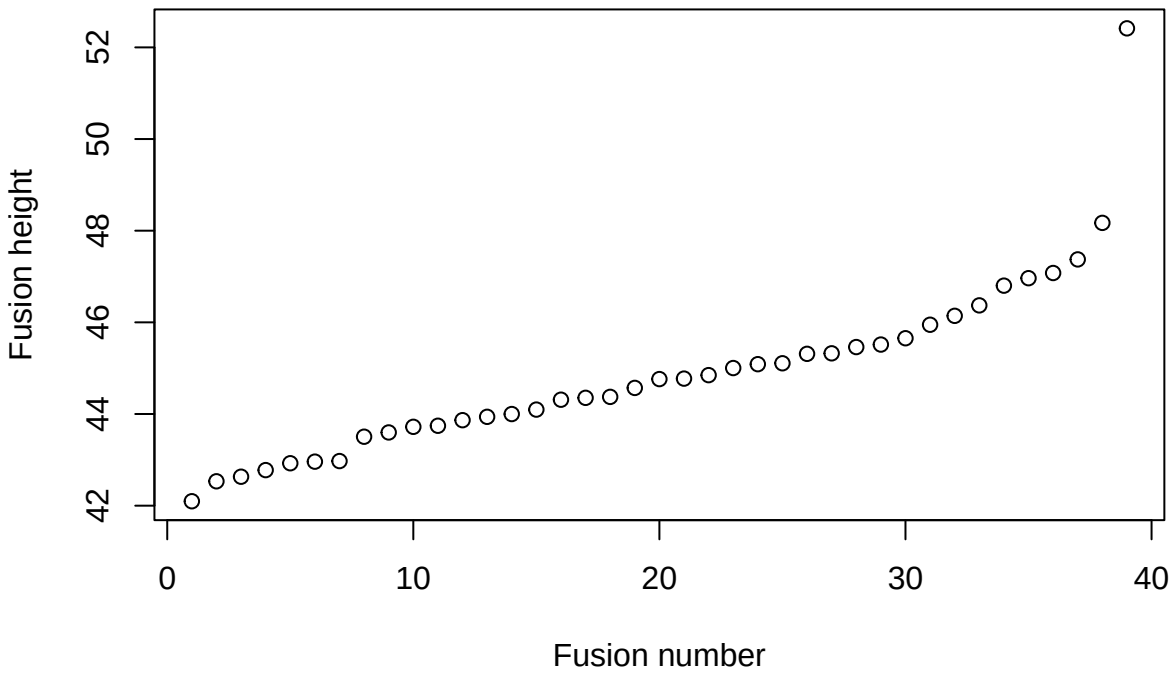


Again just like before we observe two distinct clusters. So as we did before, we would like to look at the heights in order to decide where to cut the dentogram. We do this with the following commands:

```
hc_c_complete$height
```

```
##  [1] 42.09653 42.53182 42.63166 42.77586 42.92528 42.96053 42.97320 43.50457
##  [9] 43.59758 43.72022 43.74351 43.86501 43.93872 43.99898 44.09622 44.31208
## [17] 44.35277 44.37378 44.56944 44.76197 44.77231 44.84821 45.00296 45.08712
## [25] 45.10672 45.31276 45.32377 45.46096 45.51552 45.65311 45.94784 46.14201
## [33] 46.36994 46.80087 46.96363 47.07659 47.37353 48.17044 52.41502
```

```
plot(hc_c_complete$height, xlab="Fusion number", ylab="Fusion height")
```



Again we observe a jump between 48.5 and 52.5 so again we cut the dentogram at 49 and we end up with two clusters which we can examine with the following command:

```
cutree(hc_c_complete, h=49)
```

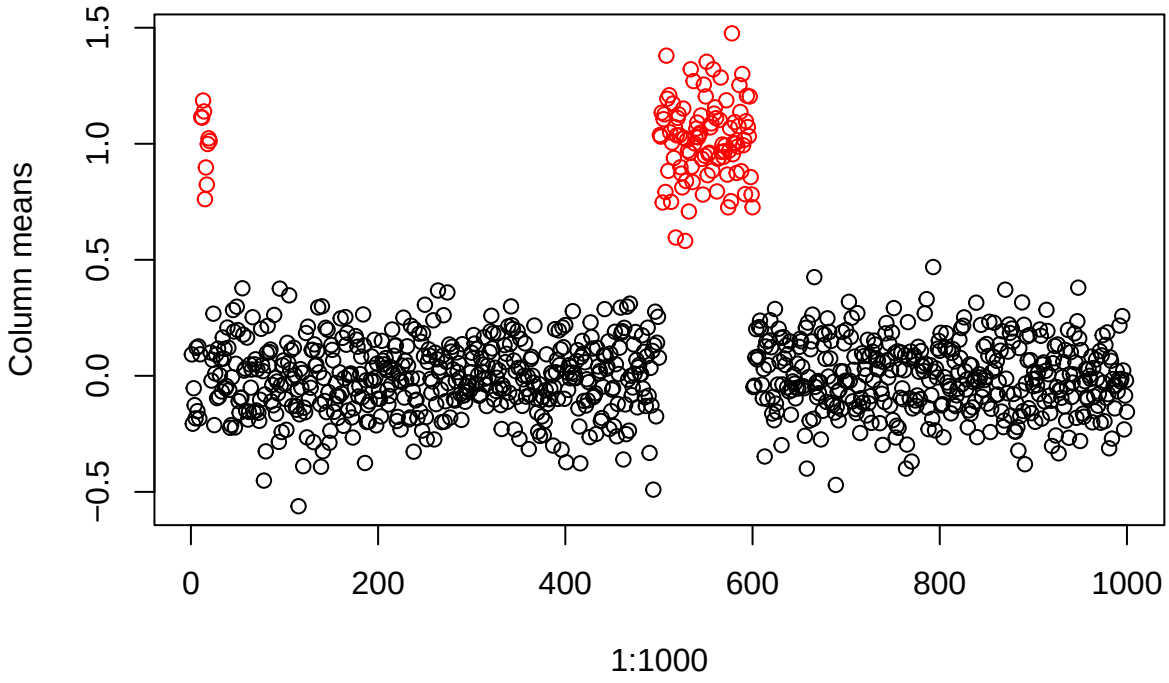
```
##  V1  V2  V3  V4  V5  V6  V7  V8  V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
## V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38 V39 V40
##   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
```

So we observe the same clusters as before. As a conclusion we can say that in this particular case the results does not really depend on the type of linkage, since regardless of the type of linkage we end up with the same clusters.

c)

Now we would like to know which are these genes that affect the survival of the patients. So we would like to know which genes differ the most across the two groups. One way we could do that is copmute and plot the column means in order to see if some genes differ compared to others:

```
plot(1:1000, colMeans(gexpr_t), col = ifelse(colMeans(gexpr_t)>0.5 ,"red", "black"),
     ylab = "Column means")
```



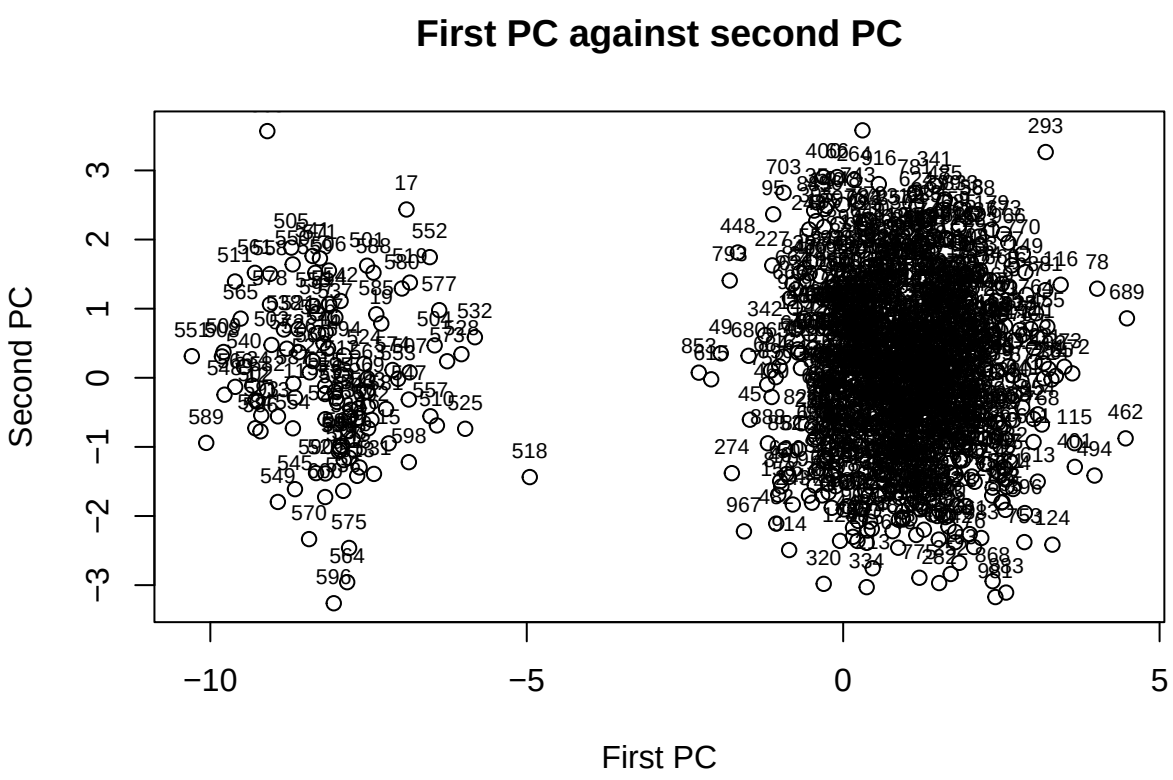
Just be looking at the column means of the genes in the 40 patients, we observe that some specific genes have quite a higher mean compared to the most other genes. Now what we would like to do is find which are these genes. So the red colored genes are the ones who probably affect the health of the patients.

In order to find out which genes are the red genes, we can perform principal component analysis to the original data set using the command:

```
pca1 = prcomp(x=gexpr)
```

Now we can plot the first and second principal components and add text labels to the plot in order to identify the specific genes that are causing the problem:

```
plot(pca1$x[,1], pca1$x[,2], xlab="First PC", ylab="Second PC", main = "First PC against second PC")
text(pca1$x[,1], pca1$x[,2], labels=rownames(gexpr), cex=0.7, pos=3)
```



Again we observe the separation in the genes, and by the labels we can see which genes are the ones that seem separated from the others. However, since the plot is a bit "crowded" it is not easy to identify all the problematic genes. So we just need to create a vector which will show all of the problematic genes. We can do this with the following command:

```
pca1vector = pca1$x[,1]
pca1vector[pca1vector < -4]
```

```
##   [1]  -8.662973  -9.252725  -8.931341  -8.140032  -7.181091  -7.505148
##   [7]  -6.900684  -7.641193  -7.297997  -7.749216  -7.522551  -9.285554
##  [13]  -9.033315  -6.455122  -8.719213  -8.126355  -6.851374  -9.816359
##  [19]  -9.791750  -6.419551  -9.609794  -7.923072  -7.679089  -8.122808
##  [25]  -8.184554  -8.297196  -8.176223  -4.952085  -6.846395  -8.180459
##  [31]  -8.610344  -8.429328  -7.996676  -7.568524  -5.972627  -8.602761
##  [37]  -7.891827  -6.031942  -8.181009  -7.795163  -7.414534  -5.819636
##  [43]  -7.805625  -9.336146  -8.014749  -7.899713  -8.023870  -8.837236
##  [49]  -7.817879  -9.467449  -8.383850  -7.941456  -7.659686  -7.783884
##  [55]  -8.662590  -8.211294  -6.863915  -9.775649  -8.932683  -8.698852
##  [61] -10.292049  -6.532170  -7.033146  -8.694854  -8.375620  -7.947247
##  [67]  -6.521650  -9.056556  -8.341870  -8.378776  -9.295200  -7.911970
##  [73]  -7.516726  -7.837521  -9.520662  -9.606732  -7.665598  -9.099219
##  [79]  -7.524227  -8.439420  -8.267621  -8.791900  -6.257470  -7.120033
##  [85]  -7.807433  -8.186686  -6.383068  -9.057344  -7.721661  -6.972973
##  [91]  -7.223292  -9.101020  -7.510004  -9.288659  -7.377215  -9.206890
##  [97]  -8.687749  -7.421920 -10.064668  -8.329243  -7.970035  -7.453630
## [103]  -9.194464  -7.889684  -8.035233  -8.049327  -8.312795  -6.862814
## [109]  -7.956691  -8.185498
```

```
different_genes = which(pca1vector < -4)
different_genes
```

```
##   [1]  11  12  13  14  15  16  17  18  19  20 501 502 503 504 505 506 507 508
##  [19] 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526
##  [37] 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544
##  [55] 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562
##  [73] 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580
##  [91] 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598
## [109] 599 600
```

So now we have identified the genes that differ the most accross the two groups. These are stored in the different genes vector.

Question 2

a)

Here we would like to perform K means clustering, by writing a function that does this exact thing. The following function performs K means clustering, by just entering the data frame and the number of clusters that we would like:

```
kmeans_function = function(x,k){
#making sure that the data entered are in matrix form
x = as.matrix(x)
#choosing the initial means randomly
initial_means = x[sample(nrow(x),size=k,replace=FALSE),]
old_clusters = rep(1,nrow(x))
clusters = rep(2, nrow(x))
# Create an empty matrix which will later be populated with the distance of the matrix
#points to the means
distance_from_means = matrix(nrow = nrow(x), ncol = k)
while(all(old_clusters != clusters)){
  # Set the old clusters equal to the clusters found
  old_clusters = clusters
  for(i in 1:nrow(x)){
    for(j in 1:k){
      # Find the euclidian distance for each each row to the k
      # randomly selected initial means
      distance = sqrt(sum((x[i,] - initial_means[j,])^2))
      # populate the columns of the matrix with the distances
      distance_from_means[i,j] = distance
    }
}
# Assign clusters based on which distance is closer to the means
clusters = apply(distance_from_means, 1, which.min)
# Find the means of the different clusters
centers = apply(x, 2, tapply, clusters, mean)
# Set the new means as the initial means in order to run the for loop with the new means
initial_means = centers
```

```
}
#Create a for loop, that goes over the rows of the matrix finds the total within variation
within_clusters_sum_of_squares = 0
for(g in 1:nrow(x)){
  within_clusters_sum_of_squares =
    within_clusters_sum_of_squares + sum((x[g,] -  centers[clusters[g],])^2)
}
#Create a list that contains the cluster partition, the centers
# and the witihn cluster sum of squares
y = list(clusters,centers, within_clusters_sum_of_squares)
# Put headers on the list elements
names(y) = c("cluster partition", "cluster means", "within clusters sum of squares")
# Return the list
return(y)
}
```

Now we can apply this function to our data set in order to see if it performs as it should. We check the clusters to see if they are the same as in part 1:

```
km = kmeans_function(gexpr_t,2)
km[[1]]
```

```
##  [1] 1 2 2 1 1 1 2 2 2 1 1 1 1 1 2 2 1 1 1 1 1 1 2 1 1 2 1 2 1 1 2 2 2 2 1 1 1 2
## [39] 1 2
```

We see that the function works as expected.

b)

Now taking K =3, we apply the kmeans function to the US Arrests data. First we need to load the data:

```
# Load nclSLR package
library(nclSLR)
data(USArrests, package="nclSLR")
```

Now we run the function from part a multiple times and find the replication that gives the smallest value of SSw. We can replicate the function and find the SSw for each run using the commands below:

```
# set seed to make sure we get the same result every time
set.seed(1)
run = replicate(50, kmeans_function(USArrests[,1:4],3), simplify = FALSE)
ssw = sapply(run, "[[", 3)
ssw
```

```
##  [1]  69169.23 114507.22  79654.68  86143.08  63186.99  66791.09  72928.25
##  [8]  91694.32  61819.82  56722.89  69221.27  61449.96  81992.63  56022.55
## [15]  66946.30  87375.22  95777.40  79317.36  67253.57  66657.78  52009.86
## [22]  66946.30 109566.02  83148.22  80060.21  51922.53  51174.45  96466.71
## [29]  56722.89  56722.89 103320.49 112642.80 108684.94  81568.76  51174.45
## [36]  47964.27  88191.21  47964.27 102896.76 280142.77  81788.29  71967.84
## [43]  64205.98 108797.24  95723.11  70591.02 114428.92 188448.23  81501.79
## [50] 117048.68
```

Now in order to identify the smallest SSw we use:

```
which.min(ssw)
```

```
## [1] 36
```

So now we extract that run that gave the smallest SSw in order to to compare it with the built in algorithm:

```
min_run = run[which.min(ssw)]
min_run
```

```
## [[1]]
## [[1]]$`cluster partition`
##  [1] 3 3 3 2 3 2 1 3 3 2 1 1 3 1 1 1 1 3 1 3 2 3 1 3 2 1 1 3 1 2 3 3 3 1 1 2 2 1
## [39] 2 3 1 2 2 1 1 2 2 1 1 2
##
## [[1]]$`cluster means`
##      Murder  Assault UrbanPop     Rape
## 1  4.270000  87.5500 59.75000 14.39000
## 2  8.214286 173.2857 70.64286 22.84286
## 3 11.812500 272.5625 68.31250 28.37500
##
## [[1]]$`within clusters sum of squares`
## [1] 47964.27
```

Now we run the K means built in R algorithm and extract the cluster partition, centers and total within variation in order to compare if we get the same results with the ones when using the function from part a:

```
km = kmeans(USArrests[,1:4],3)
km$cluster
```

```
##        Alabama         Alaska        Arizona       Arkansas     California
##              1              1              1              2              1
##       Colorado    Connecticut       Delaware        Florida        Georgia
##              2              3              1              1              2
##         Hawaii          Idaho       Illinois        Indiana           Iowa
##              3              3              1              3              3
##         Kansas       Kentucky      Louisiana          Maine       Maryland
##              3              3              1              3              1
##  Massachusetts       Michigan      Minnesota    Mississippi       Missouri
##              2              1              3              1              2
##        Montana       Nebraska         Nevada  New Hampshire     New Jersey
##              3              3              1              3              2
##     New Mexico       New York North Carolina   North Dakota           Ohio
##              1              1              1              3              3
##       Oklahoma         Oregon   Pennsylvania   Rhode Island South Carolina
##              2              2              3              2              1
##   South Dakota      Tennessee          Texas           Utah        Vermont
##              3              2              2              3              3
##       Virginia     Washington  West Virginia      Wisconsin        Wyoming
##              2              2              3              3              2
```

```
km$centers
```

```
##      Murder  Assault UrbanPop     Rape
## 1 11.812500 272.5625 68.31250 28.37500
## 2  8.214286 173.2857 70.64286 22.84286
## 3  4.270000  87.5500 59.75000 14.39000
```

```
km$tot.withinss
```

```
## [1] 47964.27
```

We observe that we end up with the same partition as well as the same cluster means, so the function seems to be working as expected

c)

No, it is possible that the procedure will not give the same result. This happens because the k means algorithm is guaranteed to converge to a minumun value of SSW, but this may be the local and not the global minimum. So depending on the starting point the kmeans algorithm may find a partition with a local mimimun SSW abd stop there, while the procedure in part 2, which may have a different starting point could find the global maximums and thus the results could be different. Even if the algorithm is run with different initial starting points, this may find the global minimum, but again there is no guarantee that this will always happen.

Question 3

a)

First we need to load the diabetes data using the following code:

```
## Load the nclSLR package
library(nclSLR)
## Load the data
data(diabetes)
# We also need to standardise the predictor variables
diabetes[,1:10] = scale(diabetes[,1:10])
```

Now we can define the training and test sets:

```
train_set = diabetes[1:350,]
test_set = diabetes[351:442, ]
```

b)

Now we can fit the linear regression model with all the predictors to the training data:

```
lsq_train = lm(dis ~ ., data= train_set)
```

The next step is to find the test error for the model. In order to do that we need to compute the fitted values for the test data, and then compute the test error over the test set. The command below gives the fitted values:

```
yhat_test = predict(lsq_train, test_set)
head(yhat_test)
```

```
##       351       352       353       354       355       356
## 249.73714  87.35192  59.67330 171.13304 194.26137 132.80131
```

Now we can find the test error:

```
test_error = mean((test_set$dis - yhat_test)^2)
test_error
```

```
## [1] 2842.256
```

We find the test error to be 2842.256.

c)

From practical 5 we see that the model selected by subset selection includes sex, bmi, map,tc, ldl and ltg. So we fit the model with just these variables to the training set:

```
lsq_train_ss = lm(dis ~ sex + bmi + map + tc + ldl + ltg , data = train_set)
```

Just like before we compute the fitted values for the test set and then we find the test error:

```
yhat_test_ss = predict(lsq_train_ss, test_set)
head(yhat_test_ss)
```

```
##       351       352       353       354       355       356
## 244.10590  86.19854  60.12041 162.33036 196.62034 129.32308
```

```
test_error = mean((test_set$dis - yhat_test_ss)^2)
test_error
```

```
## [1] 2800.964
```

The test error is 2800.964. As expected it has reduced.

d) i)

I)

First we need to load the pls library using:

```
library(pls)
```

```
##
## Attaching package: 'pls'
```

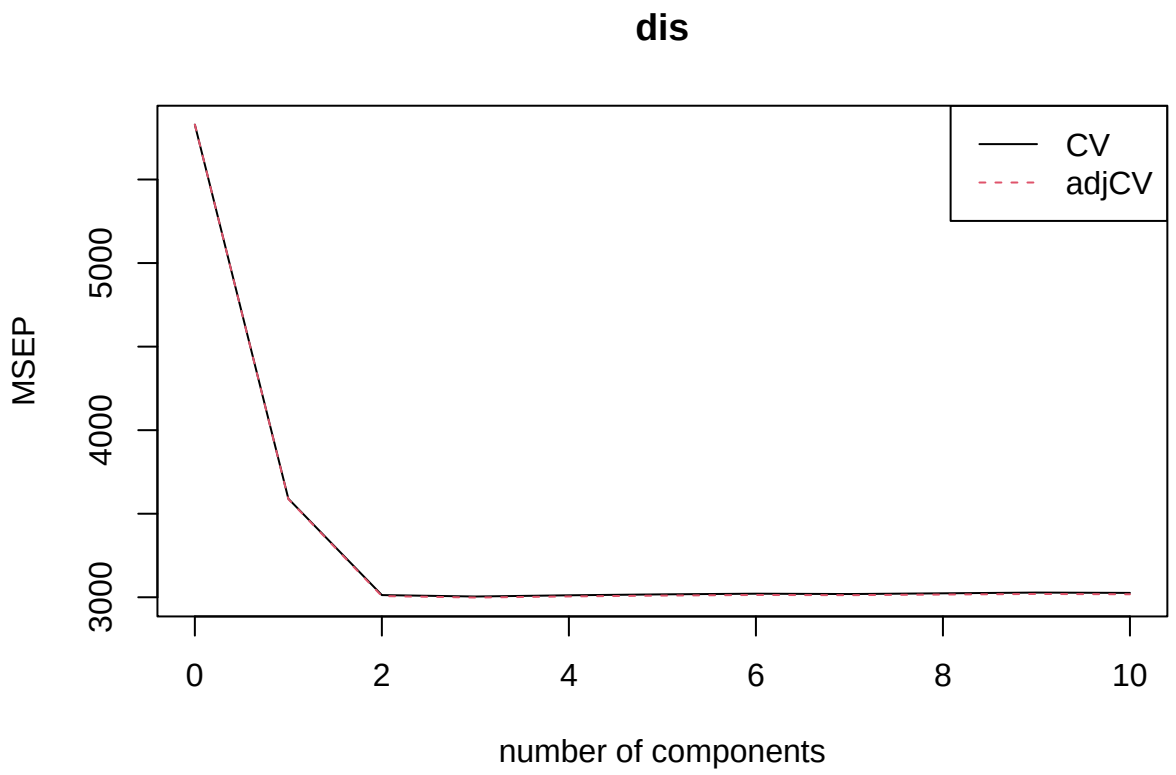
```
## The following object is masked from 'package:stats':
##
##     loadings
```

Then we can fit the model on the training data using partial least squares:

```
plsr_fit = plsr(dis ~ ., data = train_set, scale=FALSE)
```

Now we can use cross validation in order to identify the optimal number of components. We can also plot the MSE against the number of components to see if there is a kink. For performing cross validation and creating the plot we need the following commands:

```
plsr_cv_fit = plsr(dis ~ ., data=train_set, scale=FALSE, validation = "CV")
plot(plsr_cv_fit, plottype="validation", legend="topright", val.type="MSEP")
```



Looking at the plot we observe a clear "elbow" at 2, so we can say that the optimal number of transformed variables is 2. ## II) Now in order to compute the test error we need to find the fitted values using the test data using the optimal number of transformed identified in part I. This can be done with the following commands:

```
yhat_m2 = predict(plsr_fit, test_set, ncomp = 2)
test_error_1 = mean((test_set$dis - yhat_m2)^2)
test_error_1
```

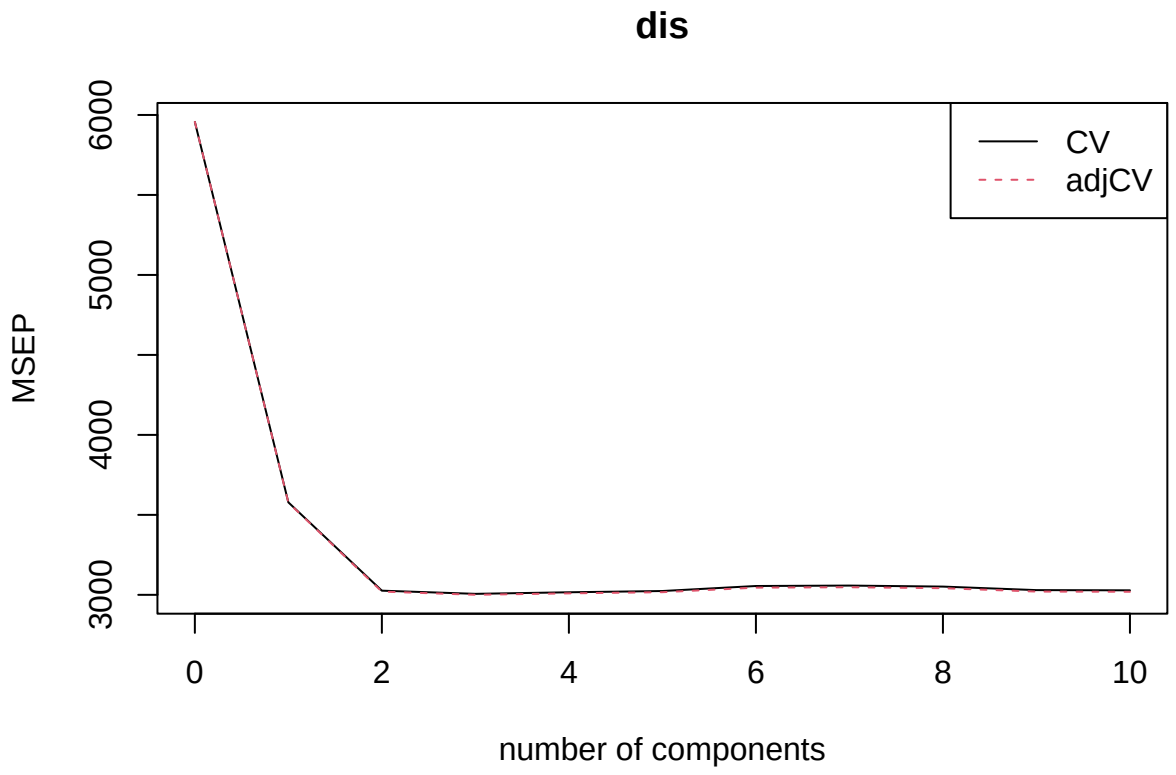
```
## [1] 2929.477
```

The test error is 2929.477 ## ii) ## I) Now we fit the model based on the whole data set with the following command:

```
plsr_fit_full = plsr(dis ~ ., data = diabetes, scale=FALSE)
```

Then just like before we perform cross validation and make a plot in order to identify the optimal number of transformed variables:

```
plsr_cv_fit_full = plsr(dis ~ ., data= diabetes, scale=FALSE, validation = "CV")
plot(plsr_cv_fit_full, plottype="validation", legend="topright", val.type="MSEP")
```



Again we observe a clear "elbow" at 2, so the number of optimal transformed variables is 2. ## II) Now fitting the model to the whole data set and fixing the transformed variables to the optimal values identified in part I, we can find the regression coefficients and compare them to the estimated coefficients in the full model fitted by least squares. This can be done with the commands below:

```
coef(plsr_fit_full, intercept=TRUE, ncomp=2)
```

```
## , , 2 comps
##
##                     dis
## (Intercept) 152.1334842
## age           0.4610834
## sex          -8.9967187
## bmi          24.3847591
## map          15.4490374
## tc           -3.9505440
## ldl          -7.3710240
## hdl         -11.0813316
## tch           5.7652353
## ltg          18.9021295
## glu           7.6758346
```

```
lsq_fit = lm(dis ~ ., data=diabetes)
lsq_fit$coefficients
```

```
## (Intercept)         age         sex         bmi         map          tc
## 152.1334842  -0.4767713 -11.4199566  24.7542756  15.4471632 -37.7230553
##         ldl         hdl         tch         ltg         glu
##  22.7021828   4.8116462   8.4316274  35.7752058   3.2202565
```

Looking at the coefficients we observe that the coefficients for bmi and map, are almost the same. Moreover we now have 4 negative coefficients compared to two negative ones that we had when using least squares. Also we observe that the absolute value of the coefficients for hld and glue has increased. On the other hand all of the other values have gone closer to zero, when compared to the least squares fitted model.

III)

Now if we want to interpret the first PLS direction we need the following commands:

```
plsr_load_full = loadings(plsr_fit_full)
#Get the directions
C_full = unclass(plsr_load_full)
# Print the first dierction
C_full[,1]
```

```
##        age        sex        bmi        map         tc        ldl        hdl
##  0.2147773  0.1667704  0.3740632  0.3191504  0.3001002  0.2996655 -0.3206969
##        tch        ltg        glu
##  0.4312452  0.4207026  0.3502178
```

Now we can think of the first axis as a contrast between hdl and all the other variables.

e)

We observe that we get the smallest test error (2800.964) when we use just the 7 variables identified by subset selection in part c. Moreover comparing the errors for least squares (2800.964) and partial least squares (2929.477) we see that the error when using least squares is lower. So the best model is the one identified by subset selection, since it has the lowest test error.