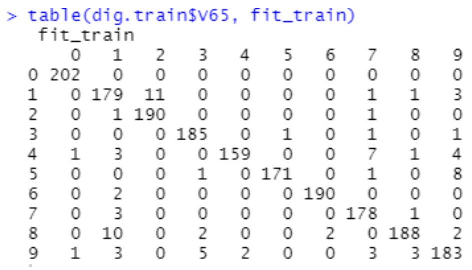
# Lab report – Lab 1 block 1

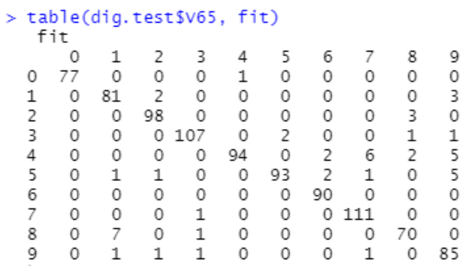
## Assignment 1

1. The data was loaded into the script with the read.csv command. Since the data in the csv file did not have named columns, the argument header=FALSE is used to communicate that the first row is to be treated as data, and not column names. The data was then partitioned into training, test, and validation, 50%, 25%, and 25%, as specified in the lab PM, and using the code provided in the lecture. The seed was verified to be set to 12345, as required in the course.

2. The kknn-function, with k=30, was used to create a model. As.Factor was used to make the kknn-function read the data as classes and thus provide a nominal, not continuous, return value. Two models were created: both had training data as training data, but one had training targets as its test data, and the other had test targets as its test data. This was so that we could see how far the kknn got from predicting both the real target values and the test targets. Comparison was made through confusion matrices (truth on y-axis, prediction on x-axis), and through calculating the misclassification error.



```
> table(dig.train$V65, fit_train)
   fit_train
      0   1   2   3   4   5   6   7   8   9
  0 202   0   0   0   0   0   0   0   0   0
  1   0 179  11   0   0   0   0   1   1   3
  2   0   1 190   0   0   0   0   1   0   0
  3   0   0   0 185   0   1   0   1   0   1
  4   1   3   0   0 159   0   0   7   1   4
  5   0   0   0   1   0 171   0   1   0   8
  6   0   2   0   0   0   0 190   0   0   0
  7   0   3   0   0   0   0   0 178   1   0
  8   0  10   0   2   0   0   2   0 188   2
  9   1   3   0   5   2   0   0   3   3 183
```



```
> table(dig.test$V65, fit)
   fit
      0   1   2   3   4   5   6   7   8   9
  0  77   0   0   0   1   0   0   0   0   0
  1   0  81   2   0   0   0   0   0   0   3
  2   0   0  98   0   0   0   0   0   3   0
  3   0   0   0 107   0   2   0   0   1   1
  4   0   0   0   0  94   0   2   6   2   5
  5   0   1   1   0   0  93   2   1   0   5
  6   0   0   0   0   0   0  90   0   0   0
  7   0   0   0   1   0   0   0 111   0   0
  8   0   7   0   1   0   0   0   0  70   0
  9   0   1   1   1   0   0   0   1   0  85
```

*Figure 1 Confusion matrix of training data (left), and confusion matrix of test data (right)*

In both cases we can see that they have strong diagonals, indicating overall good predictions, but some classifications are noticably wrong. In both cases, the kknn-function predicted 8 as being 1, and the digit 9 was notoriously classified as other digits, but without a preference for a single digit. The observation was confirmed with low misclassification rates on ~4,5% for training model and ~5,3% for test model, indicating errors in the models but else strong predictions.

3. By looking at the 8th column in the prob variable in the training model, the probability for assigning the digit the class "8" could be seen. By adding the column to the training data and then ordering the rows by the probability column, from lowest to highest and then highest to lowest, the hardest digits to categorise as 8 where 1793 (10% probability), 869 (13,3% probability), and 2068 (16,7% probability); and the easiest, among many, were 1890 and 1982, both had 100% probability. Below the three hardest cases to classify can be seen, followed by the two easiest cases. The pictures are flipped both horizontally and vertically.

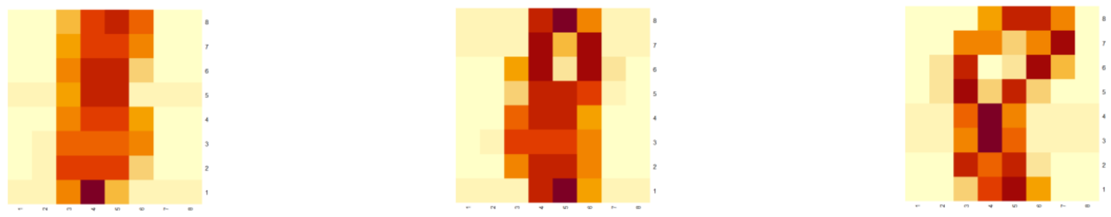


*Figure 2 The three hardest cases to classify as 8*

The three digits are not resembling an 8, and the hardest one (furthest to the left), is similar to an 1, which can explain the many misclassifications of 8 as 1 previously seen in the confusion matrix. The 8 furthest to the right is also resembling a 6, note that they are upside down and flipped horizontally.

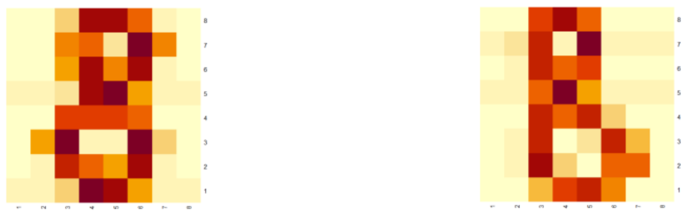


*Figure 3 The two among the easiest to classify as 8*

These have distinct loops, compared to the ones above, and less of a straight 1-like shape.

4. The kknn algorithm uses the majority vote of the neighbours to classify data. This means that for small number of neighbours (small k), the model will be volatile, and the predictions can vary with only small changes of the neighbour's classes. This is not the case when more neighbours are used (large k), since then the change of one neighbour will have a small effect on the total vote. For example, when k=30 the change of one neighbour will only have a 1/30 impact on the classification. Compare this to when k=3 -> 1/3 impact on the decision. Thus, the variance decreases with increasing k, but the bias increases.

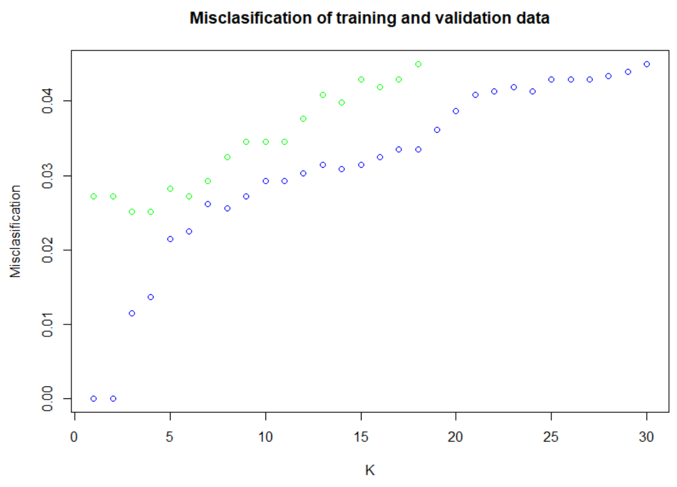


*Figure 4 Misclassification rate of training (blue) and validation (green) depending on the value k*

The misclassification is zero when using the training data with small k:s and grows steadily with increasing k. A similar pattern is seen for the validation, but here the misclassification rate is never zero and there is initially a small dip in misclassification rate before it rises. Optimal k is therefore in the dip and is both 3 and 4. K=4 is chosen for further study, and we can calculate the classification error as 2,5%, about 1,5 pp lower than the value of k=30. 2,5% is larger than k=1 for the training data, but the usage of k=1, provides no help in classification since it is too volatile. Furthermore, when choosing optimal k, we should study the performance on the validation data, since it indicates how the model performs on new data, which the training data does not.

5. Empirical risk (cross-entropy loss) is calculated through - log likelihood on the validation set, and the smallest loss is selected. This is the same as selecting the maximum log likelihood. Log likelihood has the following formula

$$L(\theta) = \sum_{i=1}^{N} \log\big(P_\theta(y_i)\big) = \log\big(P_\theta(y_1)\big) + \log\big(P_\theta(y_2)\big) + \cdots + \log\big(P_\theta(y_N)\big)$$

$$where\ in\ this\ case\ P_\theta(y_i) = P_\theta(C = y_i|y_i)$$

For example, the probability to classify y as 8 given that y = 8. Each probability is logged (1e-15 was added before logged to avoid numerical problems) and summed up, for each value of k. The optimal k was found to be 6, by plotting the log likelihoods against their value of k and selecting the largest value. K=6 had the largest log likelihood, which was -113.768. With k=6, we thus get a slightly higher value than the optimal k=4 calculated by using the misclassification error. Using cross-entropy provides us with the k which will take us closest to the correct values, by using likelihood to calculate the highest sum of correct classifications. Minimizing misclassifications only give true-false indications, and rate all false as totally false, even though they were nearly classified correctly. This black and white validation system makes it impossible to differentiate between two k:s which had the same number of false, but where one were always closer to be right than the other. Log likelihood provides us with the possibility to differentiate between the tow models thanks to the continuous quality to the function.

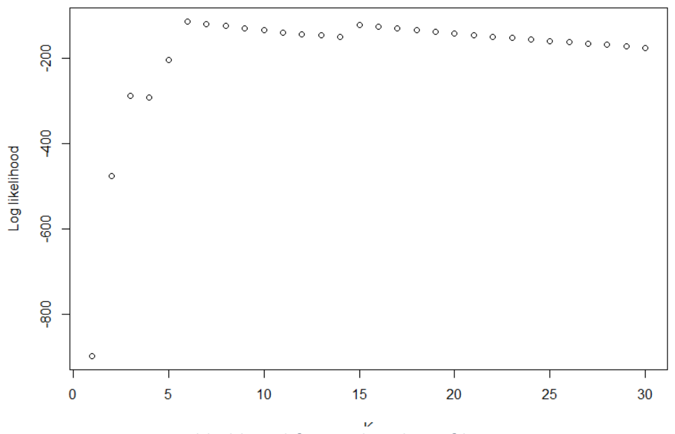


*Figure 5 Log likelihood for each value of k*

## Assignment 2

1. As motor_UDPRS is normally distributed and can be modelled using Ridge regression, we can model the problem as follows, based on the lecture slides:

$$\boldsymbol{y} \sim N(\boldsymbol{y} | \boldsymbol{w_o} + \boldsymbol{Xw}, \boldsymbol{\sigma^2 I})$$

$$\boldsymbol{w} \sim \boldsymbol{N}\left(\boldsymbol{0}, \frac{\sigma^2}{\lambda}\boldsymbol{I}\right)$$

2. The dataset was scaled using the scale-function in r, and then split into 60/40 training/test-set.
3. Some background on the functions implemented:
   a. The Loglikelihood-function was implemented according to the log-likelihood of the normal distribution; the following function, n being the number of training cases:

   $$\mathcal{L}(Y, X; w, \sigma) = -\frac{n}{2} ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}||Y - Xw||_2^2$$

   b. The Ridge-function utilizes the loglikelihood function to calculate the following:

   $$Ridge = -loglikelihood + \lambda \sum w^2$$

   c. The RidgeOpt-function uses the optim-function to minimize Ridge, thereby calculating the optimal sigma and w. A help function is utilized to convert between the function inputs and the parameter vector required for optim.
   d. DF calculates the degrees of freedom using the following function:

   $$trace\left(X(X^T X - \lambda I)^{-1} X^T\right)$$

4. Using the RidgeOpt function to calculate the parameters using the training data, we get the following arrays of values for w for each lambda:

   **1**: 0.172996666 -0.168829861 -0.013092660 -0.067795432 -0.003197100 0.533470862 -0.144982157 -0.148552102 -0.373728940 0.311096373 -0.152180785 -0.184526376 -0.238338787 0.004985117 -0.276027705 0.228518289

   **100**: 0.04705232 -0.12989117 0.01581422 -0.02608552 0.01590113 0.03746697 0.02145255 -0.07630543 -0.10038417 0.22443941 -0.07633449 -0.14197326 -0.18287605 0.02776323 -0.24230036 0.21604228

   **1000**: 0.005719971 -0.040722825 -0.003431963 -0.006668571 -0.003460276 0.005683451 0.012870550 -0.017065189 -0.009538218 0.071697409 -0.017166487 -0.036796192 -0.078572822 0.045315073 -0.126031291 0.104457311

   We use these parameters to predict the responses for each set of feature values. These can be seen in the code. The MSE score lies around 0,9. Looking especially at the MSE results of the test data, lambda=100 results in the lowest MSE score of 0.92632, indicating that these parameters result in the best model according to the MSE method.
   In this case we are doing regression, I.e., working with continuous data that need to predict a continuous value. Therefor MSE is working well rather than for example misclassification which is used when classifying data into discrete classifications.

Comparing to the Cross-validation method, in this case we have a quite large dataset which is better suited for Holdout, as cross-validation would be very computationally demanding.

5. Using AIC results in lower lambda leading to a lower score, meaning that lambda= 1 results in the lowest risk of 9553,60.
   AIC is often better than Hold-out, as it only uses training data to determine the risk, and effectivity of the predicted model. Therefor you can use the entire dataset as a training set and use AIC instead of checking against a validation set, thereby theoretically receiving a better model.

## Assignment 3

1. The given data from the csv-file was read into the script using the read.csv-command with the header set to TRUE since the first row of the file is column names. Afterwards, the data is prepared for training and testing by using the set.seed function and randomly splitting the data in two equally large sets. These two sets are afterwards prepared for the model by splitting the fat (response variable) from the rest of the data to create the coefficient. This creates two sets of matrices that can be used in the regression. The underlying probabilistic model is thus:

$$p(y|x,w) = N(w_0 + w_1 x, \sigma^2)$$

$$y \sim N\left(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x},\ \sigma^2\right), \qquad where$$

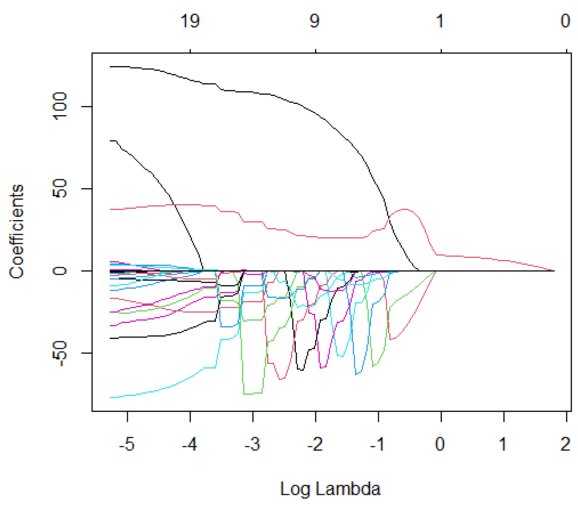
$$\boldsymbol{w} = \{w_0, \dots w_d\}$$
$$\boldsymbol{x} = \{1, x_1, \dots x_d\}$$

   After training the model on the training set using the lm-command the model is used to predict the test data using the training data using the predict function. Both the training and test error is calculated using the squared mean error between the predicted model sets and target values in the sets. The mean error for the training set was 0.00571 and for the test set it was 722.4294. Given these error values the training set is quite good, but the error is too large for the test set. Hence the model quality is not that good.
2. In the LASSO regression the following objective function is being minimized:

$$\hat{w}^{lasso} = argmin \sum_{i=1}^{N} \left(y_i - w_0 - w_1 x_{1j} - \cdots w_p x_{pj}\right)^2$$

$$\boldsymbol{subject\ to} \sum\nolimits_{\boldsymbol{j=1}}^{\boldsymbol{p}} |w_i| \le s$$

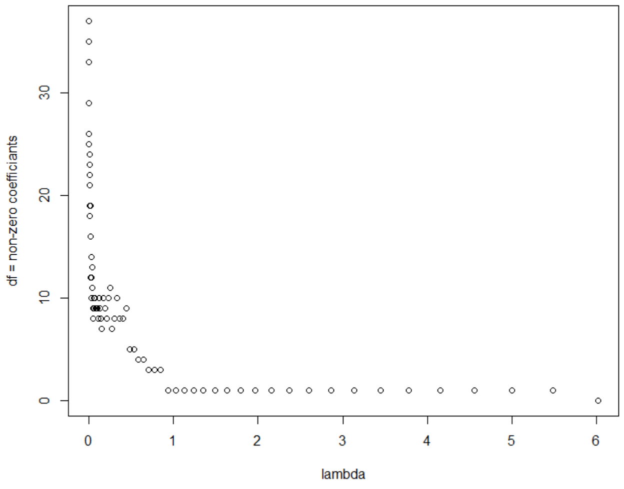
3. To use the LASSO regression the library “glmnet” with alpha=1 is used. To illustrate how the regression coefficients depend on the penalty factor the degrees of freedoms in the lasso model is used. By using the degrees if freedom in the plot we can see for which values for the penalty how many features are being included in the model:



From the figures above it is possible to read that just below log(lambda)=0 for the penalty factor there are three non-zero coefficients, that is three features. This equates to a value for the penalty factor od around [0.8;0.9]. To exactly find the value, the command “which(lasso.mod$lambda <= 1)” is used to find which features are active in the model. The earliest datapoint that fills these criteria is the 22nd.

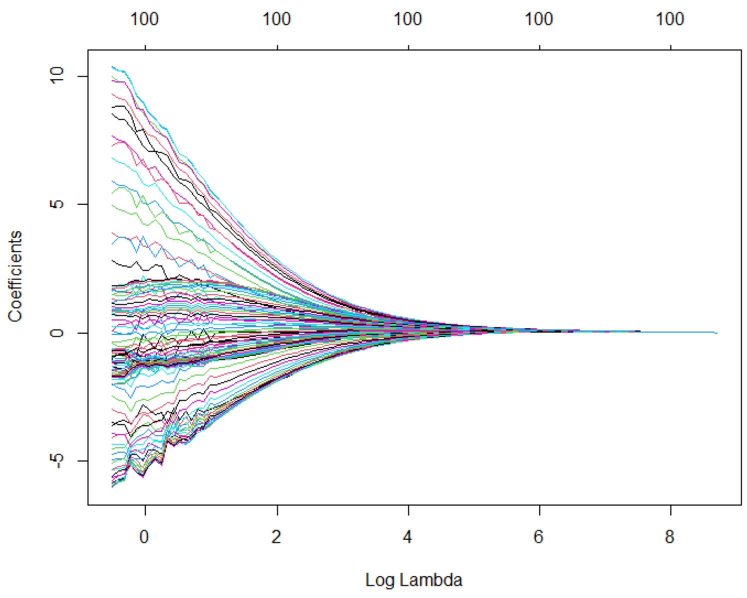
To find what lambda value corresponds with this one “lasso.mod$lambda[22]” is used, which returns that lambda is 0.8530452. Therefore, at the penalty factor= 0.8530452 we select three features.

4. Plotting the lambda values from the LASSO and the degrees of freedom gives the figure below



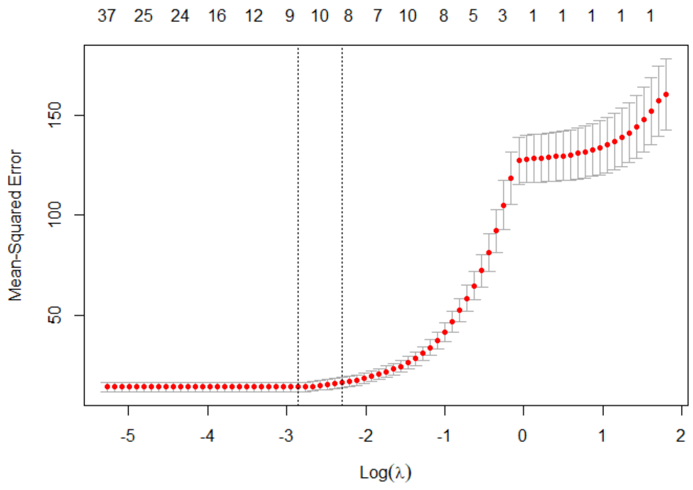
The trend is expected, since the closer to 0 (the more negative log lambda), the more coefficients are not 0 (same result as the log plot above). Which makes sence given the mathematical model leaves less room for features to be included as lambda’s value is increased in the minimization optimization.

5. To use the LASSO regression the library “glmnet” with alpha=0 is used.

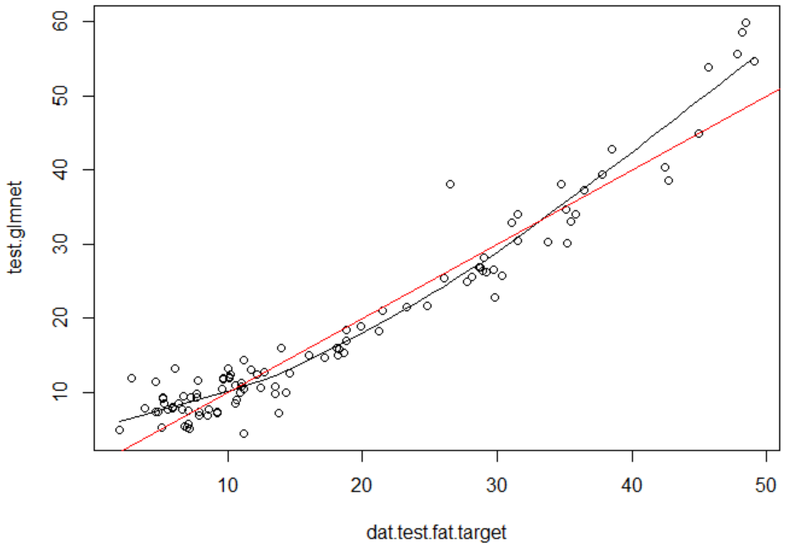


The number of coefficients reduce as lambda increases, which is expected. Also, the number of lambdas seem to be fewer than in the LASSO. Lastly, all features converge towards zero smoothly in contrast to the LASSO. However, there are no non-zero coefficients for any of the lambda values.

6. To find the optimal value for the model the command cv.glmnet() is used. As the plot below shows, the MSE is lower for lower values for the penalty factor. The dotted lines show the lambda.min, which gives the lowest minimizes out-of-sample loss in CV, and the lambda1.se is the largest lambda within one standard deviation from lambda.min. The lambda.min value is 0.05744535, which gives the log value -2.856921, as seen in the plot. The number of coefficients is 9.



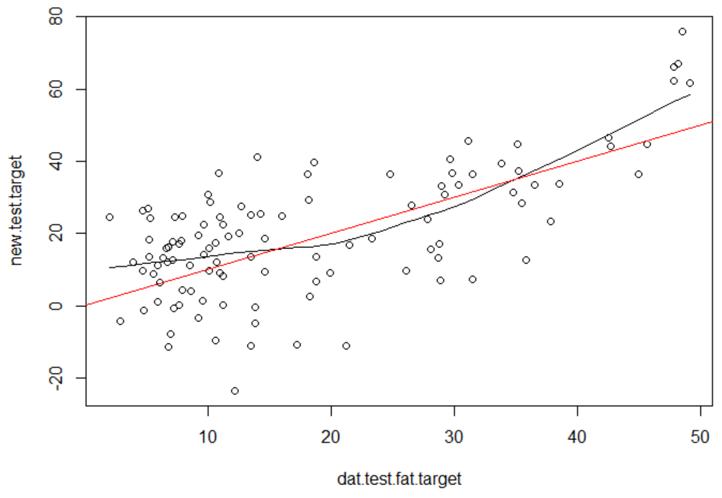
The plot shows overlapping confidence intervals for the MSE for lambda=-2 and the lambda.min. This means it is impossible to say with certainty which one of them is statistically better. In the figure below a scatter plot is shown with a line which corresponds to a perfect fit between model and target data. By comparing the lines, the model is neither good or horrible, and it has some overfitting towards the endpoints. By looking at the residuals, the mean squared residuals is 13.67339, which is significantly better than the regression model from 1).



7. To generate new target values the model from 1) is used:

$$y = w_0 + w_1 x + \epsilon$$
$$y \sim N\left(\boldsymbol{w}^T \boldsymbol{x},\, \sigma^2\right)$$

To perform this it is needed to create a intercept for the data set by creating a matrix with only one as values. After this, residuals are calculated from the test set's target and the LASSO model. These are then randomly added to the matrix using the rnorm() function. The figure below shows the scatter plot of the new target data and test target data. The squared residuals is 181.5103, which together with the plot shows that the data generation is in the right direction, but not very good as there is quite a large displacement from the test data at times and worse compared to the previous model from task 6.



## Statement of contribution

The assignments were divided between each member of the team, with each member being responsible for one assignment each. This included completing the assignment and writing the report for the corresponding assignment. If any issues or questions arose, the questions where discussed together in the group in order to help further progress. For this lab, the assignments were divided as follows:

- Assignment 1: Ingrid Wendin, ingwe018
- Assignment 2: David Åström, davas593
- Assignment 3: Per Bark, perba583

## Appendix

### Code – Assignment 1

```
# Create project and store the .csv file in it before working
mydata = read.csv("optdigits.csv", header= FALSE)

#1. Partition data
n=dim(mydata)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
dig.train=mydata[id,]

id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.25))
dig.valid=mydata[id2,]

id3=setdiff(id1,id2)
dig.test=mydata[id3,]

#2. Fit 30-nearest neighbor - why continuous data?
library(kknn)

# Testing with training data
dig.model_train = kknn(as.factor(V65) ~ ., train=dig.train, test=dig.train, k=30,
kernel="rectangular")
fit_train = fitted(dig.model_train)

# Testing with test data
dig.model = kknn(as.factor(V65) ~ ., train=dig.train, test=dig.test, k=30,
kernel="rectangular")
fit = fitted(dig.model)

# Confusion matrix train data
table(dig.train$V65, fit_train)

# Confusion matrix test data - several 8 was thought to be 1, 9 had several
misclass, it predicted many to be 9
table(dig.test$V65, fit)

misclass = function(X,X1) {
  n = length(X)
  return (1-sum(diag(table(X,X1)))/n)
}

# Misclassification train - ~4% misclass, slightly better than test but no
overfitting
misclass(dig.train$V65, fit_train)

# Misclassification test - ~5% misclass
misclass(dig.test$V65, fit)

#3. Find 8:s in training data

  # add column with probability to classify the digit as an 8
dig.train_prob8 = cbind(dig.train,prob8=dig.model_train$prob[,9])

  # subset with only 8 and order ascending
dig.train_prob8 = subset(dig.train_prob8,V65==8)
dig.train_prob8 = dig.train_prob8[order(dig.train_prob8$prob8),]
dig.train_prob8

  # Lowest: We can see that rowname=1793 had 10% prob to be classified as an 8,
rowname=869 had 13,3%, rowname=2068 had 16,7% chance

dig.train_prob8 = dig.train_prob8[order(dig.train_prob8$prob8, decreasing=TRUE),]
dig.train_prob8

  # Highest: We can see that rowname=1890 had 100% prob to be classified as an 8,
rowname=1982 had 100% chance

  # create heatmap - good values - Note that the output in the heatmap has been
flipped two times compared to the matrix
  # Can see the loops. It is easy to see that it is an 8 when I know that it is an
8
dig.test_num = as.matrix(mydata[1890,1:64])
dig.test_num = matrix(dig.test_num, nrow=8, byrow=TRUE)
heatmap(dig.test_num, Colv=NA, Rowv=NA)

dig.test_num = as.matrix(mydata[1982,1:64])
dig.test_num = matrix(dig.test_num, nrow=8, byrow=TRUE)
heatmap(dig.test_num, Colv=NA, Rowv=NA)

  # create heatmap - bad values
  # Cannot see the loops, only a blob/vertical line -> cannot see that it is an 8
dig.test_num = as.matrix(mydata[1793,1:64])
dig.test_num = matrix(dig.test_num, nrow=8, byrow=TRUE)
heatmap(dig.test_num, Colv=NA, Rowv=NA)

dig.test_num = as.matrix(mydata[869,1:64])
dig.test_num = matrix(dig.test_num, nrow=8, byrow=TRUE)
heatmap(dig.test_num, Colv=NA, Rowv=NA)

dig.test_num = as.matrix(mydata[2068,1:64])
dig.test_num = matrix(dig.test_num, nrow=8, byrow=TRUE)
heatmap(dig.test_num, Colv=NA, Rowv=NA)

#4. Kknn - plot - Seems like k=4 is minimal - More neighbors with larger k makes
the model more complex, but more stable with smaller variance

k.set = data.frame(K=integer(), Test=integer(), Valid=integer() )

for (k in 1:30) {
    #training
  dig.kknn = kknn(as.factor(V65)~., dig.train, dig.train, k=k,
kernel="rectangular")
  m1 = misclass(dig.train$V65, dig.kknn$fitted.values)
    #validation
  dig.kknn.valid = kknn(as.factor(V65)~., dig.train, dig.valid, k=k,
kernel="rectangular")
  m2 = misclass(dig.valid$V65, dig.kknn.valid$fitted.values)

  k.set = rbind(k.set, c(k,m1,m2))
}
colnames(k.set) = c("K", "Test", "Valid")

plot(k.set$K, k.set$Test, col="blue", main="Misclasification of training and
validation data", ylab="Misclasification", xlab="K")
```

```
points(k.set$K, k.set$Valid, col="green")


  #Test error for K = 4 - test error is 2,5% which is about the same as validation
error and worse than training error
dig.kknn.test = kknn(as.factor(V65)~., dig.train, dig.test, k=4,
kernel="rectangular")
misclass(dig.test$V65, dig.kknn.test$fitted.values)

#5. KKnn fit to training data - K = 6 is the k for maximized likelihood


k.set = data.frame(K=integer(), Log_likelihood=integer())

for (k in 1:30) {

  dig.kknn.valid = kknn(as.factor(V65)~., dig.train, dig.valid, k=k,
kernel="rectangular")

  n = dim(dig.valid)[1]
  # column indexes = target values + 1
  index = matrix(c(1:n,(dig.valid$V65+1)), ncol = 2)
  prob = dig.kknn.valid$prob[index]
  prob = matrix(prob, ncol=1)
  prob = prob + 1e-15
  m1 = apply(prob, 2, log)
  m2 = apply(m1, 2, sum)

  k.set = rbind(k.set, c(k, m2))

}

colnames(k.set) = c("K", "Log_likelihood")
plot(k.set$K, k.set$Log_likelihood, xlab="K", ylab="Log likelihood")

# Find optimal
index = which(k.set$Log_likelihood== max(k.set$Log_likelihood))
K.opt = k.set$K[index]
K.opt
```

## Code – Assignment 2

```
# import data
data = read.csv("parkinsons.csv")

# set seed, scale and divide data
set.seed(12345)
data.scaled = scale(data[,5:ncol(data)])

data.len = nrow(data)
trainIndexes = sample(1:data.len, floor(data.len*0.6))

data.train = data.scaled[trainIndexes,]
data.test = data.scaled[-trainIndexes,]

y.train = data.train[, 1]
x.train = data.train[, 3:18]

Loglikelihood = function(w, sigma) {
  n = dim(data.train)[1]

  return(-n*log(2*pi*sigma^2)/2 - sum((y.train-(x.train%*%w))^2)/(2*sigma^2))
}

Ridge = function(w, sigma, lambda) {
  return(-Loglikelihood(w, sigma) + lambda*norm(w, "2")^2)
}

Ridgeopt = function(lambda) {

  RidgeHelp = function(par, lambda) {
    sigma = par[1]
    w = par[-1]
    return(Ridge(w, sigma, lambda))
  }

  par.init = c(1, numeric(16))

  return(optim(par = par.init, fn = RidgeHelp, method = "BFGS", gr = NULL,  lambda
= lambda))
}

DF = function(lambda) {
  x = x.train
  H = x%*%solve(t(x)%*%x + lambda*diag(16))%*%t(x)
  return(sum(diag(H)))
}

# train model, get params
par.1 = Ridgeopt(1)
par.100 = Ridgeopt(100)
par.1000 = Ridgeopt(1000)

w.1 = par.1$par[-1]
# 0.172996666 -0.168829861 -0.013092660 -0.067795432 -0.003197100  0.533470862 -
0.144982157 -0.148552102 -0.373728940
# 0.311096373 -0.152180785 -0.184526376 -0.238338787  0.004985117 -0.276027705
0.228518289
sigma.1 = par.1$par[1] # 0.93450
w.100 = par.100$par[-1]
```

```
# 0.04705232 -0.12989117  0.01581422 -0.02608552  0.01590113  0.03746697
0.02145255 -0.07630543 -0.10038417  0.22443941
# -0.07633449 -0.14197326 -0.18287605  0.02776323 -0.24230036  0.21604228
sigma.100 = par.100$par[1] # 0.93758
w.1000 = par.1000$par[-1]
# 0.005719971 -0.040722825 -0.003431963 -0.006668571 -0.003460276  0.005683451
0.012870550 -0.017065189 -0.009538218
# 0.071697409 -0.017166487 -0.036796192 -0.078572822  0.045315073 -0.126031291
0.104457311
sigma.1000 = par.1000$par[1] # 0.95689

# predict y training
y.new.train.1 = x.train%*%w.1
y.new.train.100 = x.train%*%w.100
y.new.train.1000 = x.train%*%w.1000

# MSE training
MSE.train.1 = mean((y.train - y.new.train.1)^2) # 0.87328
MSE.train.100 = mean((y.train - y.new.train.100)^2) # 0.87906
MSE.train.1000 = mean((y.train - y.new.train.1000)^2) # 0.91563

x.test = data.test[,3:18]
y.test = data.test[,1]

# predict y test
y.new.test.1 = x.test%*%w.1
y.new.test.100 = x.test%*%w.100
y.new.test.1000 = x.test%*%w.1000

# MSE testing
MSE.test.1 = mean((y.test - y.new.test.1)^2) # 0.92903
MSE.test.100 = mean((y.test - y.new.test.100)^2) # 0.92632
MSE.test.1000 = mean((y.test - y.new.test.1000)^2) # 0.94792

# AIC
AIC = function(w, sigma, lambda) {
  return(2*DF(lambda)-2*max(Loglikelihood(w, sigma)))
}

AIC.1 = AIC(w.1, sigma.1, 1) # 9553.6
AIC.100 = AIC(w.100, sigma.100, 1) # 9576.9
AIC.1000 = AIC(w.1000, sigma.1000, 1) # 9720.5
```

## Code – Assignment 3

```
mydata = read.csv("tecator.csv")

# prepare data
set.seed(12345)
n = dim(mydata)[1]
id = sample(1:n, floor(n*0.5))
dat.train = mydata[id,]
dat.test = mydata[-id,]

# No pre-processing
#-----------------------1------------------------
  # remove all except channels 1-100, collect fat
dat.train.fat = dat.train[,2:102]
dat.train.fat.channels = dat.train[,2:101]
dat.train.fat.target = dat.train[,102]
dat.test.fat = dat.test[,2:102]
dat.test.fat.target = dat.test[,102]
dat.test.fat.channels = dat.test[,2:101]

  # Fitting to training data
reg.mod <- lm(Fat ~., data = dat.train.fat)
  # Probabilistic model
reg.mod$coefficients
  # Note that many variables have unsatisfactory significance levels
summary(reg.mod)

  # Predict training data and test data
reg.fit.train <- predict(reg.mod, dat.train.fat)
reg.fit.test <- predict(reg.mod, dat.test.fat)

  # Error: E((train- predict of train)^2) = 0.005709117 good
err.train.fat = mean((dat.train.fat.target-reg.fit.train)^2)
err.train.fat

  # Error: E((test - predict of test)^2) = 722.4294  #Really bad(!)
err.test.fat = mean((dat.test.fat.target-reg.fit.test)^2)
err.test.fat

  # Checking residual - control
#plot(x=dat.test$ï..Sample, y=((dat.test.fat.target-reg.fit.test)^2))

#-----------------------2------------------------
# LASSO on fat - objective function
# Objective: minimize the cost function = minimize -loglikelihood = min sum((y-
y_hat)^2) = min sum((y - predicted y)^2)
# Subject to lambda*sum(abs(w))<= s, where lambda = 1
# So objective function is minimize residuals, just like the regression, but now
with an additional boundary -> glmnet can solve this problem


#-----------------------3------------------------
# glmnet() with alpha = 1
library(glmnet)

 lasso.mod <- glmnet(as.matrix(dat.train.fat.channels),dat.train.fat.target,
alpha=1, family="gaussian" )
 # Plot function displays relation between coefficients and log lambda
 plot(lasso.mod, xvar="lambda")
 # When penalty is greater (more to the right(e0)) then few coefficients are != 0,
and when the penalty is smaller, then we can have more coefficients (lambda = e-4)
 # Most coefficients are variating between 0 and -50, but three are positive

 #df =  The number of nonzero coefficients for each value of lambda. Visually
looks like around lambda = 0 has 3

plot(x=lasso.mod$lambda, y=lasso.mod$df, xlab="lambda", ylab="df = non-zero
coefficiants") # 3 coeff around around lambda = 0,9 is confirmed

lambda = which(lasso.mod$lambda <= 1)
which(lasso.mod$df >= 3)
lasso.mod$df[22] # has 3 coefficients
lasso.mod$lambda[22] # 3 coefficients when lambda = 0.8530452

#----------------------4------------------------
# Df - penalty parameter plot -> Df vs lambda

plot(x=lasso.mod$lambda, y=lasso.mod$df, xlab="Lambda", ylab="df")
 # The trend is expected, since the closer to 0 (the greater more negative log
lambda), the more coefficients are not 0 (same result as the log plot above)

#----------------------5------------------------
# Ridge regression - alpha = 0
ridge.mod <- glmnet(as.matrix(dat.train.fat.channels), dat.train.fat.target,
alpha=0, family = "gaussian")

plot(ridge.mod, xvar="lambda")
# The ridge plot for lambda also has fewer non-zero coefficients with greater
lambda, but only a few values of lambda in this graph compared to the lasso
plot(x=ridge.mod$lambda, y=ridge.mod$df)
# no non-zero coefficients for any of the lambda values -> No value of lambda will
give 3 coefficients

#----------------------6------------------------
# Cross-validation lasso
lasso.cv.mod = cv.glmnet(as.matrix(dat.train.fat.channels),dat.train.fat.target,
alpha=1, family="gaussian")

plot(lasso.cv.mod)
 # The CV score = MSE, is low for small lambdas (small penalties), MSE takes off
around log lambda = -2,5, increases to 0 and then rises more slow, lambda.min is
the dotted line to the left
# The plot shows that the log lambda_min has an overlapping MSE interval with log
lambda = -2, and it is thus impossible to say that log lambda min is significantly
better than -2
# It looks like 9 coefficients are in the model for lambda min, but check coef()
to verify
lasso.cv.mod$lambda.min # -2.856921

coef.opt = matrix(coef(lasso.cv.mod, s='lambda.min'))
table(coef.opt[,1]!=0)# there are 9 non-zero coefficients when optimal lambda
(including intersection), as thought!

test.glmnet = predict(lasso.cv.mod, newx=as.matrix(dat.test.fat.channels),
s="lambda.min") # specifying predict for lambda min

scatter.smooth(dat.test.fat.target, test.glmnet)
```

```
abline(0,1, col="red") # line which the data should follow

# The model is not very good, but not horrible.
# Notice the that values are off at the outer points

# Checking residuals
err.test.fat = mean((dat.test.fat.target - test.glmnet)^2)
err.test.fat # squared residuals = 13.67339 much lower that ~700 before

# Plotting residuals show that some predictions are quite off - often predicted
bigger value than test
residual = dat.test.fat.target - test.glmnet
plot(x=dat.test.fat[,1], y=residual, col="purple")
abline(0,0, col="red")

#-----------------------7------------------------
# Generate new target values - y ~ N(wTx, sigma^2)
  # w_opt = coef.opt from Lasso result (+ intercept), dat.test.fat.channels = x (-
intercepts, must add x0)

  # Add w0 to first position
x0 = matrix(rep(c(1), 108), ncol=1)
x0
test.channels = dat.test.fat.channels
test.channels = cbind(test.channels, x0)
test.channels = subset(test.channels, select=c(101,1:100))

  # change to matrix and transpose
test.channels = as.matrix(test.channels)
test.channels = t(test.channels)

  # Create the y by multiplying w with x and add error from N()
s2.residual = mean(residual^2)
e = matrix(rnorm(108, 0, s2.residual), nrow = 1)
w = t(coef.opt)
new.test.target = w%*%test.channels
new.test.target = new.test.target + e
new.test.target

scatter.smooth(dat.test.fat.target, new.test.target)
abline(0,1, col="red")

# The predictions are still near the target but sometimes a bit off. One can
clearly see the displayed pattern
err.test.fat = mean((dat.test.fat.target - new.test.target)^2)
err.test.fat # squared residuals =  181.5103 worse than 13.67339 much lower that
~700 before
```