732A99 - Lab 1, block 2 Group L4

Keshav Padiyar, Jacob Welander & Henrik Olofsson

02 December, 2020

Contents

statement of contribution

Keshav Padiyar: Assignment 1

Jacob Welander: Assignment 2

Henrik Olofsson: Assignment 3

Assignment 1. Ensemble methods

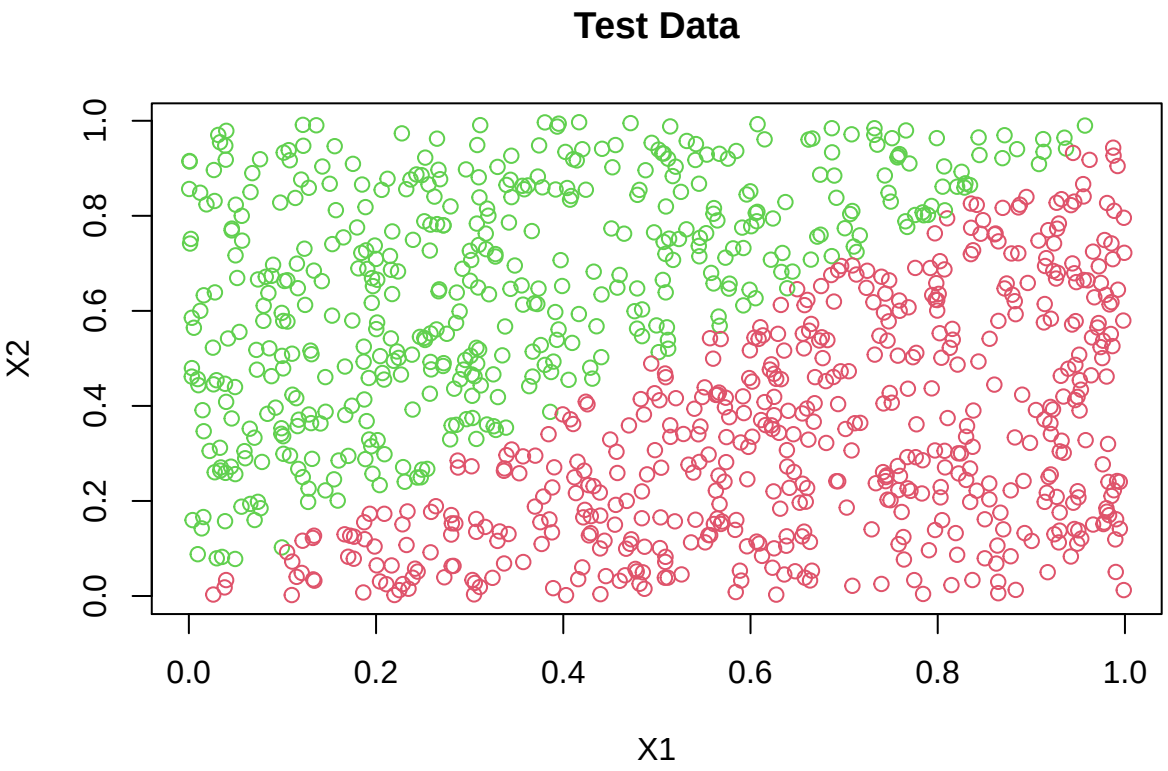
The task is to classifying Y from X1 and X2, where Y is binary and X1 and X2 continuous. Learn a random forest with 1, 10 and 100 trees.

1. Classifying Y from X1 and X2: X1<X2

Classifying Y from X1 and X2. Where $Y = \{^{1,\ X1<X2}_{0,\ other\ wise}$ **nodeSize = 25**

After repeating the above procedure for 1000 training data sets of size 100, following are the observations.

```
#a
Error_a = run_model(1000, 25, condition = "a")
```



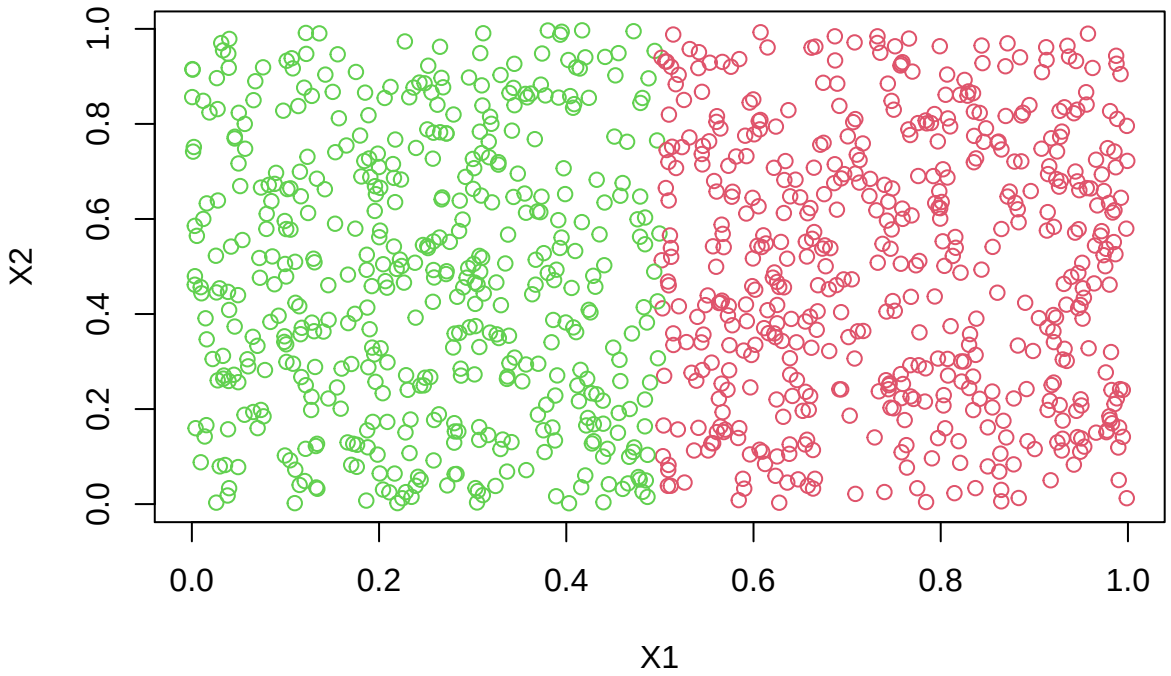

```
Mean Classification Error:
  t1_tes_err  t10_tes_err t100_tes_err
    0.209946     0.135147     0.111236

Varriance in  Classification Error:
  t1_tes_err  t10_tes_err t100_tes_err
0.0034127338 0.0009553587 0.0008664728
```

2. Classifying Y from X1 and X2: X1<0.5

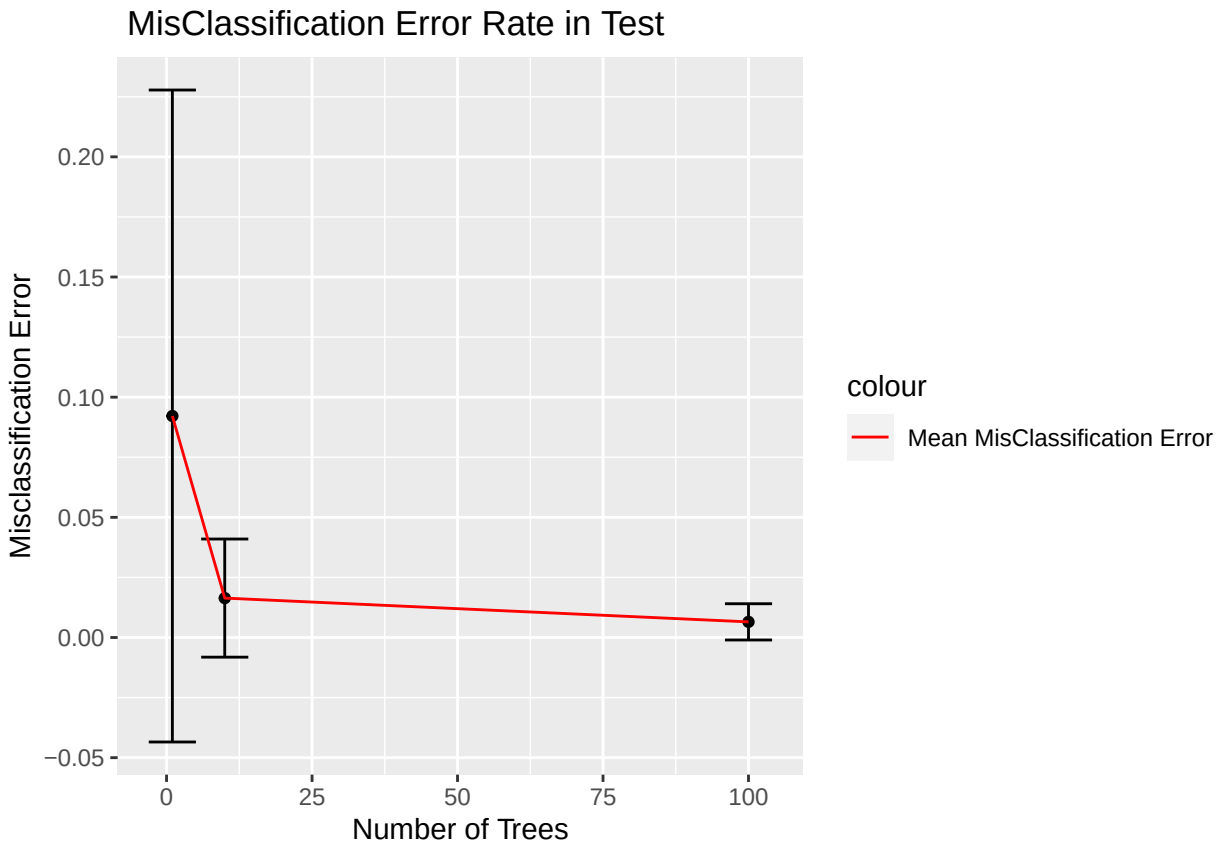
Classifying Y from X1 and X2. Where $Y = \{^{1,\ X1<0.5}_{0,\ other\ wise}$ **nodeSize = 25**

After repeating the above procedure for 1000 training data sets of size 100, following are the observations.

```
# b
Error_b = run_model(1000, 25, condition = "b")
```




```
Mean Classification Error:
  t1_tes_err  t10_tes_err t100_tes_err
    0.092162     0.016392     0.006502

Varriance in  Classification Error:
  t1_tes_err  t10_tes_err t100_tes_err
0.0184002560 0.0006041104 0.0000568088
```

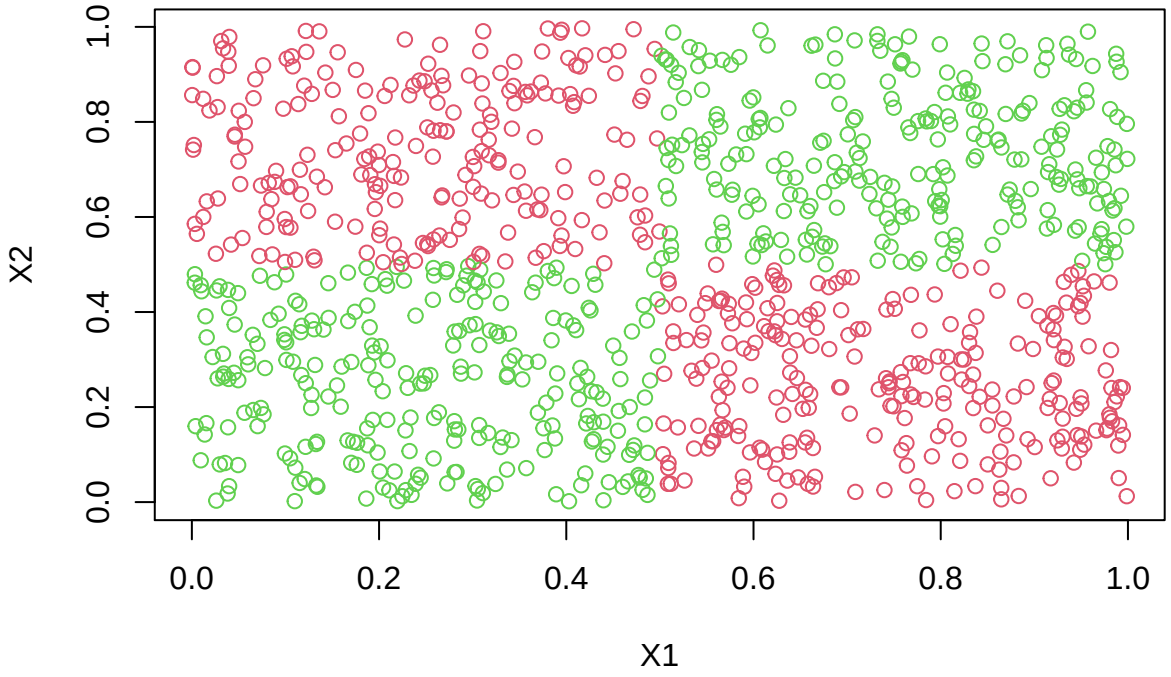


3. Classifying Y from X1 and X2: X1<0.5 & X2<0.5 | X1>0.5 & X2>0.5

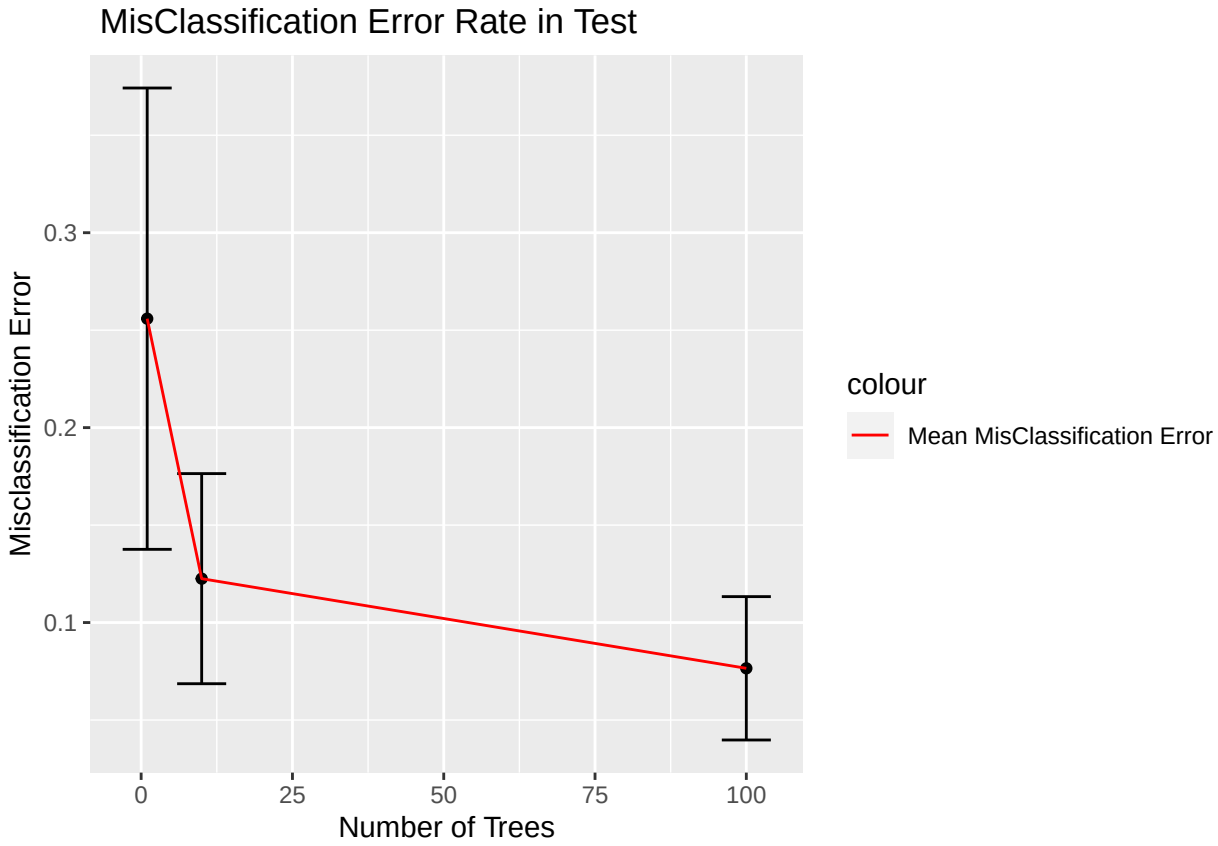
Classifying Y from X1 and X2. Where $Y = \{^{1,\ (X1<0.5\ \&\ X2<0.5)|(X1>0.5\ \&\ X2>0.5)}_{0,\ other\ wise}$ **nodeSize = 12**

After repeating the above procedure for 1000 training data sets of size 100, following are the observations.

```
# c
Error_c = run_model(1000, 12, condition = "c")
```




```
Mean Classification Error:
  t1_tes_err  t10_tes_err t100_tes_err
    0.255896     0.122515     0.076520

Varriance in  Classification Error:
  t1_tes_err  t10_tes_err t100_tes_err
 0.014005038  0.002905137  0.001352594
```



4.

4.1. What happens with the mean and variance of the error rate when the number of trees in the random forest grows?

Random forest algorithm create several subsets of data from training sample chosen randomly with replacement (bagging) with a goal of variance reduction. But this procedure can lead to correlation between the random forest trees. In order to de-correlate, algorithm randomly picks m variables out of pool of variables as candidates at each split. So as the number of trees increases the algorithm generates more training samples and hence makes the model insensitive to changes in the data. Finally classification is performed based on the majority voting.

considering Number of trees = 1:

In this case, random forest model samples single data set from the training data, hence no significant improvement in the variance. Which results in high misclassification error.

In case of Number of trees = 10 and 100:

Random forest model samples 10 and 100 data set with replacement from the training data, hence model gets trained with more combinations of observations, this makes the model more robust. As a result model gives a very less misclassification error and when run such model over multiple iterations, we can observe that error produced in the consecutive iterations are also very less. Hence the mean and Variance of the error rate decreased.

Bagging error:

$error_{bag}(D) = \frac{1}{B^2} \sum_b error^b(D)$

Bagged error $\{error_{bag}(D)\}$is never larger than the individual error $\{error^b(D)\}$.

In addition, if individual errors ($error^b(D)$) are identically distributed with same variance (σ^2) with not or positively correlated (ρ). Then, variance in bagging error is always lesser than the variance of individual error: $\rho\sigma^2 + \frac{1-\rho}{n}\sigma^2 < \sigma^2$

4.2.

The third dataset represents a slightly more complicated classification problem than the first one. Still, you should get better performance for it when using sufficient trees in the random forest. Explain why you get better performance.

In 1st data set ("a"), labels are generated with single condition (*Y=X1<X2*). Where as in 3rd data set ("c"), the labels are generated with multiple conditions (*X1<0.5 & X2<0.5)|(X1>0.5 & X2>0.5*) hence the splits in random forest trees with 3rd data set will be more complicated than that of 1st data set. Therefore the third dataset represents a slightly more complicated classification problem than the first one.

In the first case ("a"), the model runs with *nodeSize = 25* which means there can be minimum of 25 observations in the terminal nodes. Where as in 3rd case ("c) the *nodeSize = 12*, here there can be minimum of 12 observations in the terminal nodes. Setting lower nodeSize values, leads to trees with a larger depth which means that more splits are performed until the terminal nodes. Because of this the case 3 model performs better.

4.3. Why is it desirable to have low error variance ?

Decision trees are sensitive to the data. For a slight difference in training data model will result in different error rate. So it is desirable to have low error variance which depicts that the model is more robust.

Assignment 2. Mixture models

Your task is to implement the EM algorithm for mixtures of multivariate Bernoulli distributions. Use your implementation to show what happens when your mixture model has too few and too many components, i.e. set K=2,3,4 and compare results. Please provide a short explanation as well.

There are 4 steps that need to be implemented in this assignment: E-step, log-likelihood, a break if the log-likelihood doesn't change significantly and the M-step.

To compute the E-step one need to compute the posterior distribution $p\left(z_{nk} \mid \boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}\right)$ for all k and n such as:

$$p\left(z_{nk} \mid \boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}\right) = \frac{p\left(\boldsymbol{z}_{nk}, \boldsymbol{x}_n \mid \boldsymbol{\mu}, \boldsymbol{\pi}\right)}{\sum_k p\left(z_{nk}, \boldsymbol{x}_n \mid \boldsymbol{\mu}, \boldsymbol{\pi}\right)} = \frac{\pi_k p\left(\boldsymbol{x}_n \mid \boldsymbol{\mu}_k\right)}{\sum_k \pi_k p\left(\boldsymbol{x}_n \mid \boldsymbol{\mu}_k\right)}$$

The algorithm need some initial values for $\boldsymbol{\pi}$ and $\boldsymbol{\mu}$ for computing the first iteration of the algorithm and repeats and calculates new estimates for $p\left(z_{nk} \mid \boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}\right)$ each iteration.

Where $p(\boldsymbol{x}_n \mid \boldsymbol{\mu}_k)$ is calculated from a Bernoulli distribution such as:

$$p(\boldsymbol{x}_n \mid \boldsymbol{\mu}_k) = \prod_i \mu_{ki}^{x_i} \left(1 - \mu_{ki}\right)^{(1-x_i)}$$

The code element of the E-step:

```
# 2. E-step
E_step <- function(K,N,pi,mu,x,z){
for(k in 1:K){
  for(n in 1:N){
  z[n,k] <- pi[k]* prod( ( mu[k,]^x[n,] ) * (1-mu[k,])^(1-x[n,]))
  }
}
z <- z/rowSums(z)
return(z)
}
```

To calculate the log-likelihood for each iteration the formula is given below:

$$\log p\left(\{\boldsymbol{x}_n, \boldsymbol{z}_n\} \mid \boldsymbol{\mu}, \boldsymbol{\pi}\right) = \sum_n \sum_k z_{nk} \left[\log \pi_k + \sum_i \left[x_{ni} \log \mu_{ki} + (1 - x_{ni}) \log\left(1 - \mu_{ki}\right)\right]\right]$$

The code element of the log-likelihood:

```
# 2. log likelihood
log_likelihood <- function(N, K,llik, it, z, pi, x, mu){
for(n in 1:N ){
  for(k in 1:K){
    llik[it] <- llik[it] + z[n,k] *  (log(pi[k]) + sum(x[n,] * log(mu[k,]) + (1- x[n,])*log(1- mu[k,])))
  }
}
return(llik)
}
```

In the EM algorithm should compute and repeat the iteration process until $\boldsymbol{\pi}$ and $\boldsymbol{\mu}$ does not change. This can be done by calculating the absolute difference between the log-likelihood of the previous iteration and the current, where a minimum change is set.

The code element of the minimum change of log-likelihood:

```
# 2. break step
if(it > 1 ){
  if(( abs(llik[it]- llik[it-1])  < min_change)){
  break
  }
}
```

To calculate the new estimation for $\boldsymbol{\pi}$ $\boldsymbol{\mu}$ for each iteration one can implement the following calculation in the M-Step.

$$\pi_k^{ML} = \frac{\sum_n p\left(z_{nk} \mid \mathbf{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}\right)}{N}$$
$$\mu_{ki}^{ML} = \frac{\sum_n x_{ni} p\left(z_{nk} \mid \mathbf{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}\right)}{\sum_n p\left(z_{nk} \mid \mathbf{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}\right)}$$

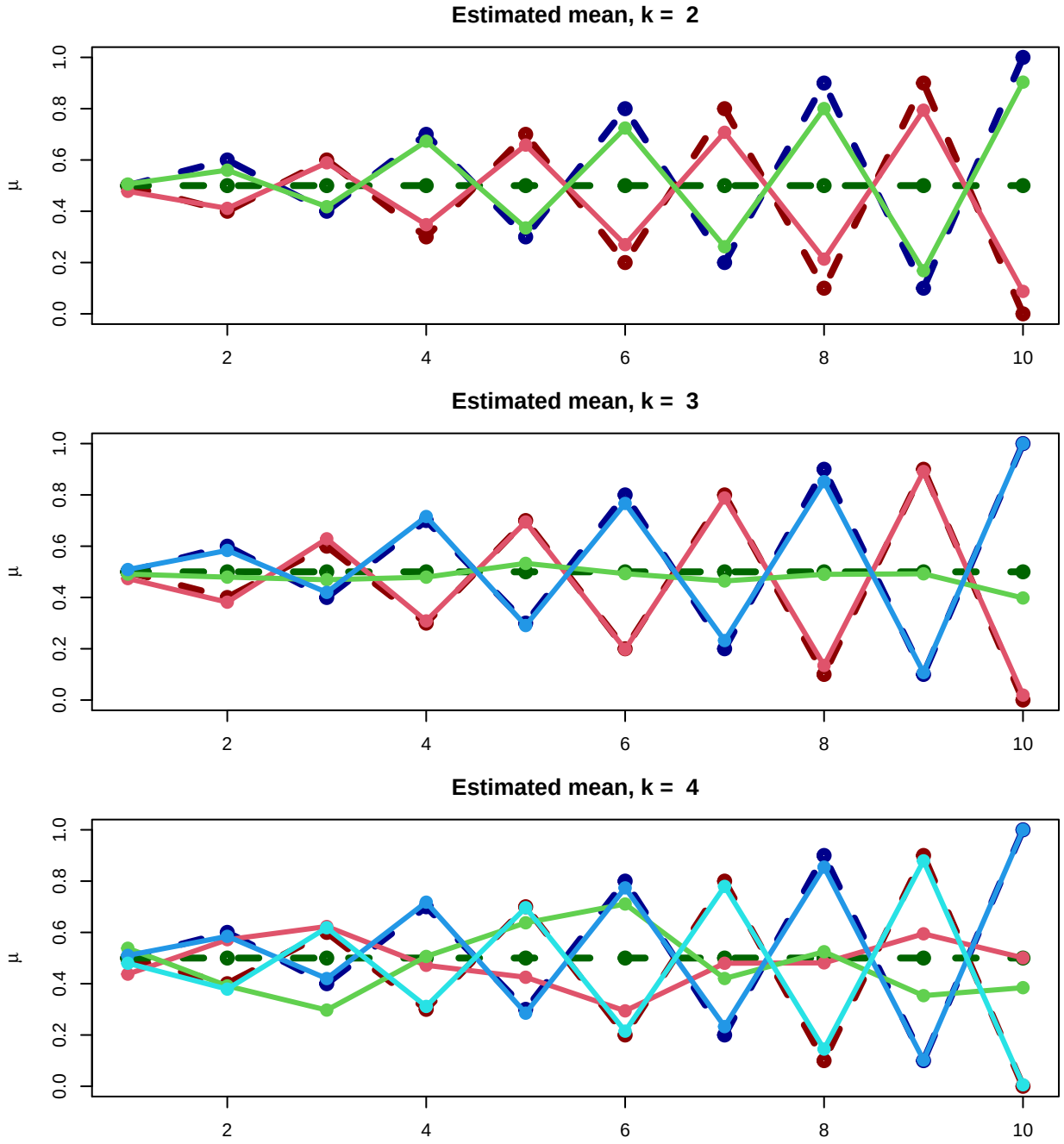
The code element of the M-step:

```
# 2. M-step
M_step <- function(z,N,x){
pi <- colSums(z)/N
mu <- t(z) %*% x /colSums(z)
return(list(pi=pi,mu=mu))
}
```

The whole Expectation Maximization Algorithm for multivariate Bernoulli is then put together but inside a function that depends on the number of K.

```
# 2. EM_algorithm
EM_algorithm <- function(K){
# CREATE DATA STEP
set.seed(1234567890)
max_it <- 100; min_change <- 0.1; N=1000; D=10
x <- matrix(nrow=N, ncol=D);true_pi <- vector(length = 3); true_mu <- matrix(nrow=3, ncol=D)
true_pi=c(1/3, 1/3, 1/3)
true_mu[1,]=c(0.5,0.6,0.4,0.7,0.3,0.8,0.2,0.9,0.1,1)
true_mu[2,]=c(0.5,0.4,0.6,0.3,0.7,0.2,0.8,0.1,0.9,0)
true_mu[3,]=c(0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5)
# Producing the training data
for(n in 1:N) {
  k <- sample(1:3,1,prob=true_pi)
  for(d in 1:D){
    x[n,d] <- rbinom(1,1,true_mu[k,d])
  }
}
z <- matrix(nrow=N, ncol=K) # fractional component assignments
pi <- vector(length = K) # mixing coefficients
mu <- matrix(nrow=K, ncol=D) # conditional distributions
llik <- vector(length = max_it) # log likelihood of the EM iterations

# Random initialization of the parameters
pi <- runif(K,0.49,0.51);pi <- pi / sum(pi)
for(k in 1:K) {
mu[k,] <- runif(D,0.49,0.51)
}
for(it in 1:max_it) {
Sys.sleep(0.5)
# E-step: Computation of the fractional component assignments
z <- E_step(K=K,N=N,pi=pi,mu=mu,x=x,z=z)
#Log likelihood computation.
llik <- log_likelihood(N=N,K=K,llik=llik,it=it,z=z,pi=pi,x=x,mu=mu)

cat("iteration: ", it, "log likelihood: ", llik[it], "\n")
flush.console()
# Stop if the log likelihood has not changed significantly
if(it > 1 ){
  if(( abs(llik[it]- llik[it-1])  < min_change)){
  break
  }
}
#M-step: ML parameter estimation from the data and fractional component assignments
pi <- M_step(z=z,N=N,x=x)$pi
mu <- M_step(z=z,N=N,x=x)$mu
}
invisible(list(true_mu = true_mu, true_pi=true_pi,
          estimated_pi=pi,estimated_mu=mu, likelihood=llik,x=x,z=z))
}
```

The graphs below shows different values at k for the EM-algorithm. Dashed lines are the true distribution and the complete line is the estimated means of k distributions.

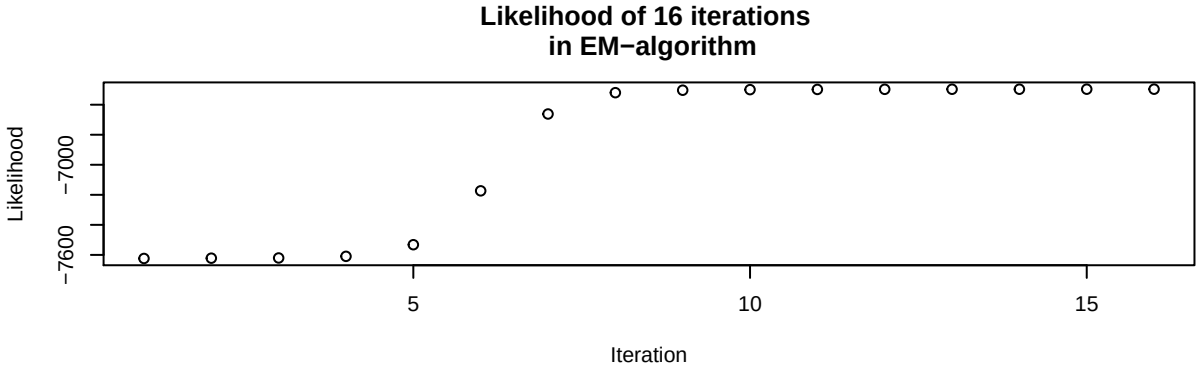


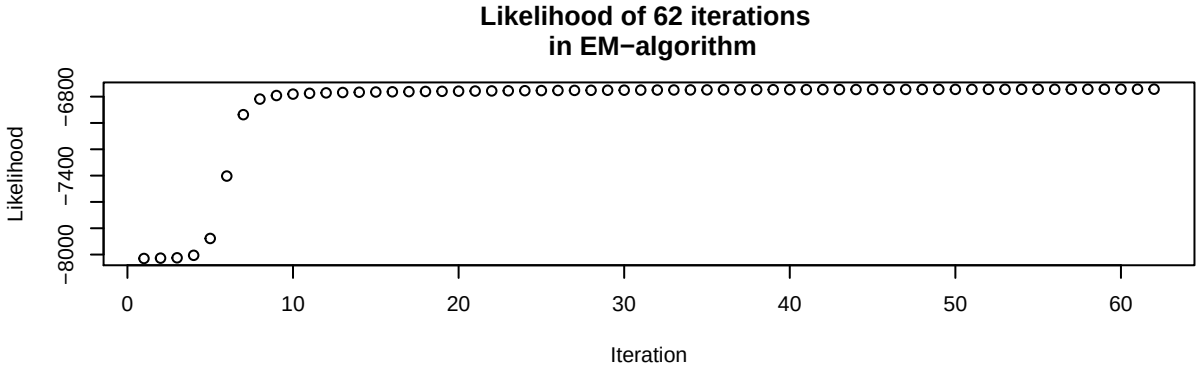
From the plots above different values at k are visualized. As seen when two distributions are estimated (k=2) the μ EM-algorithm tries to fit the ten different means but for only two distributions which differ from the true number of distributions (k=3), in this case one can see that the EM-algorithm appears to underfit compared to the true distributions since one distribution is missing.

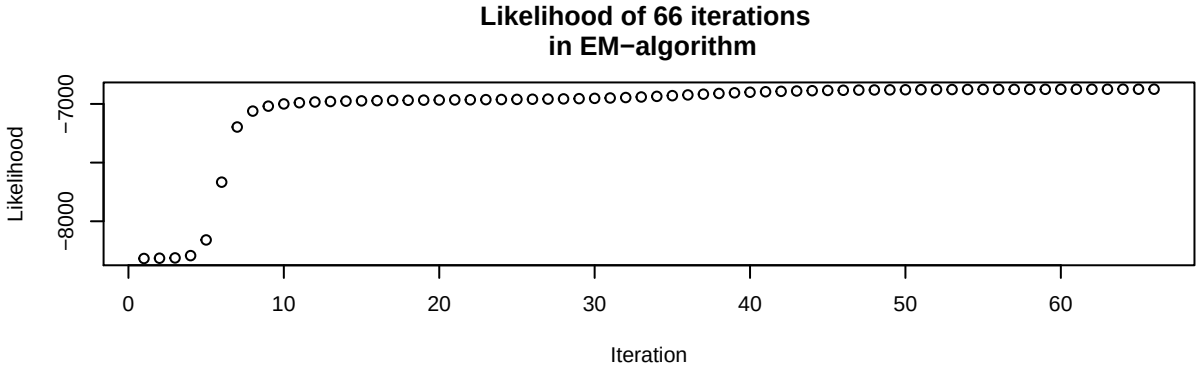
As a comparison when three distributions are estimated (k=3) for the EM-algorithm, one can see that a better fit occurs that appear to fit close to the true means, this makes sense since the true number of distributions are three.

When introducing four distributions for EM-algorithm overfitting occurs, where all true means of the distribution does not get good representation by the new estimation and the EM-algorithm tries to estimate new

distributions that doesn't exist.







K	LogLikelihood
2	-6496.662
3	-6743.326
4	-6874.497

Above the log-likelihood is visualized for each iteration. The table shows the value of the log-likelihood at the last iteration for different k. Although the log-likelihood seems to be at the highest value for $k = 2$ one cannot draw the solution only based on value of the log-likelihood which of the k is the best fit when not taking the consideration of parameters. As seen in the previous plot $k = 3$ which is the true number of distribution will be the desired number of k. Another aspect when comparing models are the starting values, these EM algorithms tend to be sensitive of what starting values, which in this way assignment differ if the seed is changed.

Assignment 3 - High-dimensional methods

Background

In this assignment we are going to use nearest shrunken centroid (NSC) classification method to classify cells as their proper types depending on which genes are expressed in the cells. The data is interesting as the number of features are almost seven times as many as the number of observations (300 x 2086). This is a task where common classification techniques are not appropriate, hence we need to use some regularization, here in the form of the NSC, where the classwise means are shrunk towards the overall mean for each non-significant feature (see Hastie et al. (2001). *The Elements of Statistical Learning*).

The distances from each class mean to the overall mean are standardized by subtracting the overall mean from each class mean and dividing by the pooled within-class standard deviation s_j (with a small correction to protect against extreme values using a small constant s_0).

$$d_{kj} = \frac{\bar{x}_{kj} - \bar{x}_j}{m_k(s_j + s_0)}$$

The shrinkage is done using soft thresholding, that is a shrinkage where a threshold parameter Δ is subtracted from each of the (absolute) values. This parameter can be selected using cross-validation. The resulting values that are negative are set to zero.

$$d'_{kj} = \text{sign}(d_{kj})(|d_{kj}| - \Delta)_+$$

To attain the new shrunken class centroids the following calculation is made:

$$\bar{x}'_{kj} = \bar{x}_j + m_k(s_j + s_0)d'_{kj}$$

These new values can then be used by a linear discriminant. As we use a threshold, some features may be set to zero and discarded, much like in the lasso, resulting in a simpler model.

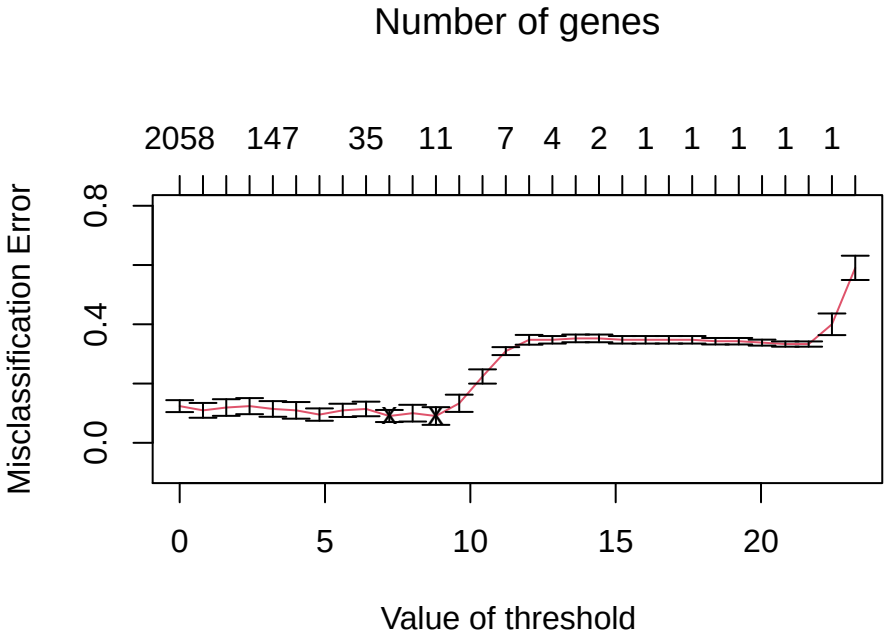
1. Nearest Shrunken Centroid Classification using Cross-Validation

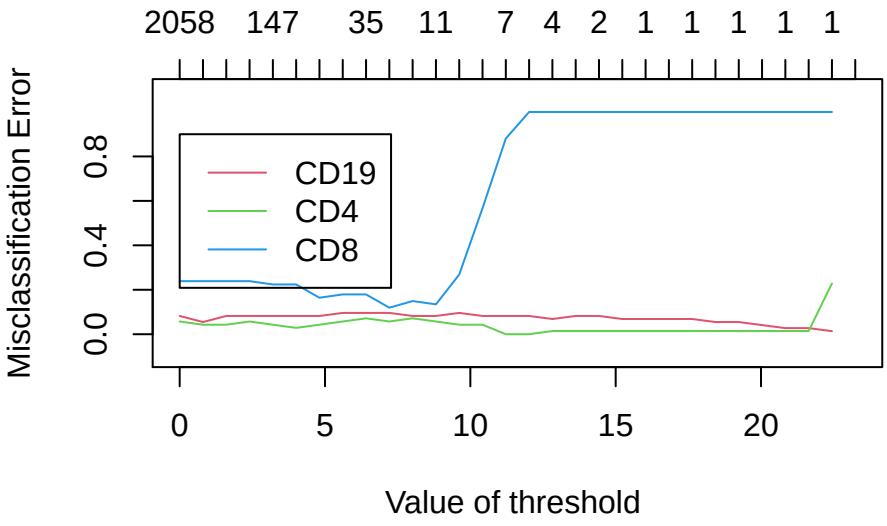
We are supposed to do NSC classification of the gene data. Data is first divided into training and test sets with proportions 70/30 without scaling and the model is then trained on the training data using package `pamr`.

Threshold and Classification Error The optimal value of the threshold is either 7.213 or 8.816 (both give the same number of errors). As less shrinkage could be argued to be less complex (the model is less sensitive), we have decided to use the former value for the threshold. In the output below one can see how the number of non-zero features are decreasing as the threshold value increases. The number of misclassifications seem to reduce a first, but then increase heavily as only a few features remain.

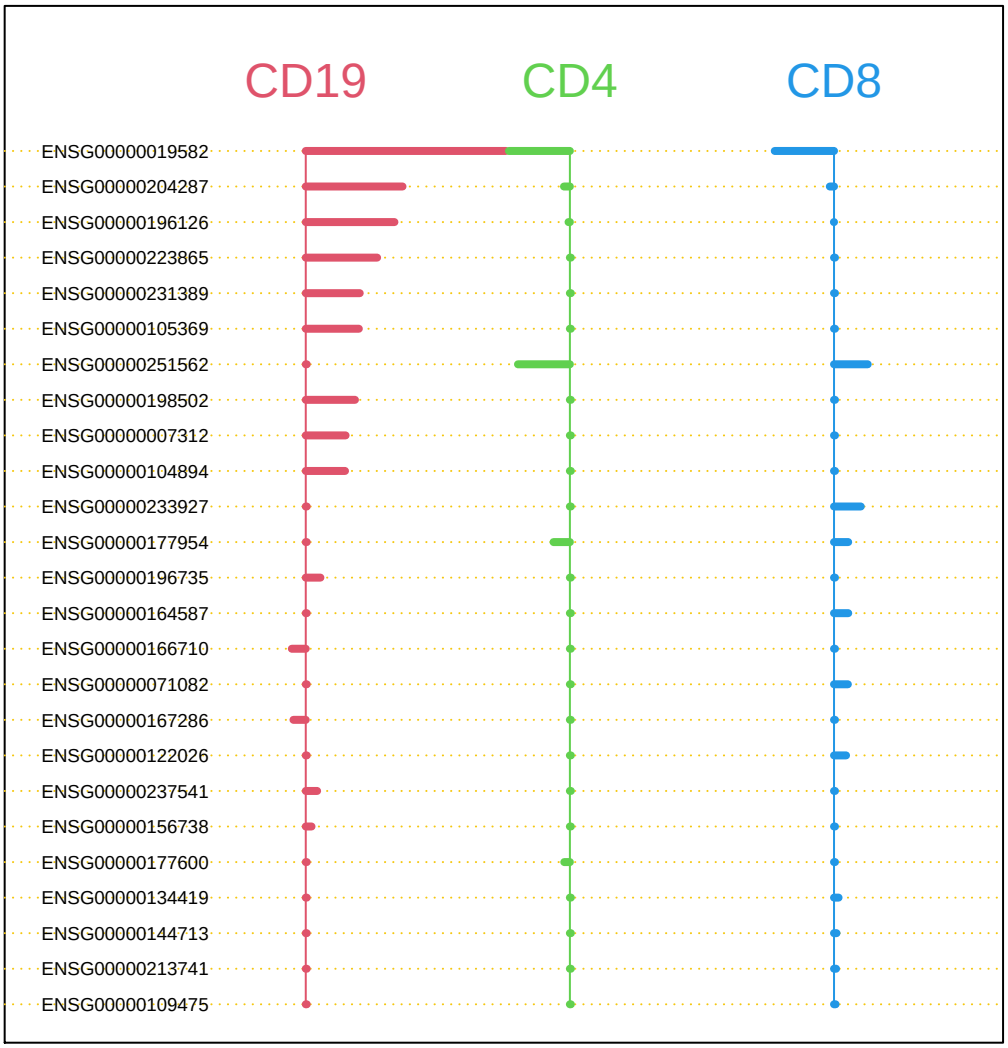
```
Call:
pamr.cv(fit = m1, data = training_data)
   threshold nonzero errors
1   0.000    2058    26
2   0.801    1040    23
3   1.603     374    25
4   2.404     226    26
5   3.206     147    24
6   4.007     102    23
7   4.809      78    20
8   5.610      54    23
9   6.412      35    24
10  7.213      25    19
11  8.014      19    21
12  8.816      11    19
13  9.617      10    28
14 10.419       8    47
15 11.220       7    65
16 12.022       4    73
17 12.823       4    73
18 13.624       3    74
19 14.426       2    74
20 15.227       1    73
21 16.029       1    73
22 16.830       1    73
23 17.632       1    73
24 18.433       1    72
25 19.235       1    72
26 20.036       1    71
27 20.837       1    70
28 21.639       1    70
29 22.440       1    84
30 23.242       0    124
```

Visual Inspection of Errors Inspecting the error rate and threshold visually reveals that it is the CD8 cell type that seems to be misclassified more often as the threshold increases.





Centroid Plot The centroid plot below shows how, for each cell type, the gene expression differs between each cell type's centroid and the overall centroid or the average expression. This means that not all cell type can have a positive (or negative) expression. The plot shows all the genes that survived the shrinkage, 25 in this case. "Surviving" in this case means that for at least one of the cell types, the shrunken centroid is non-zero. The CD19 cells are markedly different from the CD4 and CD8 cells, indicating the difference between these cells are greater.



Surviving Genes Below are the genes that survive the shrinkage, presented in a table format.

```
      id  name            CD19-score CD4-score CD8-score
 [1,] 2   ENSG00000019582 1.5153     -0.4617   -0.4489
 [2,] 15  ENSG00000204287 0.7328     -0.0448   -0.032
 [3,] 31  ENSG00000196126 0.6715     -0.0105   -0.001
 [4,] 32  ENSG00000223865 0.5403     0         0
 [5,] 37  ENSG00000231389 0.4089     0         0
 [6,] 138 ENSG00000105369 0.4019     0         0
 [7,] 1   ENSG00000251562 0          -0.3933   0.2553
 [8,] 90  ENSG00000198502 0.3727     0         0
 [9,] 172 ENSG00000007312 0.3016     0         0
[10,] 126 ENSG00000104894 0.2974     0         0
[11,] 79  ENSG00000233927 0          0         0.2028
[12,] 11  ENSG00000177954 0          -0.1239   0.1074
[13,] 110 ENSG00000196735 0.1109     0         0
[14,] 21  ENSG00000164587 0          0         0.1078
[15,] 3   ENSG00000166710 -0.1057    0         0
[16,] 50  ENSG00000071082 0          0         0.1051
[17,] 207 ENSG00000167286 -0.0933    0         0
[18,] 29  ENSG00000122026 0          0         0.0914
[19,] 192 ENSG00000237541 0.0862     0         0
[20,] 309 ENSG00000156738 0.0438     0         0
[21,] 28  ENSG00000177600 0          -0.0426   0
[22,] 38  ENSG00000134419 0          0         0.0346
[23,] 18  ENSG00000144713 0          0         0.0189
[24,] 127 ENSG00000213741 0          0         0.0156
[25,] 36  ENSG00000109475 0          0         0.0119
```

```
215313237138190172126791111021350207291923092838181273 6ENSG00000019582ENSG00000204287ENSG00000196126ENS
```

2. Most Contributing Genes

The two most contributing genes are given at the top of the previous table. These are, together with their synonyms:

- ENSG00000019582 - CD74/DHLAG
- ENSG00000204287 - HLA-DRA1

Both of these genes seem to appear in immune system cells according to https://panglaodb.se/markers.html (both B and T cells are lymphocytes).

Test Error of NSC Model The confusion matrix and calculated error for the test set are presented below.

```
NSC Confusion Matrix:
```

```
      Predicted
True   CD19 CD4 CD8
  CD19   27   0   0
  CD4     0  26   4
  CD8     0   6  27
```

```
NSC Error:

0.111

Number of non-zero parameters:

25
```

3. Elastic Net and Support Vector Machines

Elastic Net The elastic net technique combines L1 and L2 regularization, allowing for feature parameters to be reduced to zero. Here we are using elastic net with multinomial response and $\alpha = 0.5$. The penalty factor λ is selected through cross-validation. Package `glmnet` was used for model fitting with training data and prediction with test data.

```
Elastic Net Confusion Matrix:

      Predicted
True   CD19 CD4 CD8
  CD19   27   0   0
  CD4     0  29   1
  CD8     1   3  29

Elastic Net Error:

0.056

Number of non-zero parameters:

54
```

Support Vector Machine Here we are using support vector machines (SVM) to do the same classification task. SVM's can be used for both linear and non-linear classification, the latter by using the "kernel trick". In short, the SVM fits a decision boundary as a hyperplane to maximize the margin between classes. In this case we will be using the vanilladot kernel (linear) from the `ksvm` function from the package of the same name.

```
 Setting default kernel parameters

SVM Confusion Matrix:

      Predicted
True   CD19 CD4 CD8
  CD19   27   0   0
  CD4     0  30   0
  CD8     0   2  31

SVM Error:
```

```
0.022

Number of support vectors:

74
```

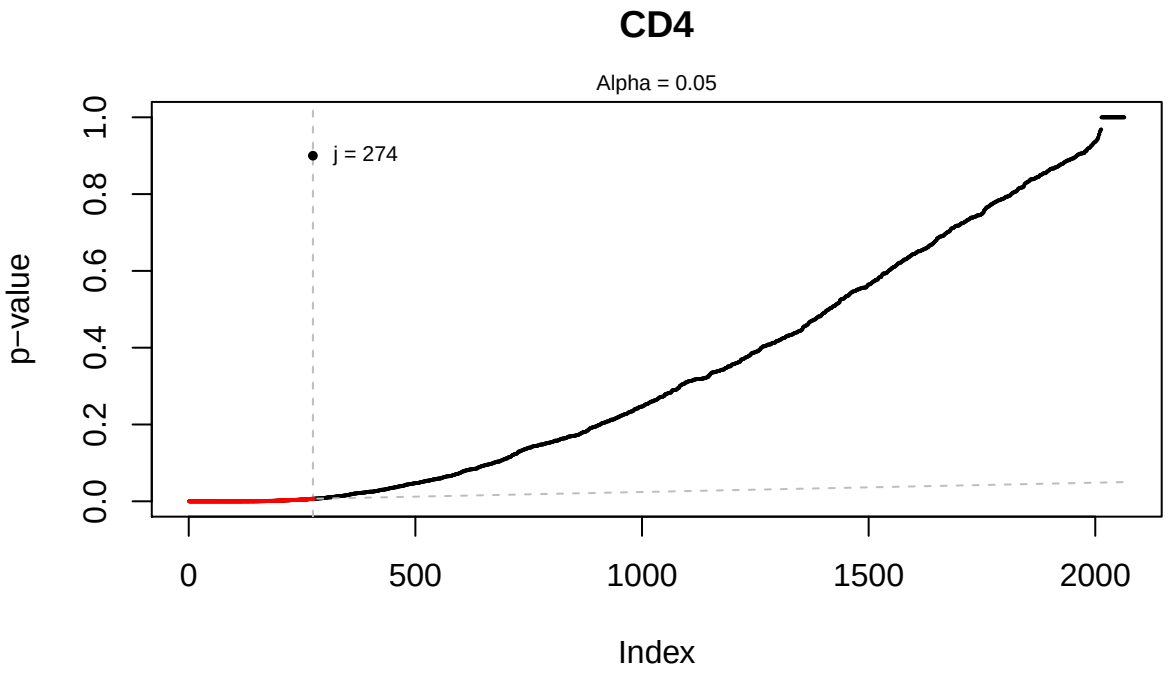
Model Comparison From the table below we can see that the best method regarding the error rate is SVM, but it also seems to be more complex with more support vectors used than the number of non-zero features in the other methods. Elastic net is the middle option here with a reasonable misclassification rate and a manageable number of features. When considering the time taken to run the different methods, the SVM comes out on top. Altogether the SVM seems to be the a great selection in this case.

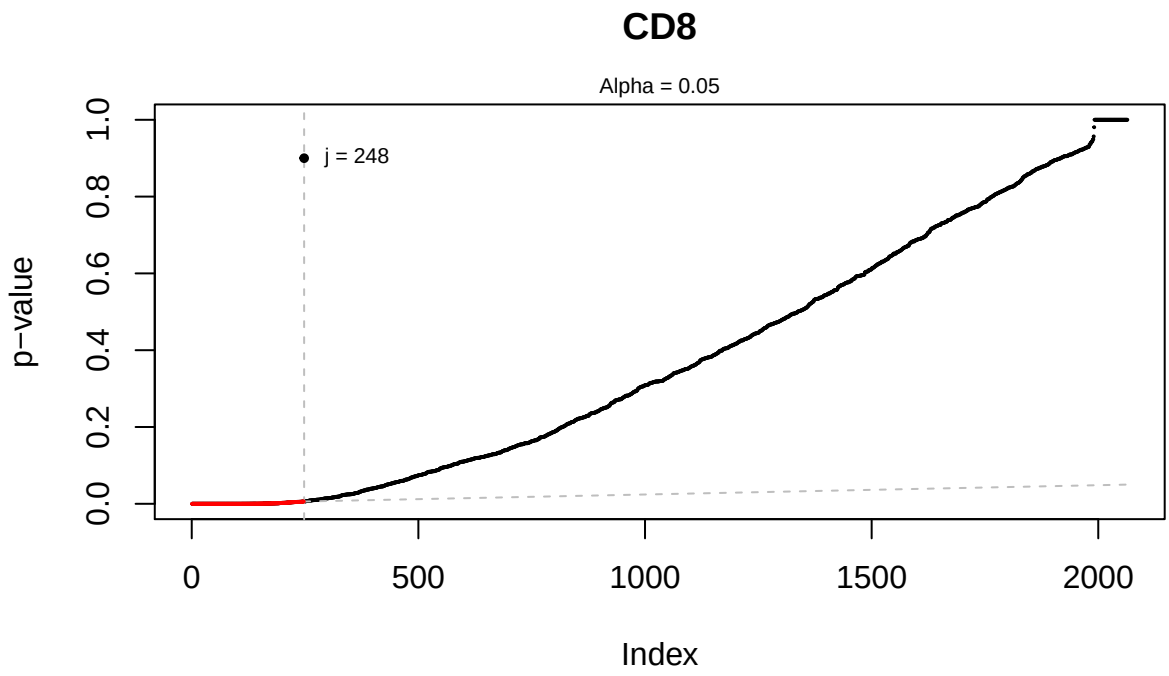
	Error	Features	Runtime
NSC	0.111	25	1.221
Elastic net	0.056	54	4.964
SVM	0.022	74	0.341

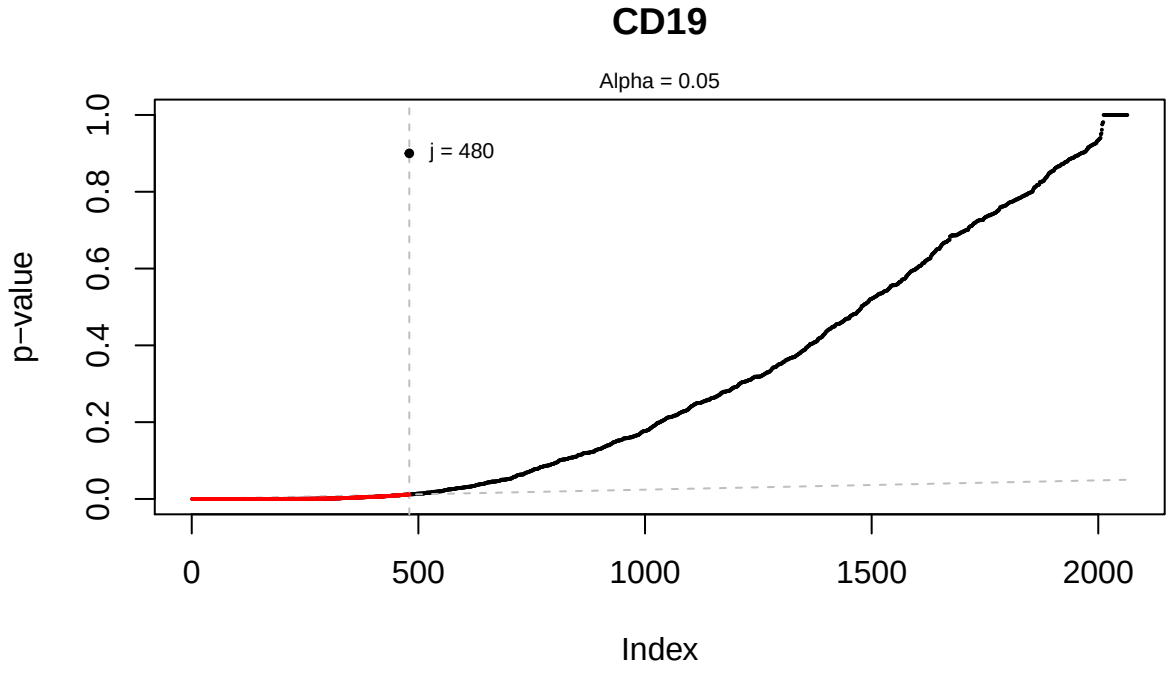
4. Benjamini-Hochberg method

For the final part of the assignment, we are going to use the Benjamini-Hochberg (BH) method to test each cell type against all others using all the features at hand (here we use t-tests). This should allow us to find the features that can discriminate well. The BH method allows us to fix a desired false discovery rate (FDR), denoted α, and utilise that it has been shown that the true FDR is smaller than α given that we order our p-values in ascending order and reject all null hypotheses that have p-values smaller than the BH rejection threshold.

Three plots are presented below, one for each cell type, using $\alpha = 0.05$ The p-values marked in red are those that are low enough to reject their corresponding null hypotheses, i.e. there is a statistically significant difference from the other cell types. It is clear that CD19 gets many more rejected hypotheses than the other types, indicating that it is easier to find features (genes) that can discriminate this cell type well from the others. It is interesting to note that this number of genes are much higher than in our previous attempts with NSC, SVM and elastic net.




CD8
Alpha = 0.05
j = 248
p−value
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
Index


CD19
Alpha = 0.05
j = 480
p−value
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
Index

Code Appendix

```
knitr::opts_chunk$set(echo = TRUE)
knitr::opts_chunk$set(comment = NA)
knitr::opts_chunk$set(warning=FALSE)
knitr::opts_chunk$set(message=FALSE)

##  Libraries Used

library("randomForest")

library("ggplot2")

library("dplyr")

## Methods Section

#  get_data: Method to generate Train and Test data as per the given condition

get_data <- function(condition){

  x1_train<-runif(100)

  x2_train<-runif(100)


  if (condition == "a"){

# 1. Classifying Y from X1 and X2: X1<X2
    #  using the condition (x1<x2)

    train<-data.frame(x1 = x1_train,x2 = x2_train,
                      y = as.factor(as.numeric(x1_train<x2_train)))

    test<-data.frame(x1 = x1_test,x2 = x2_test,
                     y = as.factor(as.numeric(x1_test<x2_test)))


  }else if (condition == "b"){

# 2. Classifying Y from X1 and X2: X1<0.5
    #  using the condition (x1<0.5)

    train<-data.frame(x1 = x1_train,x2 = x2_train,
                      y = as.factor(as.numeric(x1_train<0.5)))

    test<-data.frame(x1 = x1_test,x2 = x2_test,
                     y = as.factor(as.numeric(x1_test<0.5)))


  }else if (condition == "c"){

# 3. Classifying Y from X1 and X2: X1<0.5 & X2<0.5 | X1>0.5 & X2>0.5
```

```
    #  using the condition (x1<&x2 != 0.5)

    train<-data.frame(x1 = x1_train,x2 = x2_train,
                      y = as.factor(as.numeric((x1_train<0.5 & x2_train <0.5) |

                                               (x1_train>0.5 & x2_train>0.5))))

    test<-data.frame(x1 = x1_test,x2 = x2_test,
                     y = as.factor(as.numeric((x1_test<0.5 & x2_test <0.5) |

                                              (x1_test>0.5 & x2_test>0.5))))
  }

  return (list(train = train, test = test))

}##  End Method get_data


#randomForest_: Method to reuse the randomForest for multiple train and test datasets.
# Returns the model object, miscalssification error rates in test and train data.

randomForest_ <- function(ntree, nodesize, train, test){


   rf <- randomForest(y~(x1+x2),

                      data = train,

                      ntree = ntree, nodesize = nodesize,

                      #mtry = 2,

                      keep.forest = TRUE

                      )

  #  Calculating Misclassification Error
  misclass_train = 1- sum(diag(table(train$y, predict(rf, train))))/nrow(train)

  misclass_test = 1- sum(diag(table(test$y, predict(rf, test))))/nrow(test)

  return(

    list(rf = rf,

         misclass_train = misclass_train,

         misclass_test = misclass_test

         )
  )
} ## End Method randomForest_
```

```r
# plot_error: Method to plot mean and variances of misclassification error
plot_error <-function (data){

  cat("Mean Classification Error: \n")

  data <- data %>% select(

    t1_tr_err,t10_tr_err,t100_tr_err,
    t1_tes_err,t10_tes_err,t100_tes_err
  )

  print (colMeans(data[,4:6]))

  # Calculating Error Mean
  df_mean_error <- data.frame(x = colMeans(data),

              DataSet = c("train", "train", "train", "test", "test", "test"),

              id = c(1,10,100, 1, 10, 100)
             )

  #  Getting Variance
  var_ <- c()

  for(i in colnames(data)){

    var_[i]<-c(i = var(data[i]))

  }

  df_mean_error$var <- var_

  cat("\nVarriance in  Classification Error: \n")

  print(var_[4:6])


   df_mean_error = dplyr::filter(df_mean_error, DataSet == "test")

   p1 = ggplot()+geom_point(aes(x = df_mean_error$id, y = df_mean_error$x))+
        geom_errorbar(aes(x = df_mean_error$id,
                          ymin = df_mean_error$x-sqrt(df_mean_error$var),
                          ymax = df_mean_error$x+sqrt(df_mean_error$var)
                          ))+
        geom_line(aes(x=df_mean_error$id ,y=df_mean_error$x,col="red"))+
        labs( y = "Misclassification Error",

          x = "Number of Trees",

          title = " MisClassification Error Rate in Test")+
       scale_color_manual(labels = c("Mean MisClassification Error"), values = c("red"))
```

```
   print(p1)

 #gridExtra::grid.arrange(p1,p2, p3)
}# End plot_error


# run_model: Method executes randomforest over 1000 datasets.

run_model <- function(N, nodesize, condition){

    i = 1

    df_error = data.frame(
                  t1_tr_err = as.numeric(), t1_tes_err  = as.numeric(),
                  t10_tr_err= as.numeric(), t10_tes_err  = as.numeric(),
                  t100_tr_err  = as.numeric(), t100_tes_err  = as.numeric()
                        )

    while(i<=N){

      data = get_data(condition)

      train = data$train

      test = data$test

      if (i==1){

        plot(test$x1,test$x2, col=(as.numeric(test$y)+1),main = "Test Data",
             xlab = "X1",ylab = "X2")

      }

      rf_object_t1 <- randomForest_(1,nodesize, train, test)

      rf_object_t10 <- randomForest_(10,nodesize, train, test)

      rf_object_t100 <- randomForest_(100,nodesize, train, test)

      df_error <- (rbind(df_error,c(

                  rf_object_t1$misclass_train, rf_object_t1$misclass_test,
                  rf_object_t10$misclass_train, rf_object_t10$misclass_test,
                  rf_object_t100$misclass_train, rf_object_t100$misclass_test)
                    )
                  )
      i=i+1

    }

    colnames(df_error) <- c("t1_tr_err","t1_tes_err", "t10_tr_err",
                            "t10_tes_err","t100_tr_err","t100_tes_err"
                            )
```

```
    #  graphical representation of mean and variances of misclassification
    #error with different number of trees and nodes
    plot_error(df_error)

    return(

      list(df_error = df_error,

           rf_1 = rf_object_t1,

           rf_10 = rf_object_t10,

           rf_100 = rf_object_t100)

        )
} ## End un_model

##  End of Methods Section
## Defining Test Data

set.seed(1234)

x1_test<-runif(1000)

x2_test<-runif(1000)

#a
Error_a = run_model(1000, 25, condition = "a")
# b
Error_b = run_model(1000, 25, condition = "b")
# c
Error_c = run_model(1000, 12, condition = "c")

#########################  End Random Forest Code  ##############################
# 2. E-step
E_step <- function(K,N,pi,mu,x,z){
for(k in 1:K){
  for(n in 1:N){
  z[n,k] <- pi[k]* prod( ( mu[k,]^x[n,] ) * (1-mu[k,])^(1-x[n,]))
  }
}
z <- z/rowSums(z)
return(z)
}
# 2. log likelihood
log_likelihood <- function(N, K,llik, it, z, pi, x, mu){
for(n in 1:N ){
  for(k in 1:K){
    llik[it] <- llik[it] + z[n,k] *  (log(pi[k]) + sum(x[n,] * log(mu[k,]) + (1- x[n,])*log(1- mu[k,])))
  }
}
return(llik)
}
```

```r
# 2. break step
if(it > 1 ){
  if(( abs(llik[it]- llik[it-1])  < min_change)){
  break
  }
}
# 2. M-step
M_step <- function(z,N,x){
pi <- colSums(z)/N
mu <- t(z) %*% x /colSums(z)
return(list(pi=pi,mu=mu))
}
# 2. EM_algorithm
EM_algorithm <- function(K){
# CREATE DATA STEP
set.seed(1234567890)
max_it <- 100; min_change <- 0.1; N=1000; D=10
x <- matrix(nrow=N, ncol=D);true_pi <- vector(length = 3); true_mu <- matrix(nrow=3, ncol=D)
true_pi=c(1/3, 1/3, 1/3)
true_mu[1,]=c(0.5,0.6,0.4,0.7,0.3,0.8,0.2,0.9,0.1,1)
true_mu[2,]=c(0.5,0.4,0.6,0.3,0.7,0.2,0.8,0.1,0.9,0)
true_mu[3,]=c(0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5)
# Producing the training data
for(n in 1:N) {
  k <- sample(1:3,1,prob=true_pi)
  for(d in 1:D){
    x[n,d] <- rbinom(1,1,true_mu[k,d])
  }
}
z <- matrix(nrow=N, ncol=K) # fractional component assignments
pi <- vector(length = K) # mixing coefficients
mu <- matrix(nrow=K, ncol=D) # conditional distributions
llik <- vector(length = max_it) # log likelihood of the EM iterations

# Random initialization of the parameters
pi <- runif(K,0.49,0.51);pi <- pi / sum(pi)
for(k in 1:K) {
mu[k,] <- runif(D,0.49,0.51)
}
for(it in 1:max_it) {
Sys.sleep(0.5)
# E-step: Computation of the fractional component assignments
z <- E_step(K=K,N=N,pi=pi,mu=mu,x=x,z=z)
#Log likelihood computation.
llik <- log_likelihood(N=N,K=K,llik=llik,it=it,z=z,pi=pi,x=x,mu=mu)

cat("iteration: ", it, "log likelihood: ", llik[it], "\n")
flush.console()
# Stop if the log likelihood has not changed significantly
if(it > 1 ){
  if(( abs(llik[it]- llik[it-1])  < min_change)){
  break
  }
```

```
}
#M-step: ML parameter estimation from the data and fractional component assignments
pi <- M_step(z=z,N=N,x=x)$pi
mu <- M_step(z=z,N=N,x=x)$mu
}
invisible(list(true_mu = true_mu, true_pi=true_pi,
          estimated_pi=pi,estimated_mu=mu, likelihood=llik,x=x,z=z))
}

## Plotting function
plot_mu <- function(KX){
par(mar=c(2,4,3,1))
plot(KX$true_mu[1,], type="o",lty = 2,lwd=4, col="dark blue", ylim=c(0,1),ylab=expression(mu), main= pas
points(KX$true_mu[2,], type="o",lty = 2,lwd=4, col="dark red")
points(KX$true_mu[3,], type="o",lty = 2,lwd=4, col="dark green")
for(i in 1:nrow(KX$estimated_mu)){
points(KX$estimated_mu[i,], type="o",lty = 1,lwd=3, col=1+i)
}
}

plot_likelihood <- function(KX){
plot(KX$likelihood[KX$likelihood!=0],xlab="Iteration",ylab="Likelihood", main=paste("Likelihood of",sum
}

KX <- lapply(2:4, function(k){EM_algorithm(K=k)})
par(mfrow=c(3,1),mar=c(0,0,0,0))
plot_mu(KX[[1]])
plot_mu(KX[[2]])
plot_mu(KX[[3]])
par(mfrow=c(3,1))
plot_likelihood(KX[[1]])
plot_likelihood(KX[[2]])
plot_likelihood(KX[[3]])

knitr::kable(data.frame(K=c(2,3,4) ,
  LogLikelihood= c(KX[[1]]$likelihood[ sum( KX[[1]]$likelihood != 0)],
    KX[[2]]$likelihood[ sum( KX[[2]]$likelihood != 0)],
    KX[[3]]$likelihood[ sum( KX[[3]]$likelihood != 0)]) ))
# Assignment 3 - Setup
knitr::opts_chunk$set(echo = TRUE)

# Set working directory here
# setwd(dir = "C:/")

# Dependencies
library(pamr)
library(glmnet)
library(kernlab)
# Assignment 3 - Part 1
# Reading data from file and making initial data prep
genes0 <- read.csv("geneexp.csv", row.names = "X", stringsAsFactors = TRUE)
genes <- as.data.frame(genes0[,1:ncol(genes0)-1])
genes[is.nan(as.matrix(genes))] <- 0
```

```r
# Setting cell type as factor
genes <- as.data.frame(cbind("CellType" = genes0$CellType, genes))

# Data partioning into train and test
n=dim(genes)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.7))
train=genes[id,]
test=genes[-id,]

# Splitting target and features for training set
x_train <- t(train[,2:ncol(train)])
y_train <- train[, 1]

# Splitting target and features for test set
x_test <- t(test[,2:ncol(test)])
y_test <- test[, 1]

# Making lists of data to use with pamr package
training_data <- list("x" = x_train,
                      "y" = y_train,
                      geneid = as.character(1:nrow(x_train)),
                      genenames = rownames(x_train))

test_data <- list("x" = x_test,
                  "y" = y_test,
                  geneid = as.character(1:nrow(x_test)),
                  genenames = rownames(x_test))

# Training NSC model
m1 <- pamr.train(data = training_data)

# Using cross-validation to find optimal model parameter
# Also timing the time taken to run the operation for later comparison
ptime <- proc.time()
set.seed(12345)
m2 <- pamr.cv(m1, training_data)
ptime_nsc <- proc.time()-ptime

# The by cross-validation process for the threshold value
message(print(m2))
# Plotting the cross-validation data
pamr.plotcv(m2)

# Finding the threshold that minimises error
min_cv <- m2$threshold[which.min(m2$error)]

# Centroid plot for optimal model
par(cex = 1.5)
pamr.plotcen(m1, training_data, threshold = min_cv)

# Listing the surviving genes
message(pamr.listgenes(m1, training_data, threshold = min_cv, genenames = TRUE))
```

```r
# Assignment 3 - Part 2
# Predicting for test data using our trained model
m2_pred <- pamr.predict(m1, newx = x_test, threshold = min_cv)

# Creating confusion matrix for the true/predicted values
m2_err <- table("True" = y_test, "Predicted" = m2_pred)
message("NSC Confusion Matrix:")
m2_err

# Finding the error
NSC <- (sum(m2_err)-(sum(diag(m2_err))))/sum(m2_err)
message("NSC Error:")
message(round(NSC,3))

# Number of contributing features
nfeat_nsc <- m2$size[which.min(m2$error)]
message("Number of non-zero parameters:")
message(nfeat_nsc)

# Assignment 3 - Part 3
# Elastic net training (and timing the operation)
ptime <- proc.time()
m3 <- cv.glmnet(t(x_train), y_train, family = "multinomial", alpha = 0.5)
ptime_enet <- proc.time()-ptime

# Making predictions using test data
m3_pred <- predict(m3, newx = t(x_test), s = "lambda.min", type = "class")

# Creating confusion matrix for the true/predicted values
m3_err <- table("True" = y_test, "Predicted" = m3_pred)
message("Elastic Net Confusion Matrix:")
m3_err

# Finding the error
elastic_net <- (sum(m3_err)-(sum(diag(m3_err))))/sum(m3_err)
message("Elastic Net Error:")
message(round(elastic_net,3))

# Number of contributing features
nfeat_enet <- m3$nzero[match(m3$lambda.min, m3$lambda)][[1]]
message("Number of non-zero parameters:")
message(nfeat_enet)

# SVM training (and timing the operation)
ptime <- proc.time()
m4 <- ksvm(x = t(x_train), y = y_train, kernel = "vanilladot")
ptime_svm <- proc.time()-ptime

# Making predictions using test data
m4_pred <- predict(m4, newdata = t(x_test))

# Creating confusion matrix for the true/predicted values
m4_err <- table("True" = y_test, "Predicted" = m4_pred)
```

```r
message("SVM Confusion Matrix:")
m4_err

# Finding the error
SVM <- (sum(m4_err)-(sum(diag(m4_err))))/sum(m4_err)
message("SVM Error:")
message(round(SVM, 3))

# Number of support vectors
nfeat_svm <- m4@nSV
message("Number of support vectors:")
message(nfeat_svm)

# Creating table to compare the models on classification error
# number of features and time to run the training
data.frame(
  "Error" = round(rbind(NSC, "Elastic net" = elastic_net, SVM), 3),
  "Features" = rbind(nfeat_nsc, nfeat_enet, nfeat_svm),
  "Runtime" = rbind(ptime_nsc[[3]], ptime_enet[[3]], ptime_svm[[3]])
)

# Assignment 3 - Part 4

# Copy of original data
bh <- genes

# This part of the code creates a number of formula objects
# that are then input into the t.test function
# First we do one-hot encoding of the cell types
dummy <- model.matrix(~0+bh$CellType)

# Setting proper column names for the new columns...
colnames(dummy) <- c("CD19", "CD4", "CD8")

# ...and binding them to the gene data
bh <- cbind(dummy, bh)

# Making sure the dummy variables are treated as factors
bh[,1:3] <- lapply(bh[,1:3], factor)

# Extracting gene names to loop through
gene_names <- names(bh)[-(1:4)]

# An object with the three different cell types CD4, CD8 and CD19
cell_types <- levels(bh$CellType)

# Defining a function that takes a formula and a data object,
# runs a t-test and returns the p-value
# This function will be called from another function later
get_p <- function(form, data = bh) {
  t.test(formula = form, data = data)$p.value
}
```

```r
# Defining the second function that takes as input a vector of gene names,
# a single cell type and a data object
p_val <- function(g = gene_names, c = cell_types, data = bh) {

  # First constructing the formula objects by applying an anonymous function
  # over the gene name list and storing these
  formulas <- lapply(g, function(x) {formula(paste(x, "~", c))})

  # Then applying the get_p function to return a p-value for each formula
  # object created above
  # lapply returns a list, so we unlist and sort the result
  p_val <- sort(unlist(lapply(formulas, get_p)))

  # Finally returning that vector of p-values
  return(p_val)
}

# calling the function for the different cell types
p_val_CD4 <- p_val(c = "CD4")
p_val_CD8 <- p_val(c = "CD8")
p_val_CD19 <- p_val(c = "CD19")

# This is an overly complex looking code to produce a very simple
# looking plot, don't be scared!
# Inputs are simply the p-values for a cell type and an optional alpha parameter
plot_fun <- function(p_val, alpha = 0.05) {

  # Calculating the BH threshold
  adj_p <- alpha*(1:length(p_val)/length(p_val))

  # Indicator for if a hypothesis has been rejected or not
  sig <- vector(mode = "character", length = length(p_val))
  sig <- ifelse(p_val < adj_p, "rejected", "not rejected")

  # Making a data frame of the vectors of p-values and rejection indicator
  d <- as.data.frame(cbind("p_value" = p_val, "significant" = sig))

  # Finding the highest value under the threshold
  max <- which.min(d[,2] == "rejected")

  # Making a plot with p-values on y axis and index on the x axis
  # Indicating what cell type is shown by cutting the name of the input data set
  # -> follow the naming convention!
  plot(p_val, pch = 20, cex = 0.2, ylab = "p-value", main = substring(substitute(p_val), 7))

  # Vertical line at the position of the highest rejected hypothesis
  abline(v = max, lty = "dashed", col = "gray")

  # The BH threshold line
  lines(adj_p, lty = "dashed", col = "gray")

  # Overplotting the rejected values with red for better contrast
  points(
```

```
    x = 1:length(d[d[, 2] == "rejected", 2]),
    y = d[d[, 2] == "rejected", 1],
    col = "red",
    pch = 20,
    cex = 0.2
  )

  # Adding a single dot on the dashed vertical line
  points(x = max, y = .9, pch = 20, cex = 0.8)

  # Adding a text box indicating how many hypotheses are rejected
  # j is the highest numbered item that falls below the BH threshold
  text(
    paste("j =", max),
    x = max + 3,
    y = .9,
    pos = 4,
    cex = 0.7
  )

  # Text box indicating what alpha was used
  mtext(
    paste("Alpha =", alpha),
    side = 3,
    cex = 0.7,
    line = 0.1
  )
}

# Calling the plot function for different cell types
plot_fun(p_val_CD4)
plot_fun(p_val_CD8)
plot_fun(p_val_CD19)

# Feel free to try a different alpha! :)
# plot_fun(p_val_CD19, alpha = 0.10)


# Cleanup
rm(list=ls())
```