Cheat sheet TDDE01

Filip Cornell

2019-01-06

Contents

1 Uncertainty estimation

Given an estimator $\hat{f} = \hat{f}(x, D)$ (or $\hat{\alpha} = \delta(D)$), how do we estimate the uncertainty? We can do this in two ways.

1. If the distribution for data D is given, compute analytically the distribution for the estimator and derive confidence limit. This is however often difficult.
2. Use Bootstrap.

1.1 Bootstrap

Bootstrap is a method to measure the uncertainty of ones estimator w, and it works for all distribution types. There are two types of bootstrap - non-parametric bootstrap and parametric bootstrap. It can have a bad accuracy for small data sets where $n < 40$, but parametric bootstrap works even for small samples.

1.1.1 Non-parametric bootstrap

In non-parametric bootstrap, we sample data *with replacement*. Given the estimator $\hat{w} = \hat{f}(D)$ and we assume that $X \sim F(X, w)$ where F and w are unknown, do

1. Estimate $\hat{w}$ from data $D = (X_1, .., X_n)$

2. Generate $D_1 = (X_1^*, .., X_n^*)$ by sampling with replacement.
3. Repeat step 2 B times.
4. The distribution of w is given by $\hat{f}(D_1), .., \hat{f}(D_B)$

Non-parametric bootstrap can be applied to any deterministic estimator.

1.1.2 Parametric bootstrap

If the distribution is correct, parametric bootstrap is more precise. Given the estimator $\hat{w} = \hat{f}(D)$ and we assume that $X \sim F(X, w)$ where F is known and w is unknown, do

1. Estimate $\hat{w}$ from data $D = (X_1, .., X_n)$
2. Generate $D_1 = (X_1^*, .., X_n^*)$ by generating from $F(X, \hat{w})$
3. Repeat step 2 B times.
4. The distribution of w is given by $\hat{f}(D_1), .., \hat{f}(D_B)$

1.1.3 Bootstrap confidence intervals

To do a bootstrap confidence interval, do

1. Compute $\hat{f}(D_1), .., \hat{f}(D_B)$ using bootstrap and sort them in ascending order and get $w_1, .., w_B$.
2. Define $a_1 = \lceil B\frac{\alpha}{2} \rceil$ and $a_2 = \lfloor (B - B \cdot \frac{\alpha}{2}) \rfloor$
3. The confidence intervals will thus be given by (w_{a_1}, w_{a_2})

In regression both parametric and non-parametric bootstrap can be used.

1.1.4 Bootstrap example

```
data = read.csv("bank.csv", sep = ";", dec = ",")
plot(data$Time,data$Visitors, xlim = c(9,13), ylim = c(min(data$Visitors), 250))
model = glm(Visitors ~ Time, data = data, family=poisson(link = "log"))
model$coefficients
lines(x=seq(0,13,by=0.5), y =
        exp((model$coefficients[1] + model$coefficients[2]*seq(0,13, by = 0.5))))
lines(x=data$Time, y = exp(predict(model, data)), col = "blue", lwd = 3)
# The probabilistic expression for the fitted model
# is (paste into latex) $$y_i = t\cdot exp(x_i^T \beta)$$

library(boot)

# Task 2: Now, compute a prediction band.

# Statistical function to return predictions.
statistic_func = function(..data) {
  model = glm(Visitors ~ Time,
              data = ..data,
              family=poisson(link = "log"))
  preds = exp(predict(model, data1213)) # predict on the globally first sampled data.
  preds = rpois(nrow(..data), preds) # Sample out of a poisson distribution out of this.
  return(preds)
```

```
}

# A random generator
rand_gen = function(.data, model) {

  preds = exp(predict(model,.data))
  # Return the new sampled ones out of the new model.
  .data$Visitors = rpois(nrow(.data), preds)
  return(.data)
}

B = 1000
set.seed(12345)


data1213 = data.frame(Time = seq(9,13,by=0.05),
                      Visitors = as.numeric(21))
set.seed(12345)
# Sample the data first once through random gen once to start with that data.
data1213 = rand_gen(data1213, model)
set.seed(12345)
# Send that sampled data in!
pred_band = boot(data1213,
                 mle = model,
                 statistic = statistic_func,
                 R=1000,
                 sim = "parametric",
                 ran.gen = rand_gen)

e1 = envelope(pred_band)

lines(x=seq(9,13,by=0.05),y= e1$point[1,], col = "green", lwd = 3)
lines(x=seq(9,13,by=0.05),y= e1$point[2,], col = "green", lwd = 3)
```

2 Supervised learning

There are two types of models in supervised learning; generative and discriminative models.

Generative models will model the actual distribution of each class, and learns the joint probability distribution $p(x, y)$, and predictsusing bayes theorem. Generative can be used to generate new data, and are normally easier to fit. Each class is estimated separately, and we do not need to retrain when a new class is added. A disadvantage is that generative models often makes too strong assumptions about p(X|Y,w) which induces bad performance.

Discriminative models will model the decision boundary between the classes. It will the learn the conditional probability distribution $p(y|x)$. An example is logistic regression. In discriminative models, we can replace X with $\phi(X)$ and the model will still work.

3 Parametric versus non-parametric models

There are two types of models: Parametric models and non-parametric models.

3.0.0.1 Parametric models

have a certain number of parameters independently of the size of the training data, and one makes assumptions about the distribution of the data. Typical examples are logistic and polynomial regression. These models will have a number of parameters, which is not dependent on the size of the data, but rather how many the model itself requires.

3.0.0.2 Non-parametric models

has a number of parameters that grows with the size of the training data. An example is a K-NN classifier; the parameters here are the distances between the points. Gaussian processes are also non-parametric.

4 Regression

Regression is basically trying to fit, with a certain number of parameters for the data points, a model to a continuous response variable.

4.1 Linear regression - OLS (Ordinary Least Squares Regression)

A linear regression model is the most simple regression model. It is given by

$$\hat{Y_j} = \beta_0 + \sum_{i=1}^{m} \beta_i \cdot x_{i,j} + \epsilon \quad \forall j \in \{1, .., n\}, \epsilon \sim N(0, \sigma^2)$$

where β_0 is the intercept. It is called OLS because one aims to reduce the squared error. Maximizing $\hat{w} = max_w p(D|w)$ is equivalent to minimizing the error, i.e.,

$$\hat{w} = max_w p(D|w) \Leftrightarrow RSS(w) = \sum_{i=1}^{n} (Y_i - w^T X_i)^2$$

This error can usually be moved up to log for easier calculations.

For optimality, there is an optimal closed form solution given by

$$\hat{w} = (X^T X)^{-1} X^T y \Leftrightarrow \hat{y} = X \cdot \hat{w} = X(X^T X)^{-1} X^T y = Py$$

4.1.1 Degrees of freedom

One can calculate the degrees of freedom, given by

$$df(\hat{y}) = \frac{1}{\sigma 2} \sum_{i=1}^{N} Cov(\hat{y}_i, y_i)$$

With a larger covariance, we have a stronger connection. Then, the model can approximate the data better, which makes the model more flexible (more complex).

4.1.2 R example linear regression

```r
summary(fit)
predict(fit, newdata, se.fit, interval)
coefficients(fit) # model coefficients
confint(fit, level=0.95) # CIs for model parameters fitted(fit) # predicted values
residuals(fit) # residuals
```

4.2 Poisson regression

Poisson regression is given by

$$y_i = e^{x_i^T \beta + log(t)} = t \cdot e^{x_i^T \beta}, \quad \beta = \{\beta_0, \beta_1, ..\beta_n\}$$

Where t is an offset (what the hell is the offset?). Each individual term may have its own individual offset apparently.

4.3 Ridge regresion

Linear regression has a problem; it can overfit. Thus, we wish to minimize the overfitting. We keep all predictors, but we penalize the size of the coefficients, to make the model less complex. Ridge regression is then given by

$$min_w - \sum_{i=1}^{n} (y_i - w^T x_i)^2 + \lambda \cdot |w|_2^2$$

Important to note that we here use an l2-regularization. The shrinking enables us to estimate regression coefficients even if the number of parameters exceeds the number of coefficients. λ is best decided through cross-validation.

4.3.1 R code example ridge regression

```r
model=cv.glmnet(as.matrix(covariates), response, alpha=0,family="gaussian") model$lambda.min
plot(model)
coef(model, s="lambda.min")
```

4.4 LASSO

LASSO is the same principle as ridge, but we use a linear penalizer, i.e. an l1-regularizer. Thus, LASSO is given by

$$min_w - \sum_{i=1}^{n} (y_i - w^T x_i)^2 + \lambda \cdot \sum_{i=1}^{p} |w_i|$$

which also can be written as

$$\hat{w}^{lasso} = argmin \sum_{i=1} (y_i - w^T x_i)^2$$

$$\text{s.t} \sum_{j=1}^{p} |w_i| \leq s$$

With Lasso, we obtain sparse solutions that can help us reduce the number of variables. LASSO should be used when p >> n, i.e. when the number of parameters is larger than the number of data points.

5 Variable selection

To perform variable selection, we have two options; *best subset selection* or *forward and backward stepwise selection*. In the former, one consider different subsets of the full set of features and fit models to evaluate their quality. This can be problematic if one has more than 30 different features or more. The latter, *forward and backward stepwise selection*, greedily picks the best subset by starting with 0 features and always adds the feature that best improves the selected measure (can be AIC or BIC for example).

An example on what to use is `stepAIC` in library MASS in R. Check out lab 1.

6 Classification

Given the data $D = ((X_i, Y_i), i = 1, .., N), Y_i \in C_j \in \{C_1, ..., C_K\}$, we want to classify points correctly. There are *deterministic* and *probabilistic* classifiers. Deterministic classifiers have a few disadvantages. A simple mapping is not enough sometimes, and it might be difficult to embed loss, so one has to rerun the optimizer. Also, combining several classifiers into one is more problematic.

In bayesian decision theory, we can use a *loss matrix*. Then, we transform the probability into an action. If we for example do not want many false positives, we can punish the false positives harder than the false negatives to steer the classification towards this.

```
## [1] "Loss matrix is"
```

```
##      [,1] [,2]
## [1,]    0   10
## [2,]    1    0
```

The loss matrix then steer us towards not choosing this, since the formula for expected loss can be written as

$$EL = \sum_k \sum_j \int_{R_j} L_{kj} p(x, C_k) dx \tag{1}$$

If we then only have two classes, this will be

$$EL = \int_{R_1} L_{21} p(x, C_2) dx + \int_{R_2} L_{12} p(x, C_1) dx$$

ToOne then classifies $\hat{Y} = argmax_c p(Y = c|X)$.

When one uses a loss function in two classes, the rule in classifying will be

$$L_{12} p(x, C_1) > L_{21} p(x, C_2) \rightarrow \hat{y} = C_1$$

The 0/1-loss is simply to classify to the class which is more probable.

6.1 Naive Bayes

Naive Bayes classifier is a special case of LDA makes a very strong assumption by assuming conditional independence of all variables, and classifies according to

$$P(Y = y|X) = \frac{P(X|Y = y)P(Y = y)}{\sum_j P(X|Y = y_j)P(Y = Y_j)}$$

This reduces the amount of parameters, from a full network's $2 \cdot 2^p - 2$ to only $2 \cdot p$.

Naive Bayes can accept both categorical variables as well as continuous, at the same time.

6.2 Decision trees

The idea of decision trees is to split the domain of feature set into the set of hypercubes and define the target value to be constant within each hypercube. These decision trees can work well if

There are two types of decision trees: regression trees (target is continuous) and classification trees (targets are classes).

Definitions follow below.

1. Root node
2. Nodes
3. Leaves (terminal nodes). To each leaf, a value is assigned.
4. Parent node, child node
5. Decision rules - on the edges

In classification trees, the classification probability $p_{mk} = P(Y = k|X \in R_m)$ is estimated for every class in a node.

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} \mathbb{1}(y_i = k)$$

For any leaf, a label is assigned as $k(m) = argmax_k \hat{p}_{mk}$.

To fit a regression tree, CART is used. It goes as follows.

1. Let C_0 be a hypercube containing all observations.
2. Let queue $C = \{C_0\}$
3. Pick up some C_i from C and find a variable X_j and value s that split C_j into two hypercubes

   $$R_1(j, s) = \{X|X_j \leq s\} \text{ and } R_2(j, s) = \{X|X_j > s\}$$

   and solve

   $$min_{j,s}[N_1 Q(R_1) + N_2 Q(R_2)]$$

4. Remove C_j from C and add R_1 and R_2
5. Repeat 3-4 as many times as needed (or until each cube has only 1 observation)

6.3 ROC curves

In binary classification, using ROC curves helps us analyze the performance of a classifier. The TPR is the true positive rate, i.e. the probability of deticting positives, given by

$$TPR = \frac{TP}{N^+}$$

FPR is the false positive rate, which is the probability of false alarms, given by

$$FPR = \frac{FP}{N^-}$$

N^+ is how many in total that are predicted as positive. N^- are how many in total that are predicted as negative.

Precision is given by

$$Precision = \frac{TP}{TP + FP}$$

For ROC, one plots the FPR on the x-axis and TPR on the y-axis. The performance is measured as AUC (area under the curve)

6.4 Linear classification methods

If we have continuous target variables and use the square loss, the optimal is the posterior mean

$$\hat{Y}(x) = \int yp(y|x)dy$$

6.4.1 Logistic regression

Logistic regression classifies according to the softmax function,

$$P(Y = C_i|x) = \frac{e^{w_i^T x}}{\sum_{j=1}^{K} e^{w_j^T x}}$$

In binary classification, this is written as

$$P(Y = C_i|x) = \frac{1}{1 + e^{-w_i^T x}}$$

6.4.2 LDA - Linear Discriminant Analysis

Linear discriminant analysis uses the discrimininant function

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + log \pi_k$$

$$\mu_c = \frac{1}{N_c} \sum_{i:y_i=c} x_i$$

$$\hat{\Sigma}_c = \frac{1}{N_c} \sum_{i:y_i=c} (x_i - \hat{\mu}_c)(x_i - \hat{\mu}_c)^T$$

$$\hat{\Sigma} = \frac{1}{N} \sum_{c=1}^{k} N_c \hat{\Sigma}_c$$

π_k is a prior for class k, set to its proportion in the data.

The variance matrix Σ is in practice different for all classes. Here, it is however assumed to be $\hat{\Sigma}$, i.e. same for all classes. Thus, LDA has stronger assumptions than logistic regression, but LDA and other generative classifiers handle missing data easier. LDA is however not robust to gross outliers.

7 KNN - K-nearest neighbour

7.1 KNN for classification

To classify with KNN, one has training data. For each point to classify, one take the K closest training points to the point to be classified and take the most common class among these training points. Sample code or implementation of this is given below.

```
#data is the training data,
#newData returns the predicted class probabilities
knearest = function(data, K, newData) {
  # for newData by using K-nearest neighbour approach.
  responsesX = as.matrix(data[,ncol(data)])
  responsesY = as.matrix(data[,ncol(newData)])

  X = as.matrix(data[,-c(ncol(data))])
  Y = as.matrix(newData[,-c(ncol(newData))])


  Xhat = X/sqrt(rowSums(X^2))
  Yhat = Y/sqrt(rowSums(Y^2))


  C_matrix = Xhat%*%t(Yhat)

  D_matrix = 1 - C_matrix
  #For the data test set,
  # take the distance to the training data points,
  # take the k nearest neighbours and classify them
  # as the average of the training set's labels of the k points

  # return the indices of the k nearest neighbours
  # for all different new data points, generating a matrix with the k nearest neighbou
  indicesKnearest = apply(D_matrix, 2, function(col, k){
    names(col) = seq(1, length(col), 1)

    col = sort(col)

    return(strtoi(names(col[1:k])))
  }, K)
```

```
  classifications = apply(indicesKnearest, 2, function(row) {
    # Classify according to the mean of the training data
    avg = mean(responsesX[row,])
    return(round(avg))
  })


  return(classifications)
}
```

7.2 KNN density estimation

To estimate the density, one uses

7.3 Bayesian classification

Assume classes $C_{1,..,L}$. The prediction $\hat{Y}(x) = C_l$ is given by

$$l = argmax_{i \in 1,..,L} p(C_i|x) = argmax_{i \in 1,..,L} \frac{p(x|C_i)p(C_i)}{p(x)}$$

where $\frac{p(x|C_i)p(C_i)}{p(x)} \propto p(x|C_i)p(C_i)$.

8 Model fitting

In a frequentist perspective, w is a parameter that has a true value and should be estimated by model fitting, i.e. by the likelihood function,

$$w = argmax_w \prod_{i=1}^{n} p(x_{1:n}|w)$$

In a Bayesian perspective, w is a random variable with a prior distribution $p(w)$.

8.1 Avoid overfitting

When one fits a model, we wish to avoid overfitting. This is done by either selecting proper parameter values, e.g., by regularization, or selecting a proper model type through, e.g., hold-out method or cross-validation.

9 Model Selection - how to choose an appropriate model

Many times, we have several models to choose between. To choose an appropriate model, there are several systematic approaches to choose a model. Below follows a few. First however, we must introduce the error functions.

9.1 MSE

MSE, short for mean-squared error, can be described mathematically as

$$R(Y, \hat{Y}) = \frac{1}{N} \sum_{i=1}^{N} (Y_i - \hat{Y}_i)^2$$

Is good to use for continuous response variables, e.g., for regression models or neural networks.

9.2 Misclassification rate

The misclassification rate gives us the percentage of incorrectly classified points in a set. It is given by

$$R(Y, \hat{Y}) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}(Y_i \neq \hat{Y}_i)$$

9.3 Use a loss function

To choose optimal parameter values in a model, we can use a loss function. The loss function can be described as

$$min_{\hat{f}} EL(y, \hat{y}) = min_{\hat{y}} \int L(y, \hat{y}) p(y, x|w) dx dy$$

This loss function can then help us to skew our parameters towards parameters giving us what we want. For example, if it is truly important to not have any false negatives (e.g. in identification of cancer) we can skew the parameters towards this.

9.4 Train- and test partitioning

The most simple is to divide the data into a training and test set. One fits the model with the training set and predicts on the test set. The model with the best fit on the training data, meaning the smallest error (e.g., misclassification rate or MSE) is the model that is chosen. This approach works well for moderate or large data sets.

9.5 Holdout-method

The hold-out method is simple - use a validation set to minimize the risk. The model with minimum empirical risk is selected. Risk is here defined as

$$\hat{R}(y, \hat{y}) = \frac{1}{|V|} \sum_{(X,Y) \in V} L(Y, \hat{y}(X, T))$$

9.6 Cross-validation

Cross-validation is:

1. Permute observations randomly.
2. Divide data-set in K roughly equally-sized subsets
3. Remove subset i and fit the model using the remaining data.
4. Predict the function values for subset i using the fitted model.
5. Repeat steps 3-4 for different i.
6. CV = Squared difference between observed values and predicted values (another function is possible).

10 Bias-variance tradeoff

11 PCA - Principal Component Analysis

Sometimes, the data depends on the variables which we cannot measure. For example, answers on a test depends on intelligence, and stock prices depend on the market confidence. PCA finds these hidden variables and uses these to explain variables. It is very efficient, assuming the hidden component are linear.

PCA fetches the hidden components by find the direction that maximizes the variance of the projected data (which is why it is linear). The first principal component will be the direction in the data space that maximizes the variance of the projected data. The second principal component (PC2) is the direction that maximizes the variance of the projected data after the variation along PC1 has been removed and so on. These new features, $PC_1, .., PC_m$ creates a new coordinate system. Coordinates corresponding to the last principal components are very small, and these can thus be taken away.

The aim with PCA is to minimize the distance between the original and projected data through

$$min_V \sum_{n=1}^{N} ||x_n - \hat{x}_|||^2$$

To perform PCA, we do the following.

1. Centre the data such that

$$X = ||x_1 - \overline{x}_1, x_2 - \overline{x}_2, x_p - \overline{x}_p||$$

2. Find the covariance matrix such that

$$S = \frac{1}{N} X^T X$$

3. Search for eigenvectors and eigenvalues of S.
4. Coordinates of any data point $x = (x_1, .., x_p)$ in the new coordinate system will be given by

$$z = (z_1, ..., z_n), z_i = x^T u_i$$

   In matrix form, we see it as $Z = X \cdot U$

5. Discard principle components after some M

6. The new data will have dimension NxM instead of Nxp
7. The approximate data is retrieved by

$$X' = ZU_M^T$$

12 Probabilistic PCA

In probabilistic PCA, we have z_i-latent variables, x_i-observed variables

$$z \sim N(0, I)$$

$$x|z \sim N(x|Wz + \mu, \sigma^2 I)$$

Alternatively, it could be written as

$$z \sim N(0, I), \quad x = \mu + Wz + \epsilon, \epsilon \sim N(0, \sigma^2 I)$$

The observed data (X) is obtained by rotation, scaling and translation of standard normal distribution (Z) and adding some noise.

When we want to extract Z from X, we consider that

$$x \sim N(\mu, C)$$
$$C = WW^T + \sigma^2 I$$

The rotation invariance assumes that x was generated from z' = Rz, $RR^T = I$, p(x) does not change.

$$x|z' \sim N(x|Wz' + \mu, \sigma^2 I)$$

Estimation parameters: ML (Maximum likelihood). The ML estimates given by

$$\mu_{ML} = \overline{x}$$
$$W_{ML} = U_M(L_M - \sigma_{ML}^2 I)^{\frac{1}{2}} R$$

where U_M is the matrix of M eigenvectors, L_M diagonal matrix of M eigenvalues and R is any orthogonal matrix.

To estimate Z, use the mean of the posterior

$$\hat{z} = (W_{ML}^T W_{ML} + \sigma_{ML}^2 I)^{-1} W_{ML}^T (x - \mu)$$

If $R = I$ and $\sigma^2 = 0$, we get standard PCA components scaled by inverse root of eigenvalues.

$$Z = XUL^{-\frac{1}{2}}$$

Probabilistic PCA has several advantages.

- More settings to specify, making it more flexible
- It can be faster when M « p
- Missing values can be handled.

- M can be derived if a bayesian version is used.
- Probabilistic PCA can be applied to classification problems directly.
- Probabilistic PCA can generate new data.

13 ICA (Independent Component Analysis)

Since probabilistic PCA does not capture latent factors due rotation invariance, we can choose a distribution which is not rotation invariant. This will give us unique latent factors, and we choose a non-gaussain $p_i(z) = p(z)$, and we assume latent features are *independent*.

ICA assumes $x = \mu + Wz + \epsilon, \epsilon \sim N(0, \Sigma)$

We can estimate with maximum likelihood (V = W^{-1}) and if we assume it to be noise-free we get

$$max_V \sum_{i=1}^{n} \sum_{j=1}^{p} log(p_j(v_j^T x_i))$$

14 SVM - Support Vector Machines

15 Kernels

Instead of using something similar to a decision tree, we can use the moving window rule to give equal weight to all the points in a ball, and classify according to

$$y_k(x) = \begin{cases} 0 & \text{if } \sum_n \mathbb{1}(t_n = 1)k(\frac{x-x_n}{h}) \leq \sum_n \mathbb{1}(t_n = 0)k(\frac{x-x_n}{h}) \\ 1 & \text{if } tails \end{cases}$$

where $k(\frac{x-x_n}{h})$ is a kernel. There are several types of kernels.

- Gaussian kernel: $k(u) = exp(-||u||^2)$ where $||\cdot||$ is the euclidian norm.
- Cauchy kernel: $k(u) = \frac{1}{(1+||u||^{D+1})}$
- Epanechnikov kernel: $k(u) = (1 - ||u||^2)\mathbb{1}(||u|| \leq 1)$
- Moving window kernel: $k(u) = \mathbb{1}(u \in S(0, 1))$

We can use a kernel for classification through the formulas

$$y_C(x) = \frac{\sum_{x_n \in C(x,h)} t_n}{|\{x_n \in C(x, h)\}|}$$

or

$$y_S(x) = \frac{\sum_{x_n \in C(x,h)} t_n}{|\{x_n \in S(x, h)\}|}$$

or

$$y_k(x) = \frac{\sum_n k(\frac{x-x_n}{h}) \cdot t_n}{\sum_n k(\frac{x-x_n}{h})}$$

One can build one's own kernel easily by simply combining kernels.

16 Neural networks

16.1 Neurons

16.2 Activation functions

16.2.1 ReLU-function

16.2.2 tanh-function

16.3 Weights

16.4 How many layers?

17 Other important aspects to know about

17.1 Basic ML ingredients

In Machine Learning, there are a few basic components always involved.

1. One has data D with features $X_1, .., X_n$ and target variables $Y_1, .., Y_n$.
2. A model $P(x|w_1, .., w_k)$ or $P(y|x, w_1, .., w_k)$ where $w_{1:k}$ are parameters.
3. A learning procedure to learn and train the parameters. For example, Maximum likelihood or bayesian estimation, or gradient descent.
4. When one has trained the model, one predicts new data, X^{new}, by using the trained/fitted model.

In probabilistic models, one assumes a distribution of the data.

Now, given a dataset D one can assume a Frequentist or Bayesian approach.

17.1.0.1 A frequentist approach

will try to find out which combination of parameter values that fits the data best. The core foundation is the maximum likelihood function, given by

$$p(D|w) = \prod_{i=1}^{n} p(X_i|w)$$

$$p(D|w) = \prod_{i=1}^{n} p(Y_i|X_i, w)$$

17.1.0.2 A bayesian approach

assumes that all parameters are random variables, and rather not see that each parameter has a true value, but rather that each parameter has a distribution of probabilities. One can also incorporate prior, subjective beliefs of what the most likely values are. The bayesian approach's core formula is the bayes theorem, given by

Distribution name	Annotation	pdf	$E(X)$	$V(X)$
Bernoulli	$X \sim Bern(p)$	$\mathrm{p(X = 1) = p}$	p	$1-p$
Binomial	$X \sim Bin(n,p)$	$p(X=r) = \frac{n!}{(n-r)!r!}p^r(1-p)^{n-r}$	np	$np(1-p)$
Poisson	$X \sim Pois(\lambda)$	$p(X=r) = \frac{\lambda^r e^{-\lambda}}{r!}$	λ	λ
Normal	$X \sim N(\mu, \sigma^2)$	$p(X=x) = \frac{1}{\sqrt{2\pi\sigma^2}}exp(-\frac{(x-\mu)^2}{2\sigma^2})$	μ	σ^2

$$p(w|D) = \frac{p(D|w)p(w)}{p(D)} \propto p(D|w)p(w)$$

17.2 Curse of dimensionality

The curse of dimensionality states that a point has no "near" neighbours in high dimensions. Thus, using class labels of neighbours in higher dimensions can be clearly misleading, and methods such as KNN-classification might be rendered useless. It is however not hopeless; real data usually has lower effective dimensions, and one can use dimensionality reduction techniques, such as PCA. Interpolation can also be used - a small change in X should lead to a small change in Y.

17.3 Basic statistics - lecture 2

Expected value - discrete

$$E(X) = \sum_{D_X} X \cdot P(X)$$

Expected value - continuous

$$E(X) = \int_{D_X} X \cdot P(X) dX$$

Variance (assuming independence of variables)

$$V(X) = E(X - E(X))^2 = E(X^2) - E(X)^2$$

Sum rule

$$P(X) = \sum_{D_Y} P(X,Y)$$

$$P(X) = \int_{D_Y} P(X,Y) dY$$

Product rule

$$P(X,Y) = P(X|Y)P(Y) = P(Y|X)P(Y)$$

17.4 Conventional distributions

18 Uncertainty intervals

There are a few types of intervals.

1. Confidence intervals (frequentist).

2. Credible intervals (Bayesian).
3. Prediction intervals (models).

19 Good R code commands

```r
# Delete a column by name
data$name = c()
# or delete several columns by name
names_to_del = c("name1", "name2")
data = data[,!colnames(data) %in% names_to_del]

# Divide into training and test set as per Oleg
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]
test=data[-id,]

# Hold-out method: divide into training, validation and test set
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.4))
train=data[id,]
id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.3))
valid=data[id2,]
id3=setdiff(id1,id2)
test=data[id3,]

# MSE function

# Misclassification rate function
missclass=function(X,X1){
  n=length(X)
  return(1-sum(diag(table(X,X1)))/n)
}
# ROC function

# AUC function

# Classification with NN
# nn is basically a logistic regression model
library(neuralnet)
nn = neuralnet(formula=formula, data=data.crabs, hidden=0, err.fct="ce", linear.output = FALSE)


crossValidationSplits <- function(dataSet, k, seed = 123) {
  smp_size = floor(nrow(dataSet)/k)

  ## set the seed to make your partition reproducible
```

```
  set.seed(seed)

  folds <- list()

  for (i in 1:k) {
    newFold <- sample(seq_len(nrow(dataSet)), size = smp_size)
    folds[[i]] = data.frame(dataSet[newFold,])
    dataSet <- dataSet[-newFold,]
  }
  return(folds)

}

# Divides folds generated by split_into_folds into a training and validation set
CVtrainAndValidationSet <- function(folds, splitIndex = -1) {
  if (splitIndex == -1 || splitIndex > length(folds)) {
    testIndex = floor(runif(1, min = 1, max = length(folds)))
    test = folds[[testIndex]]
    train = folds
    train[[testIndex]] = NULL
  } else {
    test = folds[[splitIndex]]
    train = folds
    train[[splitIndex]] = NULL
  }
  trainSet = train[[1]]
  for (i in 2:length(train)) {
    trainSet = rbind(trainSet, train[[i]])
  }

  return(list(trainSet,test))
}
```

20 Lab 1

20.1 Assignment 1 - Spam classification, KNN and Logistic Regression

First, we initialize some functions and load the data.

```
library(readxl)
data =
  read_excel(
    "~/Desktop/Plugg_Lkpg/HT18/TDDE01/Tentaplugg/Lab_repetitions/Lab1/spambase.xlsx"
    )
data$Spam = factor(data$Spam)
log_reg = function(data, fitModel, pLimit = 0.5) {
  # Specify response in predict,
  fits = predict(fitModel, data)

  probabilities = fits

  classifications = ifelse(probabilities > pLimit, 1,0)
```

```
  return(classifications)
}

misclassificationRate = function(confMatrix) {
  return(1 - sum(diag(confMatrix))/sum(confMatrix))
}
```

20.1.1 Task 1 - divide data

Loading in the data and setting the seed and dividing it into training and test sets is a trivial task.

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

20.1.2 Task 2 - fit logistic regression model

In this task, our mission was to classify the spam using logistic regression and report the confusion matrices, which can be found in the output below.

```
#Task 2

fit = glm(Spam ~ ., data = train, family = "binomial")
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
#summary(fit)
fits = predict(fit, train)
classificationsTrain = log_reg(train, fit)
classificationsTest = log_reg(test, fit)

# Uncomment code below for the classification rates
#print("misclassification rate train")
#table(train$Spam, classificationsTrain)
#misclassificationRate(table(train$Spam, classificationsTrain))
#print("misclassification rate test")
#table(test$Spam, classificationsTest)
#misclassificationRate(table(test$Spam, classificationsTest))
```

We get an accuracy of 83 % on the training set, and a test set accuracy of 81 %. This is quite a good accuracy, although not perfect. However, they are very alike, indicating that it is probably not overfitted.

20.1.3 Task 3 - Change classification threshold to p = 0.9 from p = 0.5

In task 3, we changed the classification probability to ensuring it has 0.9 as probability to be 1 to be classified as 1.

```
## Task 3

classificationsTrain = log_reg(train, fit, pLimit = 0.9)
classificationsTest = log_reg(test, fit, pLimit = 0.9)
```

```
#print("misclassification rate train")
#table(train$Spam, classificationsTrain)
#misclassificationRate(table(train$Spam, classificationsTrain))
#print("misclassification rate test")
#table(test$Spam, classificationsTest)
#misclassificationRate(table(test$Spam, classificationsTest))
```

We see that the results are slightly worse, although not much worse. This is because some data points that lies within the interval $0.5 < p \leq 0.9$ are classified incorrectly, as they most likely actually are 1. However, the stricter rule prohibits these from being classified correctly, thus yield more false negatives. When choosing the limit 0.9, we get a significantly higher classification rate. This means that the model is not very robust and has many probabilites between 0.5 and 0.9.

20.1.4 Task 4 - KNN with K = 30

Here, the task was to use the standard classifier `kknn()` with K = 30. The misclassification rate here is higher, indicating that logistic regression is probably a better classification method in this case.

```
# Task 4

library("kknn")

# Fit with kknn. Fit the training data with k = 30
kkn.fit = kknn(Spam ~., train = train, test = train, k = 30)

#print("misclassification rate train")
#table(train$Spam, kkn.fit$fitted.values)
#misclassificationRate(table(train$Spam, kkn.fit$fitted.values))

# Fit with kknn. Fit the test data with = 30
kkn.fit = kknn(Spam ~., train = train, test = test, k = 30)

#print("misclassification rate test")
#table(test$Spam, kkn.fit$fitted.values)
#misclassificationRate(table(test$Spam, kkn.fit$fitted.values))
```

20.1.5 Task 5 - KNN with K = 1, overfitting

Changing to K = 1 worsened the results, yielding a slightly higher, although not very different, misclassification rate. We get more false positives than before. When we use 1, we overfit. This is because it picks its own point, as this is the closest one. Thus, we receive an extreme accruacy if the points is already seen.

```
# Task 5

# Fit with kknn. Fit the training data with k = 1
kkn.fit = kknn(Spam ~., train, train, k = 1)

#print("misclassification rate train")
#table(train$Spam, kkn.fit$fitted.values)
#misclassificationRate(table(train$Spam, kkn.fit$fitted.values))

# Fit with kknn. Fit the test data with k = 1
kkn.fit = kknn(Spam ~., train, test, k = 1)
```

```
#print("misclassification rate test")
#table(test$Spam, kkn.fit$fitted.values)
#misclassificationRate(table(test$Spam, kkn.fit$fitted.values))
```

We clearly overfit on the training data with K = 1, yielding a worse result on the test data.

20.2 Assignment 2 - Prior, Maxmimum likelihood and inference

20.2.1 Task 1 - import data

First, we import the data.

```
# Task 1
library(readxl)
data = read_excel("~/Desktop/Plugg_Lkpg/HT18/TDDE01/Tentaplugg/Lab_repetitions/Lab1/machines.xlsx")
```

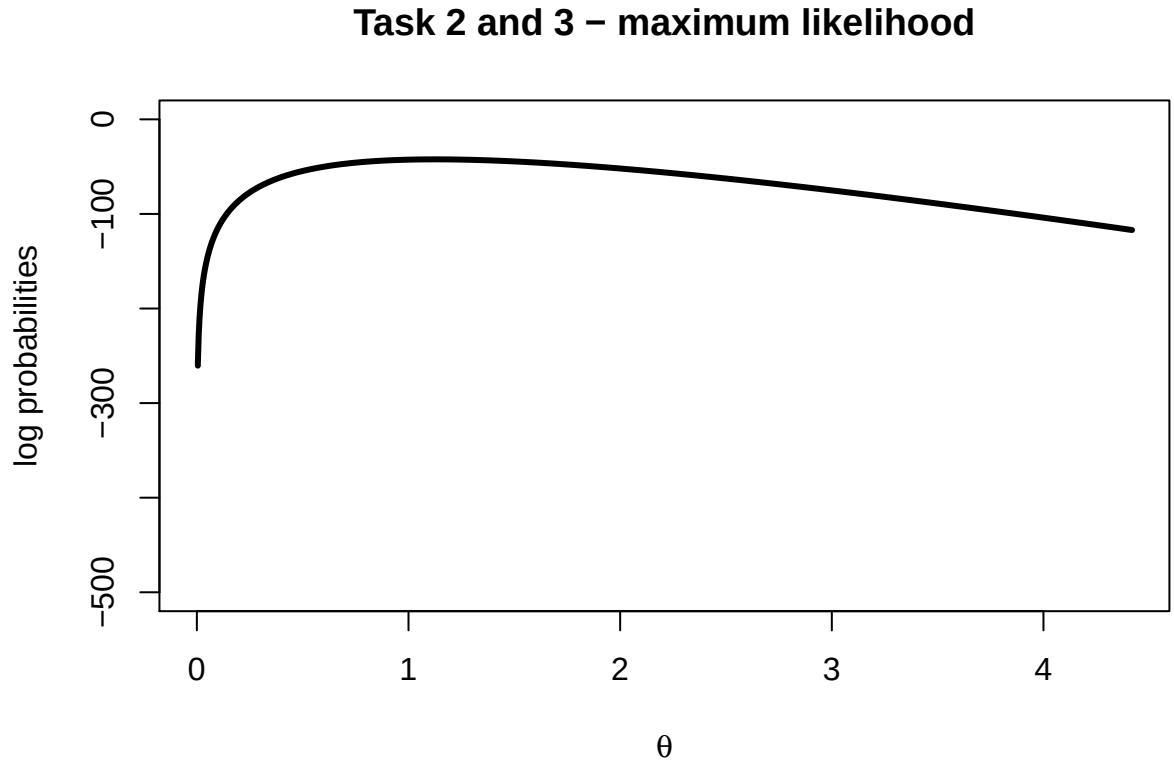
20.2.2 Task 2 - Compute log likelihood of data

Here, the task was to compute the log likelihood of $p(x|\theta) = \theta e^{-\theta x}$. The likelihood can be described as which can be described as

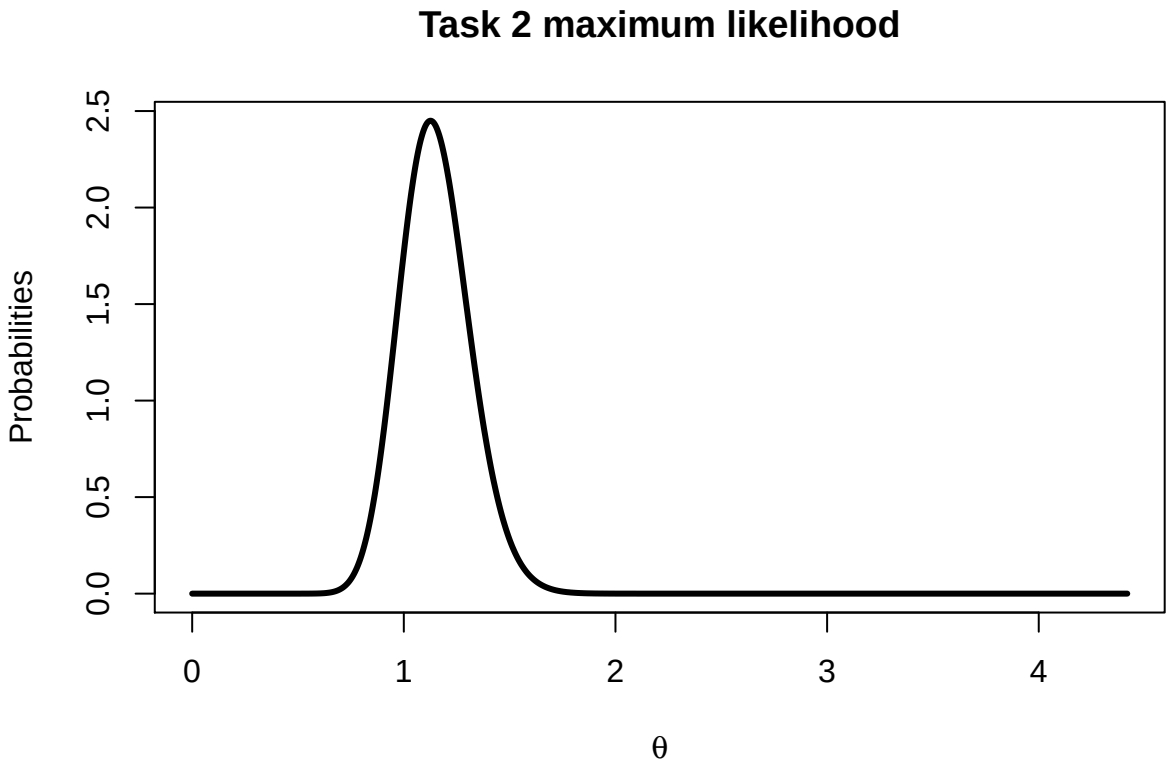
$$\sum_{i=1}^{n} log\big(p(x_i|\theta)\big) = \sum_{i=1}^{n} log\big(\theta e^{-\theta x_i}\big) = \sum_{i=1}^{n} log\big(\theta\big) - \theta x_i = nlog(\theta) - \theta \sum_{i=1}^{n} x_i$$

This results in the plot given below.

```
# Task 2

# Log likelihood function. Computes log likelihood for exponential distribution given data x and param
logLikelihood = function(theta, x) {
  return(sum(log(theta) - theta*x))
}

# Normalizes with the scale. Scale is 1/(diff between values in x-grid)
normalize = function(scale, probs) {
  return(scale*probs/sum(probs))
}

thetaGrid = seq(0,max(data), length = 1000)
scale = 1/(thetaGrid[2] - thetaGrid[1])
logProbs = apply(as.matrix(thetaGrid), 1, logLikelihood, data$Length)

# Normalize the probabilities to get a reasonable prob distribution
probsNormalizedTask2 = normalize(scale, exp(logProbs))
plot(x = thetaGrid, y = logProbs,
     type = "l", lwd = 3,
     xlab = expression(theta),
     main = "Task 2 and 3 - maximum likelihood", ylab = "log probabilities",
     ylim = c(-500,0))
```




```
plot(x = thetaGrid, y = probsNormalizedTask2,
     type = "l", lwd = 3,
     xlab = expression(theta),
     main = "Task 2 maximum likelihood",
     ylab = "Probabilities")
```



```r
maxTheta = thetaGrid[which.max(probsNormalizedTask2)]
print(paste("Max value of theta is ", maxTheta))
```

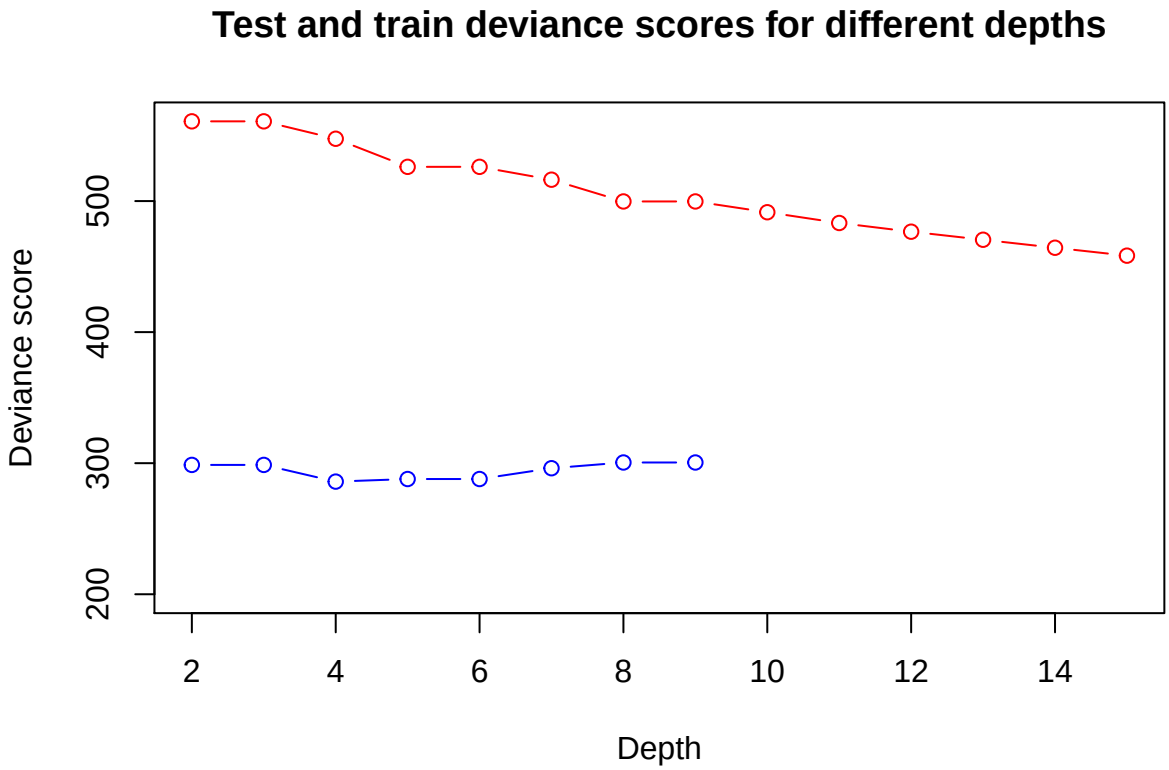
```
## [1] "Max value of theta is  1.12779300909544"
```

We also see that the most likely value of θ is around 1.128.

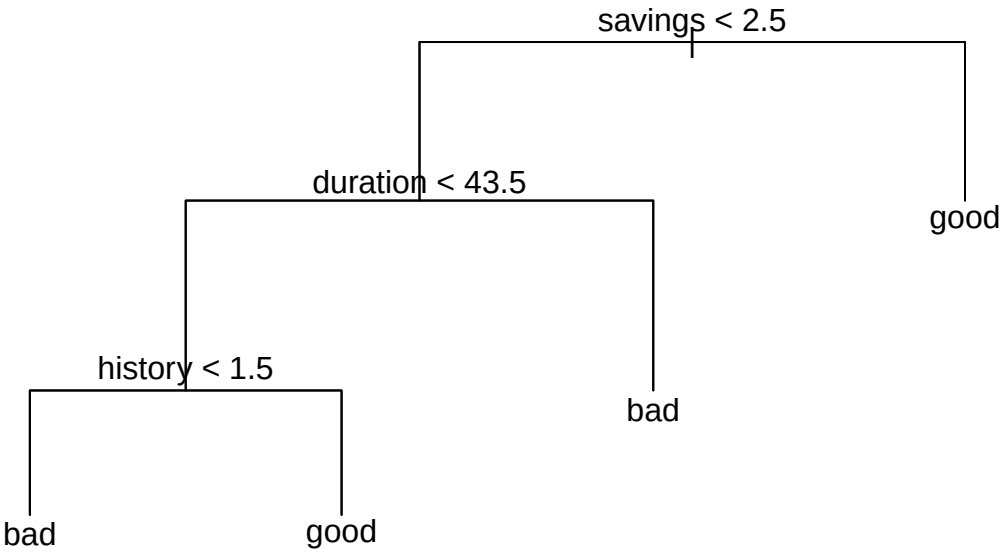
20.2.3 Task 3 - log likelihood with only 6 points (instead of 48)

In task 3, we use fewer observations, but use the maximum likelihood function just as before.

```r
# Task 3
# Cut the data to only include the 6 first observations
cutData = data$Length[1:6]
logProbsTask3 = apply(as.matrix(thetaGrid), 1, logLikelihood, cutData)
probsNormalizedTask3 = normalize(scale, exp(logProbsTask3))

# First plot the log prob for the first task, then the new
plot(x = thetaGrid, y = logProbs,
     type = "l", lwd = 3,
     xlab = expression(theta),
     main = "Task 2 and 3 - maximum likelihood", ylab = "log probabilities",
     ylim = c(-500,0))
lines(x= thetaGrid, y = logProbsTask3, type = "l", lwd = 3, col = "green")

legend("bottomright",
       lty = rep(1,2),
       col = c("black", "green"),
       lwd = c(3,3),
       legend = c(              expression(paste("p(",x[1:48],"|",theta,")")),
                  expression(paste("p(",x[1:6],"|",theta,")")))
       )
```




```
# First plot the prob dist for the first task, then the new probability distribution
plot(x = thetaGrid, y = probsNormalizedTask2,
     type = "l", lwd = 3,
     xlab = expression(theta),
     main = "Task 2 and 3 - maximum likelihood", ylab = "Probabilities")
lines(x= thetaGrid, y = probsNormalizedTask3, type = "l", lwd = 3, col = "green")

legend("topright",
       lty = rep(1,2),
       col = c("black", "green"),
       lwd = c(3,3),
       legend = c(              expression(paste("p(",x[1:48],"|",theta,")")),
                  expression(paste("p(",x[1:6],"|",theta,")")))
       )
```



This yields a distribution around a different value of θ, with a lower probability. This is because we have fewer observations, and we can thus not be as sure as in task 2.

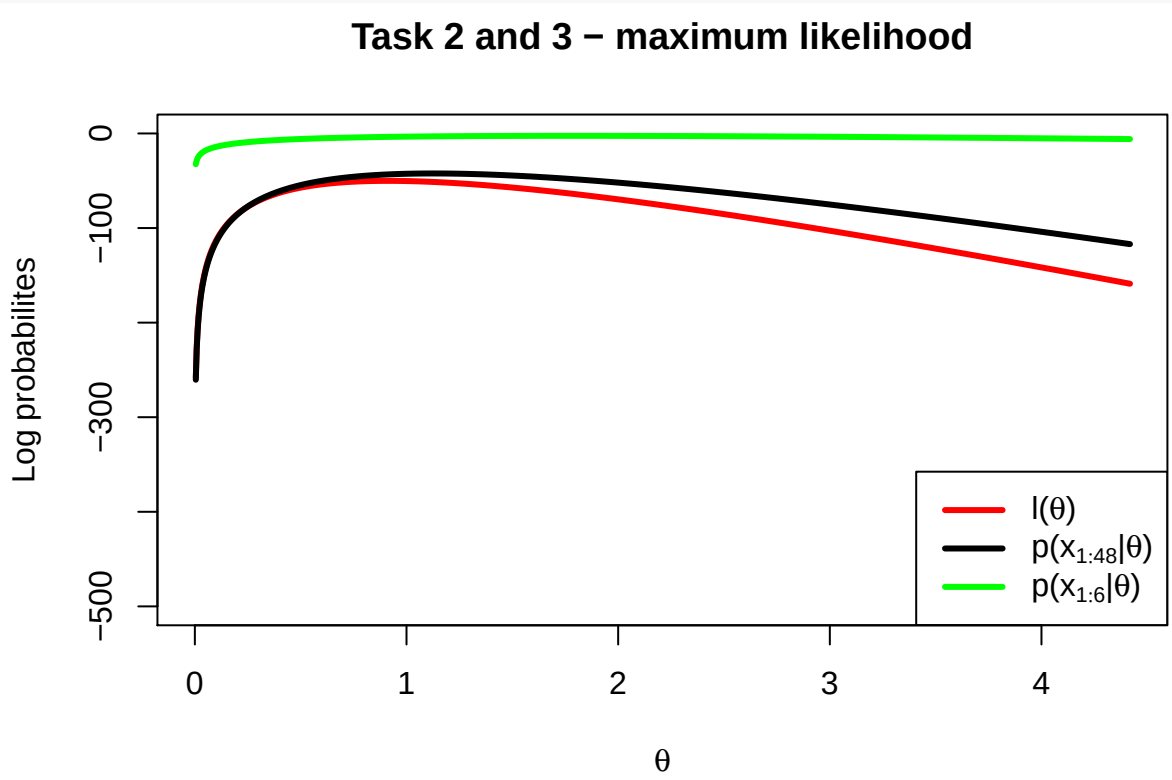
20.2.4 Task 4 - Incorporate a prior, exponentially distributed θ

In task 4, we want to use a prior $p(\theta) = \lambda e^{-\lambda\theta}$, in other words, that θ also follows an exponential distribution. With this and the maximum likelihood function, we create the posterior distribution $l(\theta)$ and obtain a new distribution for θ. We have that

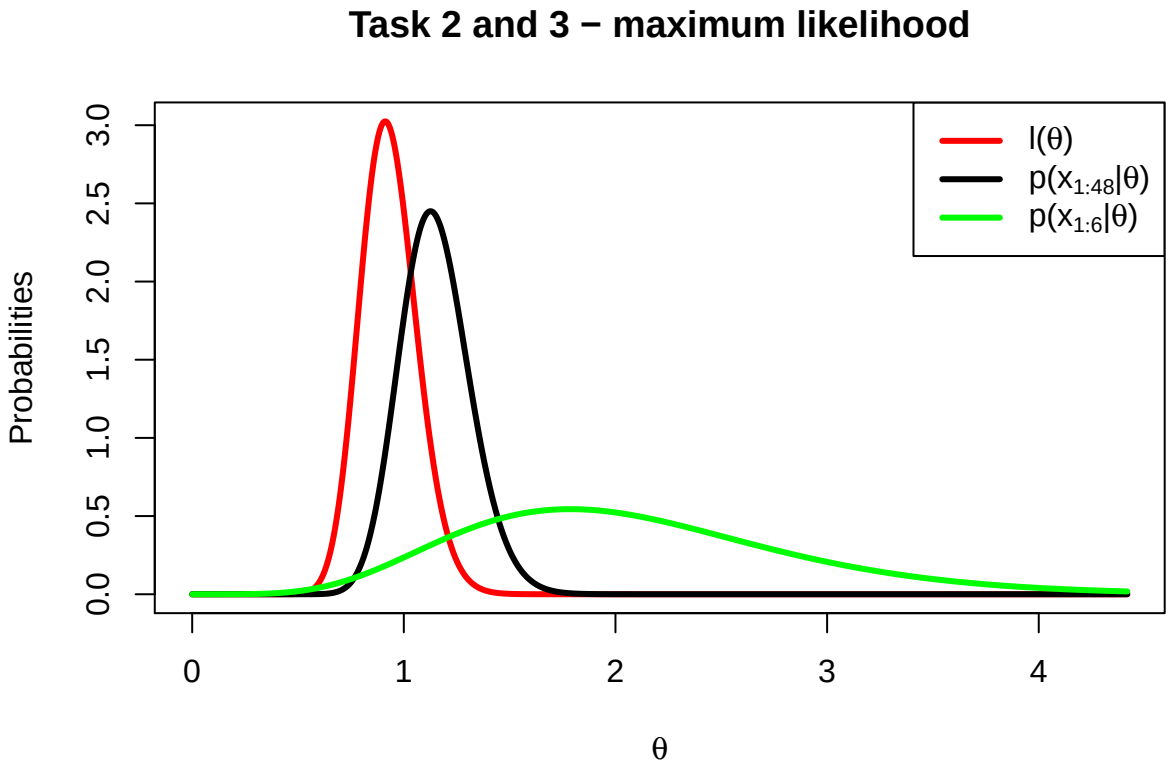
$$l(\theta) = p(x_{1:n}|\theta)p(\theta) = \Big(\prod_{i=1}^{n} \theta \cdot e^{-\theta x_i}\Big) \cdot \lambda e^{-\lambda\theta}$$

```
# Task 4

# l(theta) function. Mathematically, l(theta) = p(x_{1:n}|theta)p(theta)
l_theta = function(theta, data, lambda = 10) {


  log_dpois = log(lambda*exp(-lambda*theta))
  return(logLikelihood(theta, data) + log_dpois)
}

logProbsTask4 = apply(as.matrix(thetaGrid), 1, l_theta, data)

probsNormalizedTask4 = normalize(scale, exp(logProbsTask4))

plot(x = thetaGrid, y = logProbsTask4,
     type = "l", col = "red",
     lwd = 3, xlab = expression(theta),
     main = "Task 2 and 3 - maximum likelihood", ylab = "Log probabilites",
     ylim = c(-500,0))
lines(x=thetaGrid, y = logProbs, lwd = 3)
lines(x= thetaGrid, y = logProbsTask3, type = "l", lwd = 3, col = "green")

legend("bottomright",
```

```
       lty = rep(1,3),
       col = c("red", "black", "green"),
       lwd = c(3,3,3),
       legend = c(expression(paste("l(",theta,")")),
                  expression(paste("p(",x[1:48],"|",theta,")")),
                  expression(paste("p(",x[1:6],"|",theta,")")))
       )
```




```
plot(x = thetaGrid, y = probsNormalizedTask4,
     type = "l", col = "red",
     lwd = 3, xlab = expression(theta),
     main = "Task 2 and 3 - maximum likelihood", ylab = "Probabilities")
lines(x=thetaGrid, y = probsNormalizedTask2, lwd = 3)
lines(x= thetaGrid, y = probsNormalizedTask3, type = "l", lwd = 3, col = "green")

legend("topright",
       lty = rep(1,3),
       col = c("red", "black", "green"),
       lwd = c(3,3,3),
       legend = c(expression(paste("l(",theta,")")),
                  expression(paste("p(",x[1:48],"|",theta,")")),
                  expression(paste("p(",x[1:6],"|",theta,")")))
       )
```

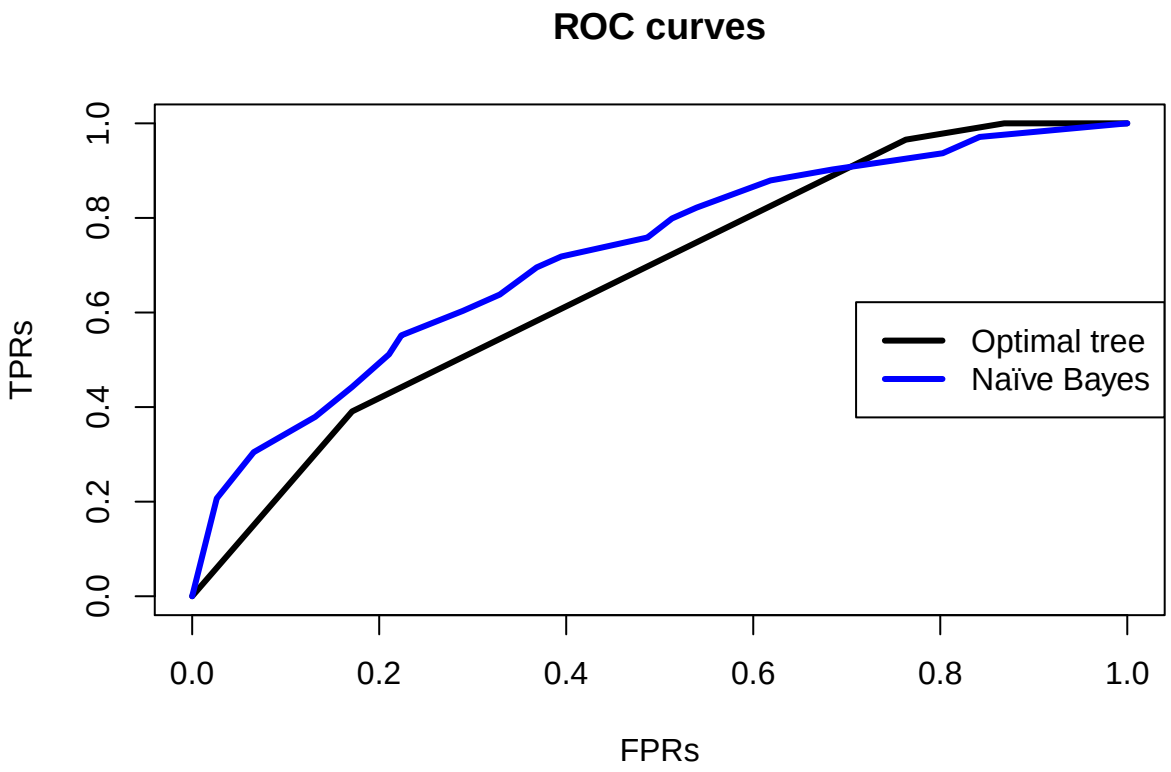


This is has an even higher probability to be true. Do note that these are not completely correctly normalized, but the relation between the three is correct.

20.2.5 Task 5 - Simulate from most probable distribution

In this task, we were supposed to simulate some observations from $p(x|\theta) = \theta e^{-\theta x}$. Since this is an exponential distribution, we can simply generate sample from the built-in function `rexp()` in R.

```
# TASK 5

# here we use maxTheta to sinulate
set.seed(12345)
sims = rexp(n=50, rate=maxTheta)

par(mfrow=c(2,1))
hist(sims, main = "Simulations",
     col = "blue", xlim = c(0,5),
     ylim = c(0,20), xlab = "Simulations", breaks = 10)
hist(data$Length, main = "Actual values",
     col = "green", xlim = c(0,5),
     ylim = c(0,20), xlab = "Actual length values", breaks = 10)
```




```
par(mfrow=c(1,1))
```

We can see that the actual values and the simulated values follows a similar distribution. This would be clearer with more samples, but is still quite clear with only 50 samples for each.

20.3 Assignment 3 - Cross-validation, linear regression and best subset selection

The task here was to use cross-validation to perform best subset selection on the swiss dataset for linear regression.

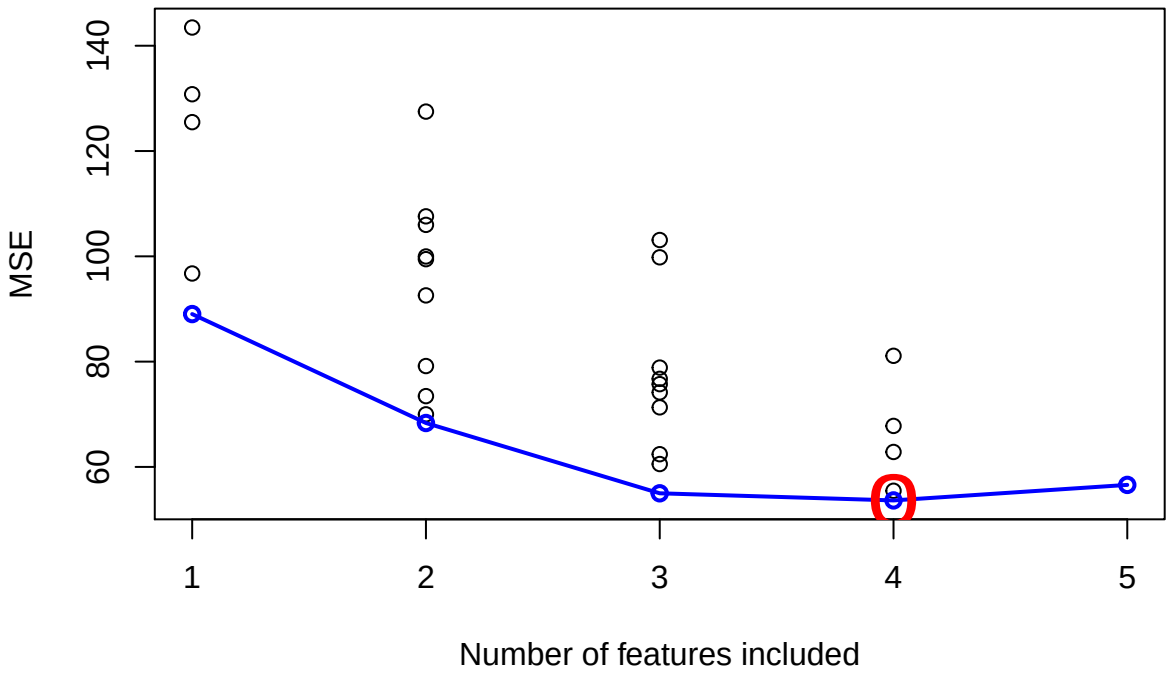
```
# Splits the data into K folds. Default set to 5.
# X is a matrix containing features
# Y is a vector (or Nx1 matrix) containing response variables
split_into_folds = function(X, Y, K = 5, seed = 12345) {
  smp_size = floor(nrow(X)/K)

  indices_chosen = as.numeric()
  folds = list()

  set.seed(seed)
  for (k in 1:(K-1)) {
    indices = sample(x=seq(1,nrow(X),1)[!(seq(1,nrow(X),1) %in% indices_chosen)], size=smp_size, replac
    x = X[indices,]
    y = Y[indices]
    fold = as.matrix(data.frame(x,y))
    folds[[k]] = fold
    indices_chosen = append(indices_chosen, indices)
  }

  # last fold
```

```
  indices = seq(1,nrow(X),1)[!seq(1,nrow(X),1) %in% indices_chosen]
  x = X[indices,]
  y = Y[indices]
  fold = as.matrix(data.frame(x,y))
  folds[[K]] = fold
  return(folds)
}

# Divides folds generated by split_into_folds into a training and validation set
CVtrainAndValidationSet <- function(folds, splitIndex = -1) {
  if (splitIndex == -1 || splitIndex > length(folds)) {
    testIndex = floor(runif(1, min = 1, max = length(folds)))
    test = folds[[testIndex]]
    train = folds
    train[[testIndex]] = NULL
  } else {
    test = folds[[splitIndex]]
    train = folds
    train[[splitIndex]] = NULL
  }
  trainSet = train[[1]]
  for (i in 2:length(train)) {
    trainSet = rbind(trainSet, train[[i]])
  }

  return(list(trainSet,test))
}


# Performs feature selection for linear regression.
# X are the features, y is the response.
# Nfolds is how many folds to use in CV.
feature_selection = function(X, y, Nfolds = 5) {
  X_new = X
  # Store all possible combinations of 2^5 combinations = 32 different possible combinations

  combinations = matrix(NA, nrow=(2^5), ncol = 6)

  i = 1
  for (i1 in c(0,1)) {
    for (i2 in c(0,1)) {
      for (i3 in c(0,1)) {
        for (i4 in c(0,1)) {
          for (i5 in c(0,1)) {
            combinations[i,] = c(i1,i2,i3,i4,i5,1)
            i = i + 1
          }
        }
      }
    }
  }
  # Permute
```

```
  k = Nfolds
  set.seed(12345)

  folds = split_into_folds(X=X,Y=y,K=5)
  cv_err = matrix(NA, nrow=(2^5), ncol = 1)
  # Loop through combinations
  for (i in 1:(2^5)) {
    cv_err[i] = CV(folds, combination = combinations[i,])
  }

  nr_feats = apply(combinations, 1, function(comb) {
    return(sum(comb) - 1)
  })

  plot(x=nr_feats[2:32], y = cv_err[2:32], ylab="MSE",
       xlab = "Number of features included",
       main = "MSE error plotted against number of features")

  smallest_error = as.numeric()
  for (j in 1:5) {
    smallest_error[j] = min(cv_err[which(nr_feats == j),])
  }
  lines(x=1:5, y=smallest_error, type = 'o', col = "blue", lwd = 2)
  points(x=nr_feats[which.min(cv_err)],
         y = cv_err[which.min(cv_err),],
         pch = "o", cex= 3,
          col = "red")
  optimal_subset = c(colnames(X),"Fertility")[combinations[which.min(cv_err),] == 1]

  return(optimal_subset)
}

MSE = function(y, y_hat) {
  n = length(y)
  return(mean((y - y_hat)^2))
}

# Send in response variable name alone and feature names alone. Returns the formula
get_formula = function(response, feats) {
  f = paste(c(response, paste(best_subset, collapse = ' + ')), collapse = " ~ ")
}

CV = function(folds, combination, k = 5) {
  errors = as.numeric()
  for (j in 1:k) {
    # Pick v
    train_test = CVtrainAndValidationSet(folds= folds,splitIndex = j)
    train = train_test[[1]]
    test = train_test[[2]]


    # Remove the columns not included
    X_new_train = as.matrix(train[,combination == 1])
```

```
    y_train = X_new_train[,ncol(X_new_train)]
    X_new_test = as.matrix(test[,combination == 1])
    y_test = X_new_test[,ncol(X_new_test)]

    X_new_train = matrix(c(rep(1,nrow(X_new_train)),
                           (X_new_train[,-ncol(X_new_train)])),
                         byrow = FALSE, nrow = nrow(X_new_train))
    X_new_test = matrix(c(rep(1,nrow(X_new_test)),
                          (X_new_test[,-ncol(X_new_test)])),
                        byrow = FALSE, nrow = nrow(X_new_test))


    if (dim(X_new_train)[2] >= 2) {
      # Fit model
      betas = solve(t(X_new_train)%*%X_new_train)%*%t(X_new_train)%*%as.matrix(y_train)
      y_hat = X_new_test%*%betas
      errors[j] = MSE(y=y_test, y_hat=y_hat)
    }

  }
  # Return the error
  return(mean(errors))
}
set.seed(12345)
data(swiss)
swiss_data = swiss
y = swiss_data$Fertility
X = as.matrix(swiss_data[,-c(1)])
best_subset = feature_selection(X,y)
```

MSE error plotted against number of features



```
print(paste("best formula is:", get_formula("Fertility", best_subset[-5])))
```

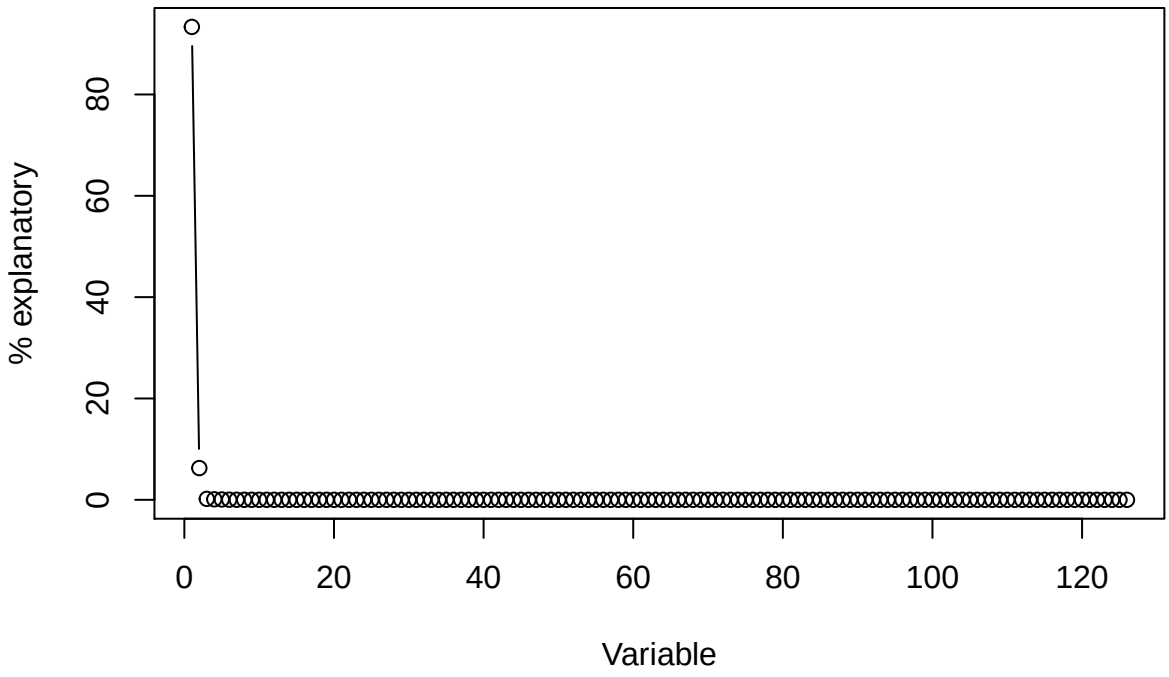
```
## [1] "best formula is: Fertility ~ Agriculture + Education + Catholic + Infant.Mortality + Fertility"
```

20.4 Assignment 4 - Linear regression and regularization

20.4.1 Task 1 - Plotting

Plotting the protein versus the moisture clearly shows that the relation between these can be described well using a linear model.

```
# Task 1
library(readxl)
data = read_excel(
  "~/Desktop/Plugg_Lkpg/HT18/TDDE01/Tentaplugg/Lab_repetitions/Lab1/tecator.xlsx")

plot(x=data$Moisture, y = data$Protein,
     xlab = "Moisture",
     ylab = "Protein",
     main = "Comparison protein and moisture")
```



20.4.2 Task 2 - Describing a probabilistic polynomial model

In this task, we are supposed to describe a probabilistic model that describes M_i. This can be described as Here, it is appropriate to use the MSE criterion, since . The MSE criterion can be described as

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

and it is appropriate to use for linear regression, since it comes from the maximum likelihood function if the error is normally distributed, which it is, according to instructions in this case.

To be able to describe different probabilistic models $M_i \quad \forall i \in \{1, .., 6\}$ I created a few functions to be able to generate these models. The models can mathematically be described as

$$M_i = \left\{ y_m = \sum_{j=0}^{i} \beta_0 x^j + \varepsilon, \quad \varepsilon \sim N(0, \sigma^2) \quad \forall m \in \{1, .., m\} \right\}$$

where m is the number of datapoints.

```
# Task 2


# Mean squared error function. y are actual values, y_hat are predictions
MSE = function(y, y_hat) {
  n = length(y)
  return((1/n)*sum((y - y_hat)^2))
}

# Converts vector x to a matrix of its polynomial up to degree specified by degree.
# Will include column of ones if include_intercept = TRUE
convert_to_poly = function(x, degree, include_intercept = FALSE) {
  if (include_intercept == TRUE) {
    x_new = matrix(NA, nrow=nrow(as.matrix(x)), ncol = (degree+1))
    for (i in 1:(degree+1)) {
      x_new[,i] = x^(i-1)
    }
  } else {
   x_new = matrix(NA, nrow=nrow(as.matrix(x)), ncol = degree)
   for (i in 1:degree) {
      x_new[,i] = x^i
      }
  }
  return(x_new)
}

# Returns the training and test error of a polynomial.
# X_train is a one-dim vector, so is y_train.
# degree is the polynomial degree.
# The function will also plot it! Thus, call a plot before calling this function or remove "lines " lin
polynomial_model = function(X_train, y_train, X_test, y_test, degree, col = "blue") {

  x_train = convert_to_poly(X_train,degree)

  x_test = convert_to_poly(X_test,degree)
  y = y_train
  train = data.frame(x_train,y)
  y = y_test
  if (degree == 1) {
    x_train = x_test
    test = data.frame(x_train, y)
  } else {
    test = data.frame(x_test, y)
  }
```

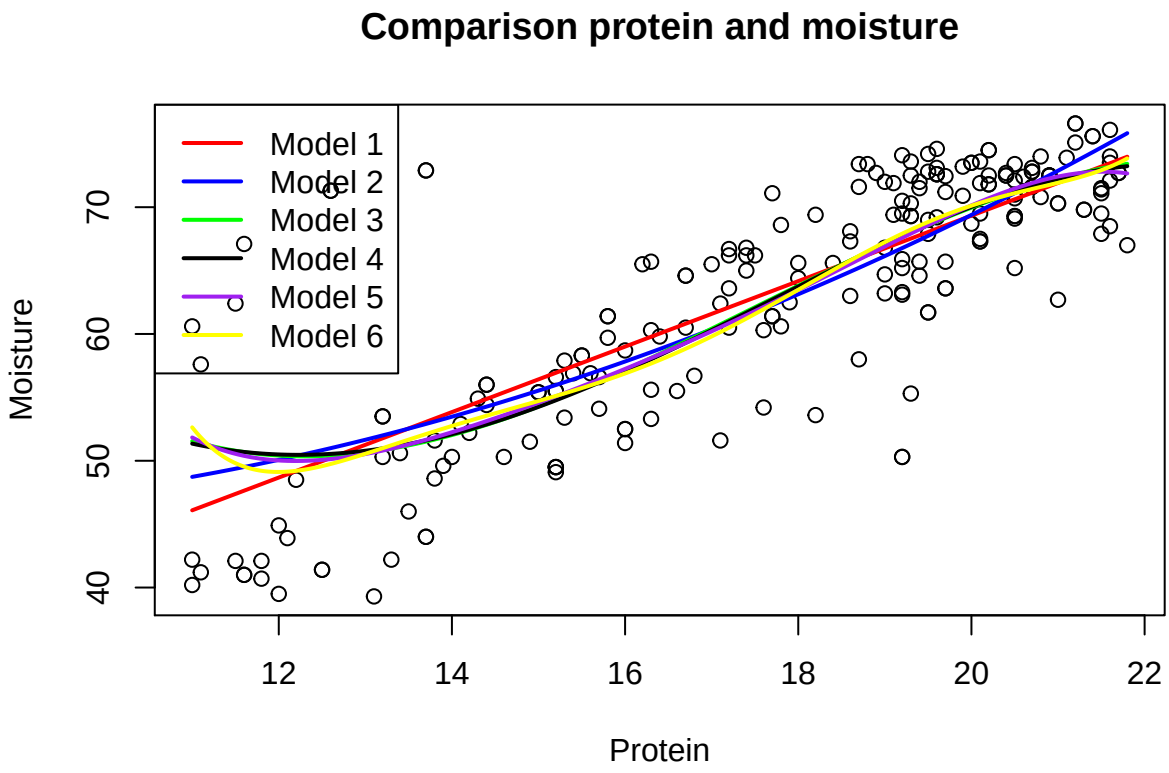
```
  lm_model = lm(y~., data = train)
  preds_train = predict(lm_model, train)
  preds_test = predict(lm_model, test)

  x_seq = seq(min(X_train, X_test), max(X_train,X_test), length = 1000)
  betas = lm_model$coefficients


  x_matrix = convert_to_poly(x_seq, degree, include_intercept = TRUE)



  lines(x=x_seq, y = t(as.matrix(betas))%*%t(x_matrix), col = colors[i], lwd = 2)
  err_train = MSE(y=y_train, y_hat = preds_train)
  err_test = MSE(y=y_test, y_hat=preds_test)
  return(c(err_train, err_test))
}
```

20.4.3 Task 3 - Compare polynomial models

In this task, we use the functions created previously to generate the models and get their MSEs for the training and the data set.

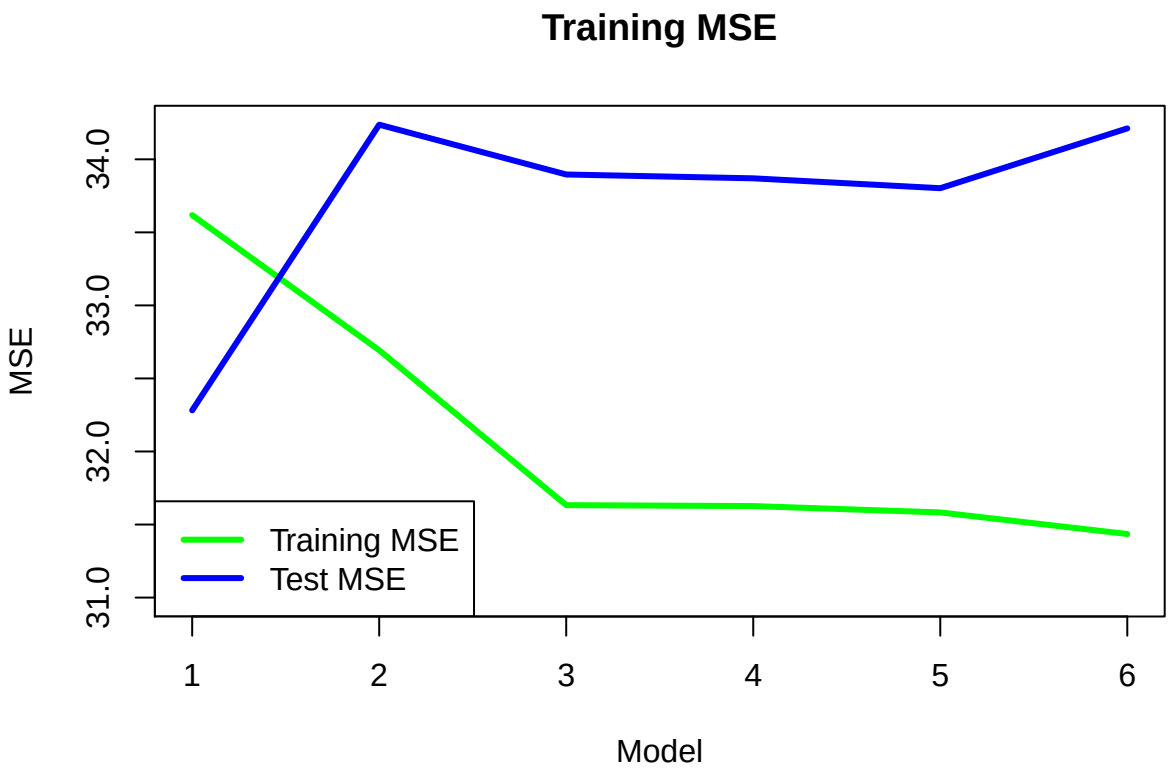
```
# Plot protein vs moisture (call before polynomial model)
plot(x=data$Protein, y = data$Moisture,
     xlab = "Protein",
     ylab = "Moisture",
     main = "Comparison protein and moisture")
# Task 3

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

X_train = train$Protein
y_train = train$Moisture
X_test = test$Protein
y_test = test$Moisture


# Initialize matrix of errors.
errors = matrix(NA, nrow=6, ncol = 2)
colnames(errors) = c("training MSE", "test MSE")
colors = c("red", "blue", "green", "black", "purple", "yellow")
for (i in 1:6) {
  # Calc train and test MSEs for each polynomial model up to degree 6
  errors[i,] = polynomial_model(X_train = X_train,
                                y_train = y_train,
                                X_test = X_test,
```

```
                                y_test = y_test,
                                degree=i,
                                col = colors[i])
}
rownames(errors) = seq(1,6,1)
colnames(errors) = c("Training MSE", "Test MSE")


# Create a legend
legend("topleft",
       lty = rep(1,6),
       col = colors,
       lwd = c(2,2,2,2,2,2),
       legend = c("Model 1",
                  "Model 2",
                  "Model 3",
                  "Model 4",
                  "Model 5",
                  "Model 6"))
```




```
plot(x=seq(1,6,1), y = errors[,1],
     col = "green",
     type = "l",
     xlab = "Model",
     ylab = "MSE",
     lwd = 3,
     main = "Training MSE", ylim = c(31, max(errors)))
lines(x=seq(1,6,1), y = errors[,2],
     col = "blue",
     lwd = 3,
```

```
     xlab = "Model",
     ylab = "MSE",
     main = "Test MSE")
legend("bottomleft",
       lty = rep(1,6),
       col = c("green", "blue"),
       lwd = c(3,3),
       legend = c("Training MSE",
                  "Test MSE"))
```



We clearly see that with a higher polynomial degree, we manage to fit the data better. However, the test set does not necessarily improve, but rather do not follow a clear pattern. Depending on which seed one sets, the MSE results varies, but it does not necessarily improve with higher polynomials. This is because the more polynomials we have, we risk having a greater bias towards the training data.

20.4.4 Task 4 - stepAIC

In task 4, we use variable selection using `stepAIC()`.

```
#Task 4 - use entire data set.

# Perform variable selection of a linear model
# in which fat is a response and Channel1-Channel100 are predictors.
# Comment on how many variables were selected.

library(MASS)
dataToUse = data
#Remove protein and Moisture and Sample. We only want the channels and Fat for this task.
dataToUse$Protein = NULL
```

```
dataToUse$Moisture = NULL
dataToUse$Sample = NULL
lm.fit = lm(Fat ~ ., data = dataToUse)

# Do the stepAIC. trace = 0 means hide iteration outputs
step = stepAIC(lm.fit, direction = "both", trace = 0)
# Print info about it.
print(summary(step))

nrVarsChosen = length(step$coefficients)
```

Printing the number of variables chosen, we get that 64 variables have been chosen. This is quite a high number, but we managed to at least remove $\frac{1}{3}$ of the variables using the variable selection.

```
print(nrVarsChosen) # Print nr vars chosen
```

```
## [1] 64
```

20.4.5 Task 5 - Ridge regression

Using the model retrieved through `stepAIC()`, we do a ridge regression.

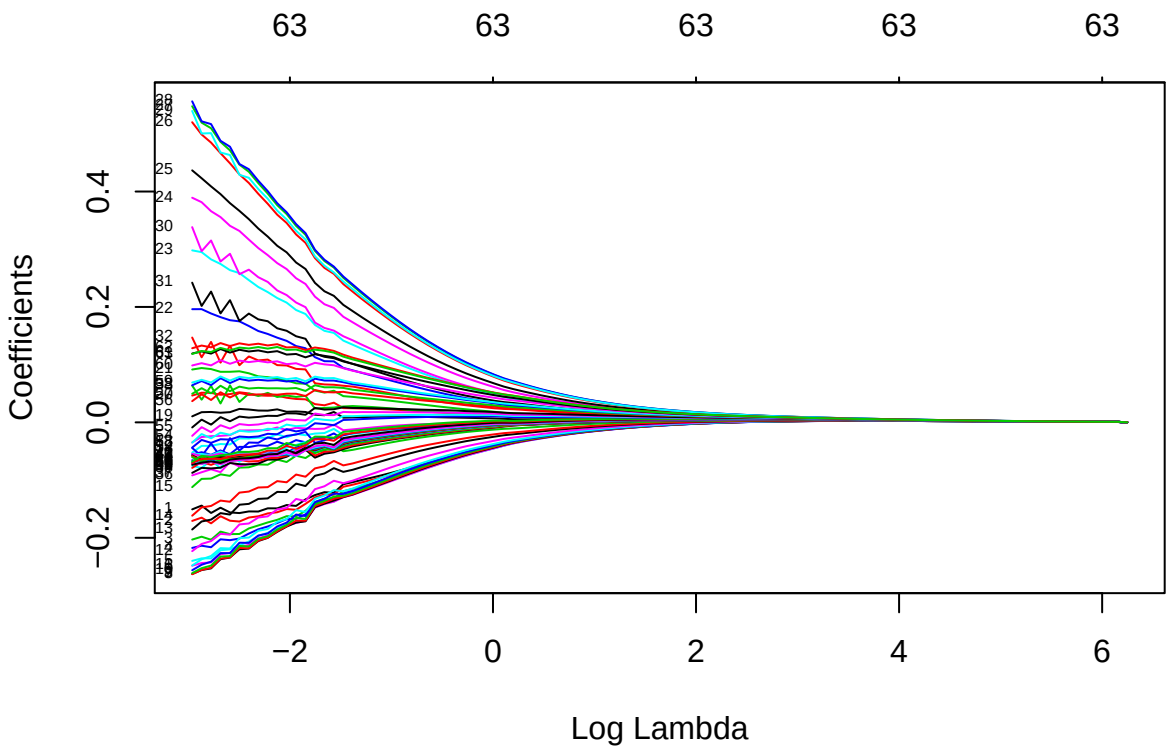
```
# Task 5 - ridge regression
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loading required package: foreach
```

```
## Loaded glmnet 2.0-16
```

```
variablesSelected = names(step$coefficients)
variablesSelected = variablesSelected[-c(1)]
dataSteps = dataToUse[,variablesSelected]
dataSteps = scale(dataSteps)
response = scale(dataToUse$Fat)
# Important - alpha = 0 for ridge regression! family sets the response type.
# For regression, gaussian.
# For classification, family = binomial.
ridgeRegModel = glmnet(as.matrix(dataSteps), response,
                       alpha = 0, family="gaussian")
# xvar="lambda" tells it to plot the lambda on the x axis.
# lambda is the penalizing coefficient
plot(ridgeRegModel, xvar="lambda", label=TRUE)
```

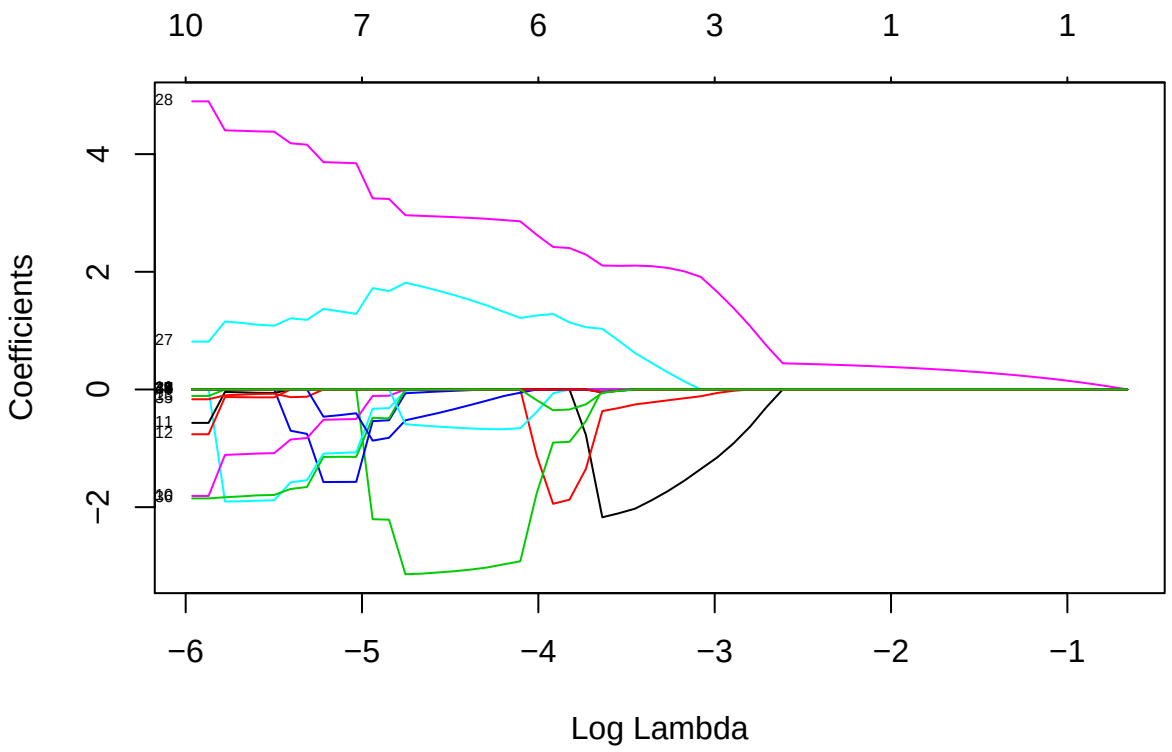


We can clearly that the coefficients are minimized, tending towards 0 the more we regularize. This is because the regularization makes them affect less, limiting their possibility to overfit.

20.4.6 Task 6 - LASSO

In task 6, we fit a LASSO regression instead of Ridge regression.

```
# Task 6 - Lasso regression
# Here, alpha = 1 instead! Important
lassoModel = glmnet(as.matrix(dataSteps), response,
                    alpha = 1, family="gaussian")
# xvar="lambda" tells it to plot the lambda on the x axis.
# lambda is the penalizing coefficient
plot(lassoModel, xvar="lambda", label=TRUE)
```



We see here as well that LASSO regression also makes the variables

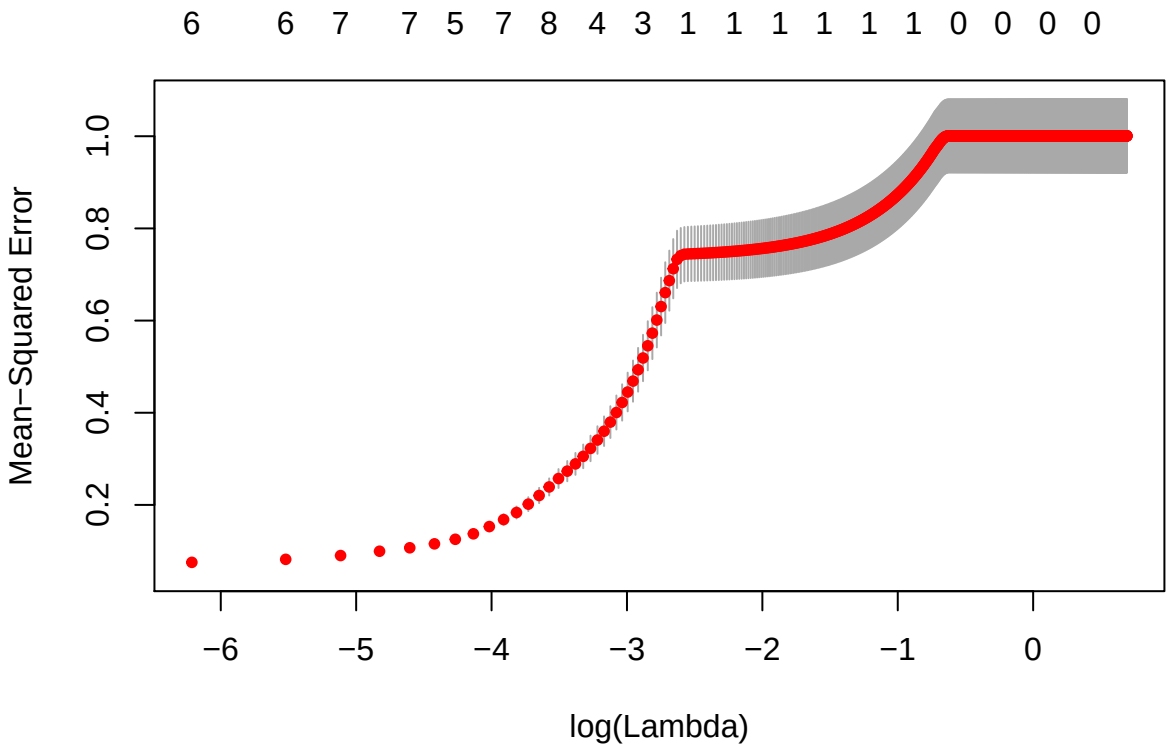
20.4.7 Task 7 - Lasso with cross-validation

Here, we use the LASSO model combined with cross-validation, to see whether we can reduce the number of variables.

```
# Task 7 - use cross-validation to find the optimal lasso model

lambda = seq(0,2, length = 1000)
# cv.glmnet does cross validation to choose the best model.
lassoCVModel = cv.glmnet(as.matrix(dataSteps), response, lambda = lambda, alpha = 1, family="gaussian")
lassoCVModel$lambda.min
```

```
## [1] 0
```

```
plot(lassoCVModel)
```




```
# Use the commented line below to print the coefficients of variables.
#coef(lassoCVModel, s = "lambda.min")
```

The LASSO model manages to reduce the variables to only 11, compared to 64 by `stepAIC`.

20.4.8 Task 8 - Compare stepAIC and LASSO

While we saw that `stepAIC` reduced the number of variables to 64, LASSO combined with cross-validation managed to reduce it to only 11, indicating that the LASSO with crossvalidation is better at reducing the number of features.

21 Lab 2

21.1 Assignment 1 - LDA and Logistic Regression

First, we initialize functions necessary for the task.

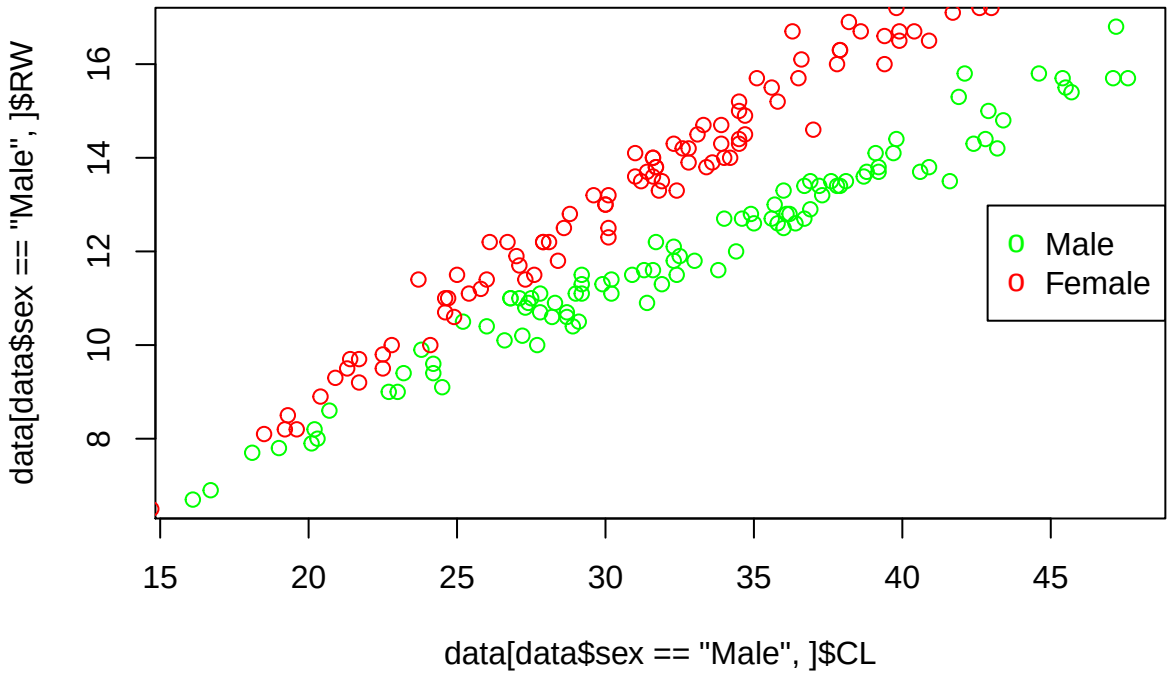
```
misclassificationRate = function(confMatrix) {
  return(1 - sum(diag(confMatrix))/sum(confMatrix))
}

log_reg = function(data, fitModel, pLimit = 0.5) {
  # Specify response in predict,
  fits = predict(fitModel, data)

  probabilities = fits

  classifications = apply(as.matrix(probabilities), 1, function(row) {
    if (row > pLimit) {
```

```
      return(1)
    } else {
      return(0)
    }
  })
  return(classifications)
}
```

21.1.1 Task 1 - are they linearly separable?

Plotting the sex of the crabs on the axis of `RW` and `CL` shows a clear possibility of separating the two sexes through linear classification.

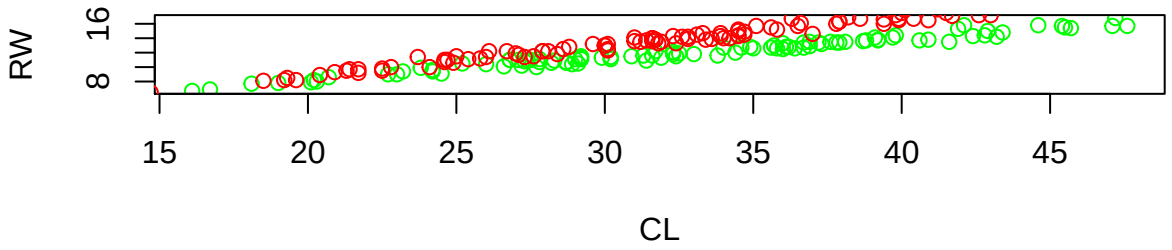
```
# Task 1

data = read.csv(
  "~/Desktop/Plugg_Lkpg/HT18/TDDE01/Tentaplugg/Lab_repetitions/Lab2/australian-crabs.csv")

# Should I use training and test? Probably yes
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]
test=data[-id,]
plot(data[data$sex == "Male",]$CL,
     data[data$sex == "Male",]$RW, col = "green")
points(data[data$sex == "Female",]$CL,
       data[data$sex == "Female",]$RW, col = "red")
legend("right",
       pch = c("o","o"),
       col = c("green", "red"),
       legend = c("Male","Female"))
```

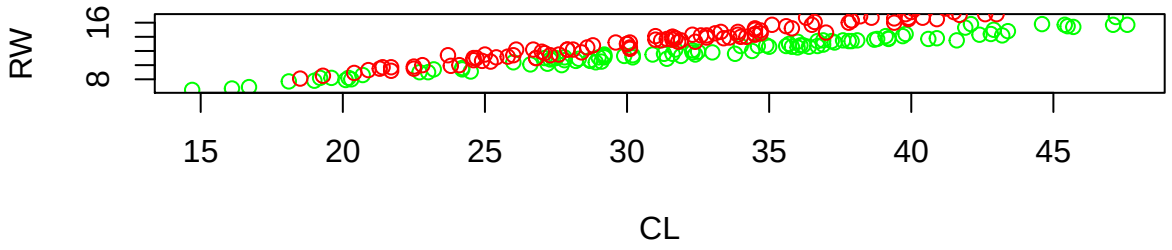


21.1.2 Task 2 - LDA

Performing a Linear Discriminant Analysis on the data through the function `lda()`, using the variables RW and CL yields an almost perfect separation of the data, which confirms what the graph indicated. The misclassification rate is only 0.035, an almost perfect score.

```
# Task 2

library(MASS)

fitModel = lda(formula = sex ~ RW + CL,
               data = data,
               prior = c(length(data$sex[data$sex == "Male"])
                         /nrow(data),length(data$sex[data$sex == "Female"])
                         /nrow(data)))

fits = predict(fitModel, data)
par(mfrow=c(2,1))
plot(data[data$sex == "Male",]$CL,
     data[data$sex == "Male",]$RW,
     col = "green", ylab = "RW", xlab = "CL")
points(data[data$sex == "Female",]$CL,
       data[data$sex == "Female",]$RW, col = "red")
plot(data[fits$class == "Male",]$CL,
     data[fits$class == "Male",]$RW,
     col = "green", ylab = "RW", xlab = "CL")
points(data[fits$class == "Female",]$CL,
       data[fits$class == "Female",]$RW,
       col = "red")
```



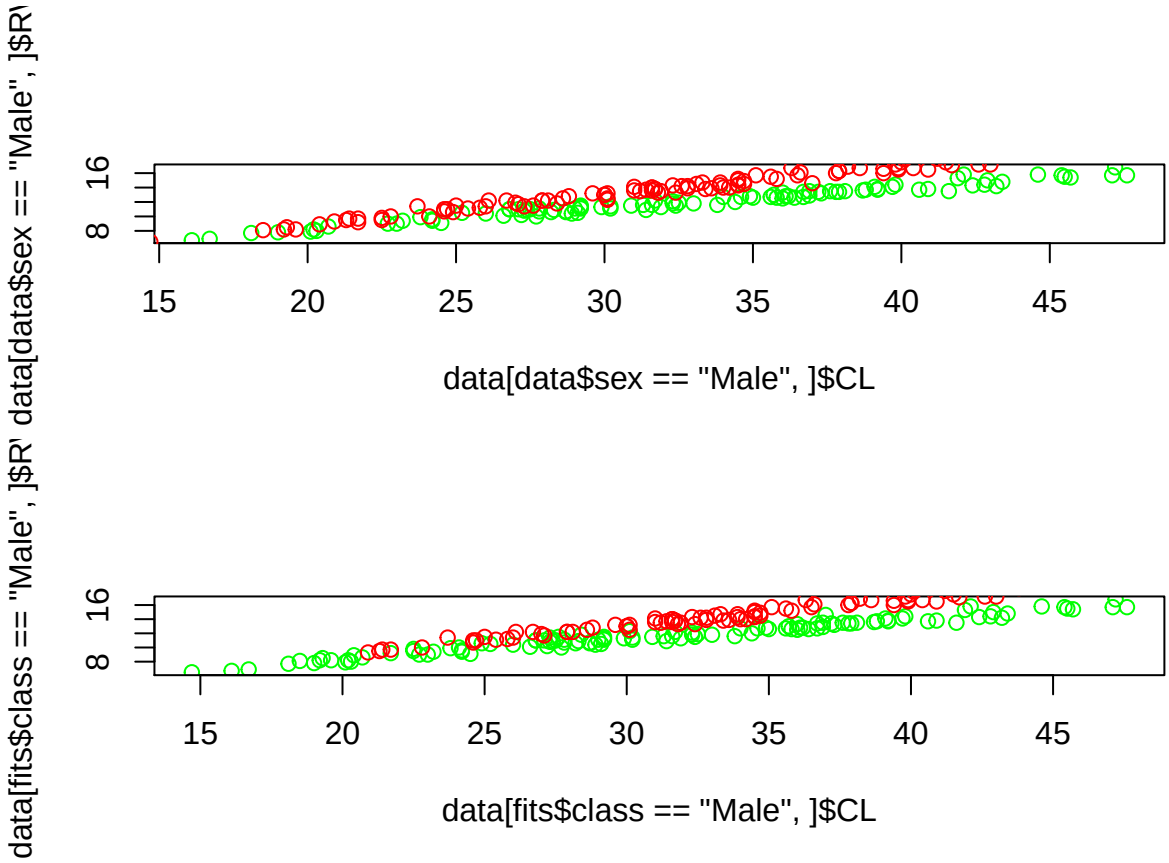



```
par(mfrow=c(1,1))

confMatrix = table(data$sex, fits$class)
#confMatrix
#print("misclassificionation rate")
#misclassificationRate(confMatrix)
```

21.1.3 Task 3 - change LDA priors for skewed distribution

In task 3, we changed the priors from $P(sex = Male) = 0.5$ and $P(sex = Female) = 0.5$ to $P(sex = Male) = 0.9$ and $P(sex = Female) = 0.1$. Although the result worsened to a misclassification rate to only 8 %, the linearly separable property of the data still yielded a high accuracy of 92 %. As can be seen in the confusion matrix, this skews some predictions of females to be predicted as males, as the priors skews the data towards this.

```
# Task 3

fitModel = lda(formula = sex ~ RW + CL, data = data, prior = c(0.1, 0.9))
fits = predict(fitModel, data)
par(mfrow=c(2,1))
plot(data[data$sex == "Male",]$CL,
     data[data$sex == "Male",]$RW, col = "green")
points(data[data$sex == "Female",]$CL,
       data[data$sex == "Female",]$RW, col = "red")
plot(data[fits$class == "Male",]$CL,
     data[fits$class == "Male",]$RW, col = "green")
points(data[fits$class == "Female",]$CL,
       data[fits$class == "Female",]$RW, col = "red")
```




```
par(mfrow=c(1,1))
confMatrix = table(data$sex, fits$class)
confMatrix
```

```
##
##          Female Male
##   Female     84   16
##   Male        0  100
```

```
print("misclassificionation rate")
```

```
## [1] "misclassificionation rate"
```

```
misclassificationRate(confMatrix)
```

```
## [1] 0.08
```

21.1.4 Task 4 - Logistic Regression with separation line for p = 0.5

In task 4, we use logistic regression to fit the model. The results are here very good as well, only one less classified incorrectly compared to LDA. The plot shows the line separating the sexes, which appears separare the vast majority of the data points correctly.

Retrieving the separating line from the logistic is trivial. Consider the logistic regression:

$$p(Y = 1|w, x) = \frac{1}{1 + e^{w^T x}} \text{ where } w^T = \beta_0 + \beta_{RW} x_{RW} + \beta_{CL} x_{CL}$$

Then, with a classification threshold of 0.5, we get that

$$0.5 = \frac{1}{1+e^{w^T x}} \Leftrightarrow 0.5+0.5e^{w^T x} = 1 \Leftrightarrow e^{w^T x} = 1 \Leftrightarrow w^T x = log(1) = 0 \Leftrightarrow w^T x = \beta_0 + \beta_{RW} x_{RW} + \beta_{CL} x_{CL} \Leftrightarrow$$

$$\Leftrightarrow -\frac{\beta_0 + \beta_{CL} x_{CL}}{\beta_{RW}} = x_{RW}$$

Which is the line we are looking for in this plot.

```
# Task 4

fit = glm(sex ~ RW + CL, data = data, family = "binomial")
```

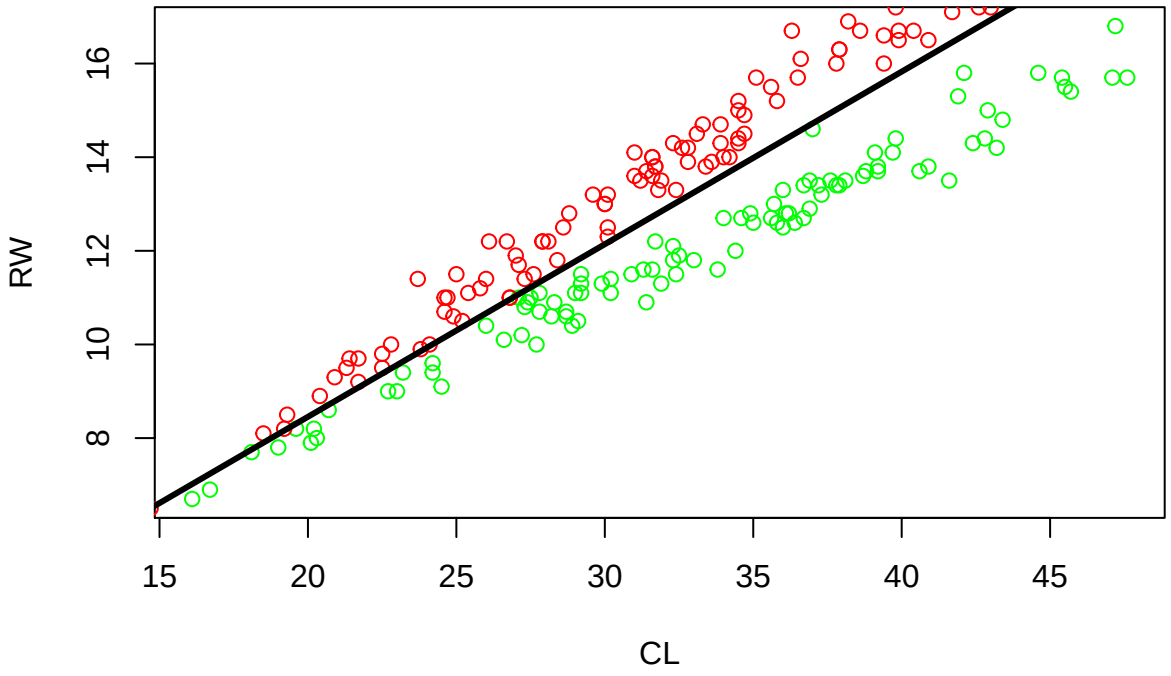
```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
#summary(fit)
fits = predict(fit, data)


probabilities = fits
pLimit = 0.5
classifications = apply(as.matrix(probabilities), 1, function(row) {
  if (row > pLimit) {
    return(1)
  } else {
    return(0)
  }
})
#classificationsTest = log_reg(test, fit)


#table(data$sex, classifications)
#print("misclassification rate")
#misclassificationRate(table(data$sex, classifications))

fit$coefficients
```

```
## (Intercept)          RW          CL
##   13.616628  -12.563893    4.630747
```

```
# Report the equation of the decision boundary and
# draw it in the plot of the classified data
plot(data[classifications == 1,]$CL,
     data[classifications == 1,]$RW,
     col = "green", xlab = "CL", ylab ="RW")
points(data[classifications == 0,]$CL,
       data[classifications == 0,]$RW, col = "red")
# Equation of decision boundary:
# $0.5 = \frac{1}{1 + exp(-(intercept + beta1*RW + beta2*CL))}$
CLseq = seq(min(data$CL), max(data$CL), length = 1000)
Xseq = matrix(c(rep(1, length = 1000),CLseq), ncol = 2, nrow = 1000)

beta_RW = fit$coefficients[2]
beta_0 = fit$coefficients[1]
beta_CL = fit$coefficients[3]

RWseq = -t(as.matrix(c(beta_0,
```

```
                        beta_CL)))%*%t(Xseq)/beta_RW
lines(x = CLseq, y = RWseq, lwd = 3)
```



21.2 Assignment 2 - Credit scoring with tree and naive bayes

In assignment 2, our task was to use the data from the file `creditscoring.xls` to compare decision trees and Naïve in their abiity to predict whether a customer will be able to pay back the loan or not.

21.2.1 Task 1 and 2 - compare Gini and Deviance trees

Task 1 was to load and partition the data, a trivial task to perform given the lectures. Then, in task 2, we fit a decisision tree in two ways; one by using the Deviance measure, another by using the Gini measure.

```
misclassificationRate = function(confMatrix) {
  return(1 - sum(diag(confMatrix))/sum(confMatrix))
}
library(readxl)
data =
  read_excel(
    "~/Desktop/Plugg_Lkpg/HT18/TDDE01/Tentaplugg/Lab_repetitions/Lab2/creditscoring.xls")
data$good_bad = factor(data$good_bad)
# Task 1

n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]
id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.25))
valid=data[id2,]
```

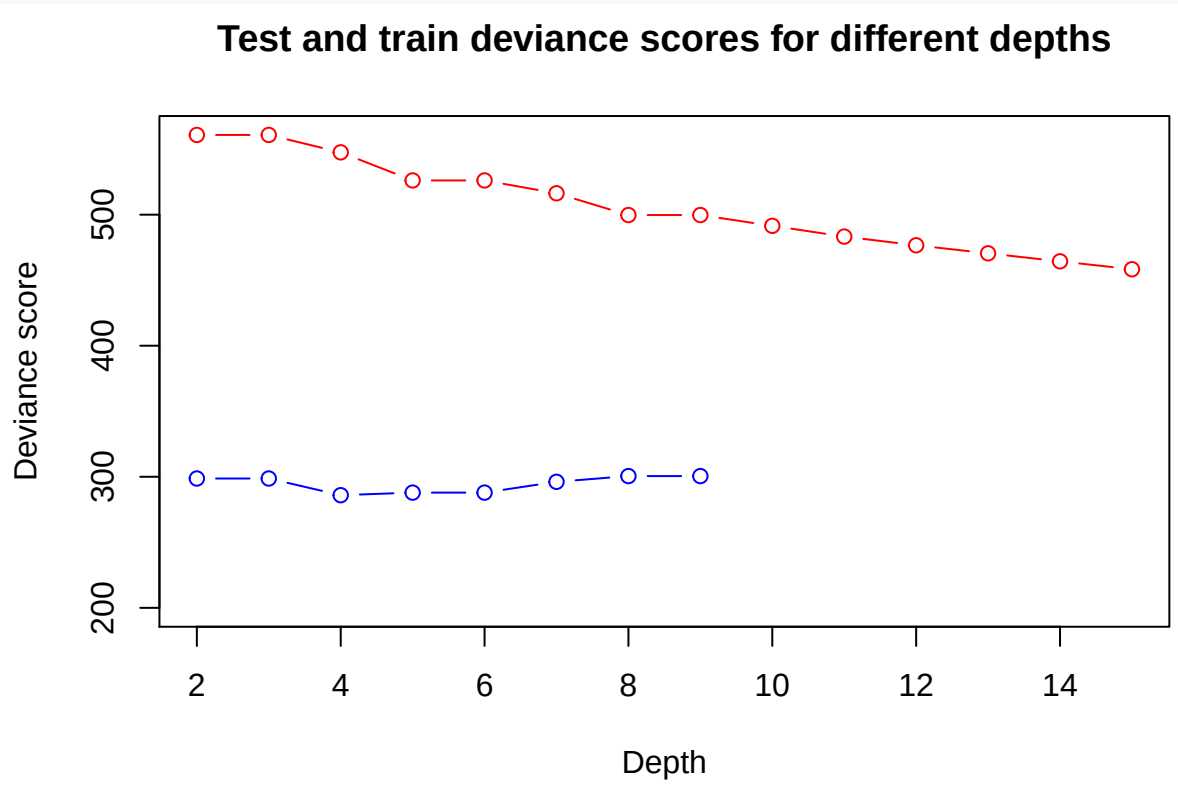
```
id3=setdiff(id1,id2)
test=data[id3,]

library(tree)

# Fitting with gini

fitGini = tree(good_bad ~., data = train, split = c("gini"))

trainClassifGini = factor(apply(predict(fitGini, train), 1, function(row) {
  return(names(which.max(row)))
}))
testClassifGini = factor(apply(predict(fitGini, test), 1, function(row) {
  return(names(which.max(row)))
}))
# Uncomment lines below to get analysis. deviance is better.
#table(train$good_bad, trainClassifGini)
#print("misclassification rate train gini")
#misclassificationRate(table(train$good_bad, trainClassifGini))

#table(test$good_bad, testClassifGini)
#print("misclassification rate test gini")
#misclassificationRate(table(test$good_bad, testClassifGini))


# fitting with deviance
fitDeviance = tree(good_bad ~., data = train, split = c("deviance"))

trainClassifDeviance = factor(apply(predict(fitDeviance, train), 1, function(row) {
  return(names(which.max(row)))
}))
testClassifDeviance = factor(apply(predict(fitDeviance, test), 1, function(row) {
  return(names(which.max(row)))
}))

# Uncomment lines below to get analysis. deviance is better.
#table(train$good_bad, trainClassifDeviance)
#print("misclassification rate train deviance")
#misclassificationRate(table(train$good_bad, trainClassifDeviance))
#table(test$good_bad, testClassifDeviance)
#print("misclassification rate test deviance")
#misclassificationRate(table(test$good_bad, testClassifDeviance))
#Deviance is best => choose deviance!
```

We see that the deviance measure yields, based on the missclassification rate, significantly better results. Due to this, we shall use this in the upcoming tasks.

21.2.2 Task 3 - Prune the tree

Deciding the optimal tree depth, we can that it is 4. The optimal tree is printed below. We can also see that the misclassification rate is slightly improved, although not significantly.

```r
# Task 3

fit = fitDeviance
trainScore=rep(0,9)
testScore=rep(0,9)

# Only do from depth 2 and forward. Possible up to depth 15
for(i in 2:15) {
  prunedTree=prune.tree(fit,best=i)
  pred=predict(prunedTree, newdata=valid,
               type="tree")
  trainScore[i]=deviance(prunedTree)
  testScore[i]=deviance(pred)
}
plot(2:15, trainScore[2:15],
     type="b", col="red",
     ylim=c(200,max(trainScore, testScore)),
     xlab = "Depth",
     ylab = "Deviance score",
     main ="Test and train deviance scores for different depths")
points(2:9, testScore[2:9], type="b", col="blue")
```




```r
depth = which.min(testScore[2:9]) + 1
print(paste("Best depth is", depth))
```

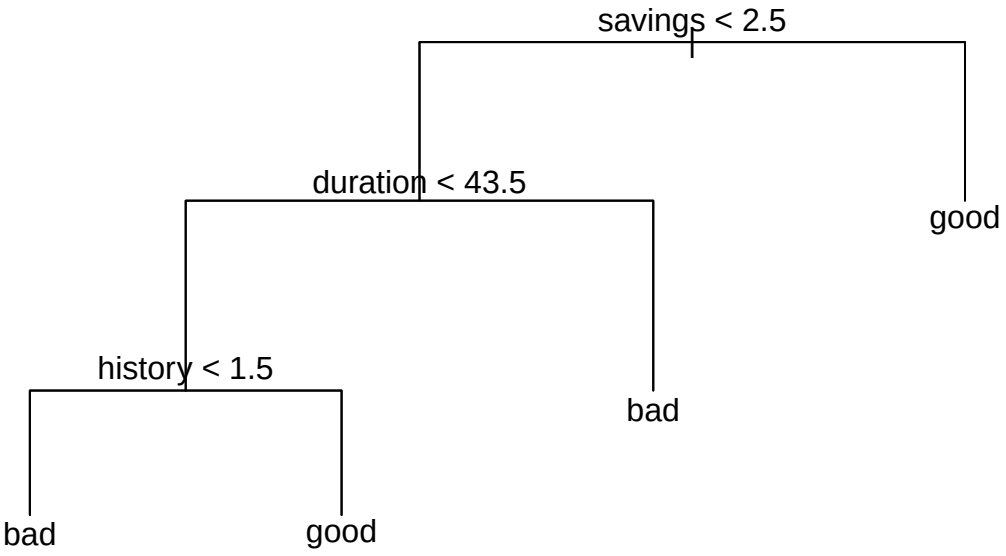
```
## [1] "Best depth is 4"
```

```r
finalTree=prune.tree(fit, best=depth)
Yfit=predict(finalTree, newdata=valid,
             type="class")
```

```
# Uncomment lines below to get analysis.
#table(valid$good_bad,Yfit)
#misclassificationRate(table(valid$good_bad,Yfit))

# Plotting the tree
print(finalTree)
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
## 1) root 494 598.00 good ( 0.2935 0.7065 )
##   2) savings < 2.5 346 446.70 good ( 0.3468 0.6532 )
##     4) duration < 43.5 325 405.90 good ( 0.3169 0.6831 )
##       8) history < 1.5 22  27.52 bad ( 0.6818 0.3182 ) *
##       9) history > 1.5 303 365.10 good ( 0.2904 0.7096 ) *
##     5) duration > 43.5 21  20.45 bad ( 0.8095 0.1905 ) *
##   3) savings > 2.5 148 134.40 good ( 0.1689 0.8311 ) *
```

```
plot(finalTree)
text(finalTree, pretty = 0)
```



21.2.3 Task 4 - Apply Naive Bayes

Using the training data to classify using Naïve Bayes, we can see that it is not as efficient as the tree using Deviance as measure of impurity.

```
# Task 4 - use naive bayes
library(MASS)

# Use package e1071.
library(e1071)
fit = naiveBayes(good_bad ~.,data=train)
classTrain = predict(fit, train)
classTest = predict(fit, test)
```

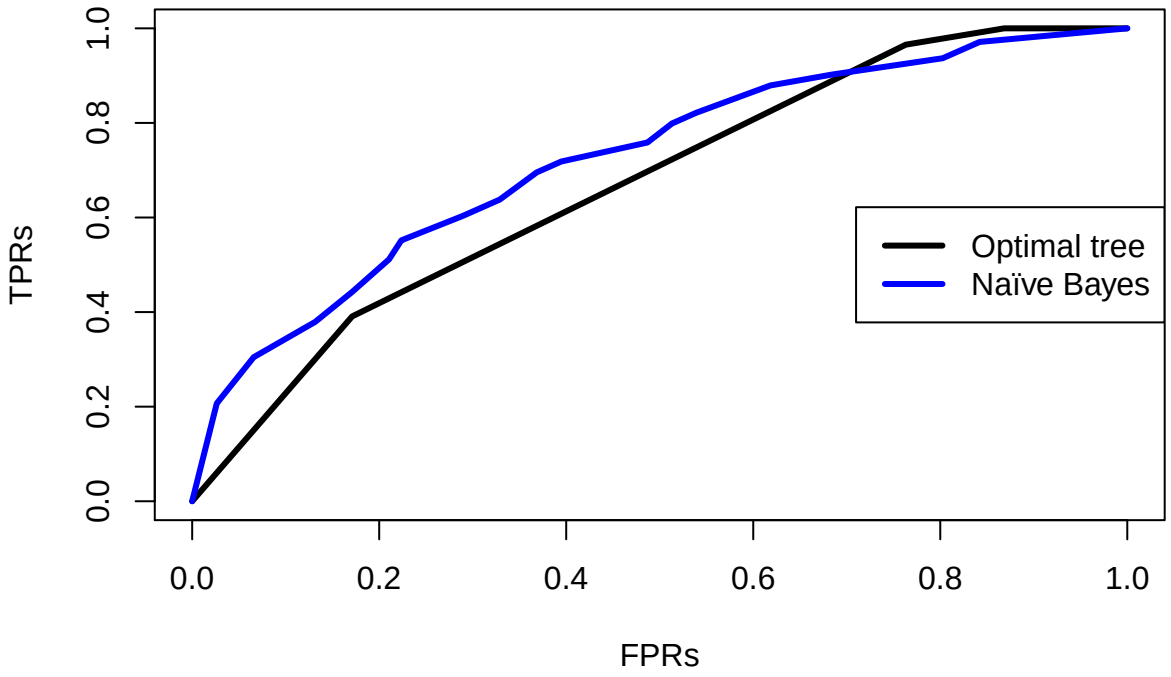
```
# Uncomment below for analysis
#table(train$good_bad, classTrain)
#misclassificationRate(table(train$good_bad, classTrain))


#table(test$good_bad, classTest)
#misclassificationRate(table(test$good_bad, classTest))
```

21.2.4 Task 5 - Compute ROC curves for Naive Bayes and optimal tree

Computing the ROC curves for the two different models, we can see in the plot that Naïve Bayes seem to perform slightly better. This is also confirmed by computing the AUC.

```
# Task 5 use optimal tree and naive bayes model to classify test data by using Y

Pi = as.matrix(seq(0,1, by = 0.05))

predictionsTest = predict(finalTree, test)
predictionsTestBayes = predict(fit, test, "raw")

testClassifs = matrix(apply(as.matrix(Pi), 1, function(pi_val, preds) {
  return(ifelse(preds > pi_val, "good","bad"))
}, predictionsTest[,2]), nrow = nrow(predictionsTest),ncol = length(Pi))
testClassifsBayes = matrix(apply(as.matrix(Pi), 1, function(pi_val, preds) {
  return(ifelse(preds > pi_val, "good","bad"))
}, predictionsTestBayes[,2]), nrow = nrow(predictionsTestBayes),ncol = length(Pi))

# TPR corrected version.
# Send in predictions and labels as FACTORS.
# Predictions can also be sent in as characters or
# whatever the levels values are of predictions
TPR = function(predictions, labels) {
  y = labels
  y_hat = predictions
  conf_m = table(Actual = y, Predicted = y_hat)

  # True positive rate - true positives divided by all actually being positive
  if (dim(conf_m)[2] == 1 && levels(labels)[2] %in% y_hat) {
    return(1) # If we only have good,
    #all are classified as positive correctly
    #(although many are predicted incorrectly) and this is thus 1
  } else if (dim(conf_m)[2] == 1 && levels(labels)[1] %in% y_hat) {
    return(0)
  } else {
    TP = conf_m[2,2] # True positives
    N_plus = sum(conf_m[2,])
    return(TP/N_plus)
  }

}

# FPR corrected version
# Send in predictions and labels as FACTORS
```

```r
FPR = function(predictions, labels) {
  y = labels
  y_hat = predictions
  conf_m = table(Actual = y, Predicted = y_hat)
  # False positive rate - false positives divided by all actually being negative
  if (dim(conf_m)[2] == 1 && levels(labels)[2] %in% y_hat) {
    # If we only have "good", it means all are classified as positive,
    # and the false positives is thus as many as the number of false ones.
    #Thus, the FP/N_minus = 1
    return(1)
  } else if (dim(conf_m)[2] == 1 && levels(labels)[1] %in% y_hat) {
    return(0) # If we only have bad,
    #we do not have a single false positive as there are no positives. Hence, 0
  } else {
    FP = conf_m[1,2]
    N_minus = sum(conf_m[1,])
    return(FP/N_minus)
  }

}
TPRs = apply(testClassifs,2, TPR, test$good_bad)
FPRs = apply(testClassifs,2, FPR, test$good_bad)
plot(x=FPRs, y = TPRs,
     type = "l", xlim = c(0,1), ylim = c(0,1),
     lwd = 3, main = "ROC curves")

TPRsBayes = apply(testClassifsBayes,2,TPR, test$good_bad)
FPRsBayes = apply(testClassifsBayes,2,FPR, test$good_bad)
lines(x=FPRsBayes, y = TPRsBayes, col = "blue", lwd = 3)
legend("right",
       lty = rep(1,2),
       col = c("black", "blue"),
       lwd = c(3,3),
       legend = c("Optimal tree","Naïve Bayes"))
```




```
# AUC function. Calculates the area under the curve
AUC = function(TPR, FPR) {
  # TPR is y, FPR is x
  # Order after FPR
  xInd = order(FPR)
  x = FPR[xInd]
  y = TPR[xInd]
  area = 0
  for (i in 2:length(TPR)) {
    area = (x[i]-x[i-1])*(y[i] + y[i-1])/2 + area
  }
  return(area)
}

print("AUC bayes")
```

```
## [1] "AUC bayes"
```

```
AUC(TPRsBayes, FPRsBayes)
```

```
## [1] 0.7225499
```

```
print("AUC tree")
```

```
## [1] "AUC tree"
```

```
AUC(TPRs, FPRs)
```

```
## [1] 0.669994
```

```
# Check which probability threshold is best to choose.
print("best classification probability threshold tree")
```

```
## [1] "best classification probability threshold tree"
```

```
print(Pi[which.max(TPRs - FPRs)])
```

```
## [1] 0.75
```

```
print("best classification probability threshold tree")
```

```
## [1] "best classification probability threshold tree"
```

```
print(Pi[which.max(TPRsBayes - FPRsBayes)])
```

```
## [1] 0.7
```

21.2.5 Task 6 - incorporate loss function in Naive Bayes

In task 6, the task was use a loss function and then use Naïve Bayes to decide. If one calculates on paper the decision rule, one arrives at

$$y_i = \begin{cases} good & \text{if } p(y_i = bad|x) < 10p(y_i = good|x) \\ bad & \text{otherwise} \end{cases}$$

Since we want to avoid classifying bad ones as good incorrectly. This yields the result below.

```
# Task 6

# use loss matrix to decide.

# USE NAIVE BAYES - which was
predictionsTrain = predict(fit, train, "raw")
predictionsTest = predict(fit, test, "raw")


trainClassifsWLoss = apply(predictionsTrain, 1, function(row) {
  # Important: 10 is for false positives, 1 is for false negatives
  # Out of this, if a task comes you can deduct correct use here.
  # row[1] is probability for bad (negative),
  # row[2] is probability for good (positive)
  # losses = c(row[2], 10*row[1]) should be equal to line below
  losses = c(1 - row[1], 10*(1 - row[2]))
  return(which.min(losses))
})

testClassifsWLoss = apply(predictionsTest, 1, function(row) {
  losses = c(1 - row[1], 10*(1 - row[2]))
  # c(bad, good)
  return(which.min(losses))
})

# Uncomment lines below to obtain good results.

#print("Training conf matrix rom task 4")
#table(train$good_bad, classTrain)
#misclassificationRate(table(train$good_bad, classTrain))

#print("Test conf matrix rom task 4")
```

```
#table(test$good_bad, classTest)
#misclassificationRate(table(test$good_bad, classTest))

#table(train$good_bad, trainClassifsWLoss)
#misclassificationRate(table(train$good_bad, trainClassifsWLoss))
#table(test$good_bad, testClassifsWLoss)
#misclassificationRate(table(test$good_bad, testClassifsWLoss))
```

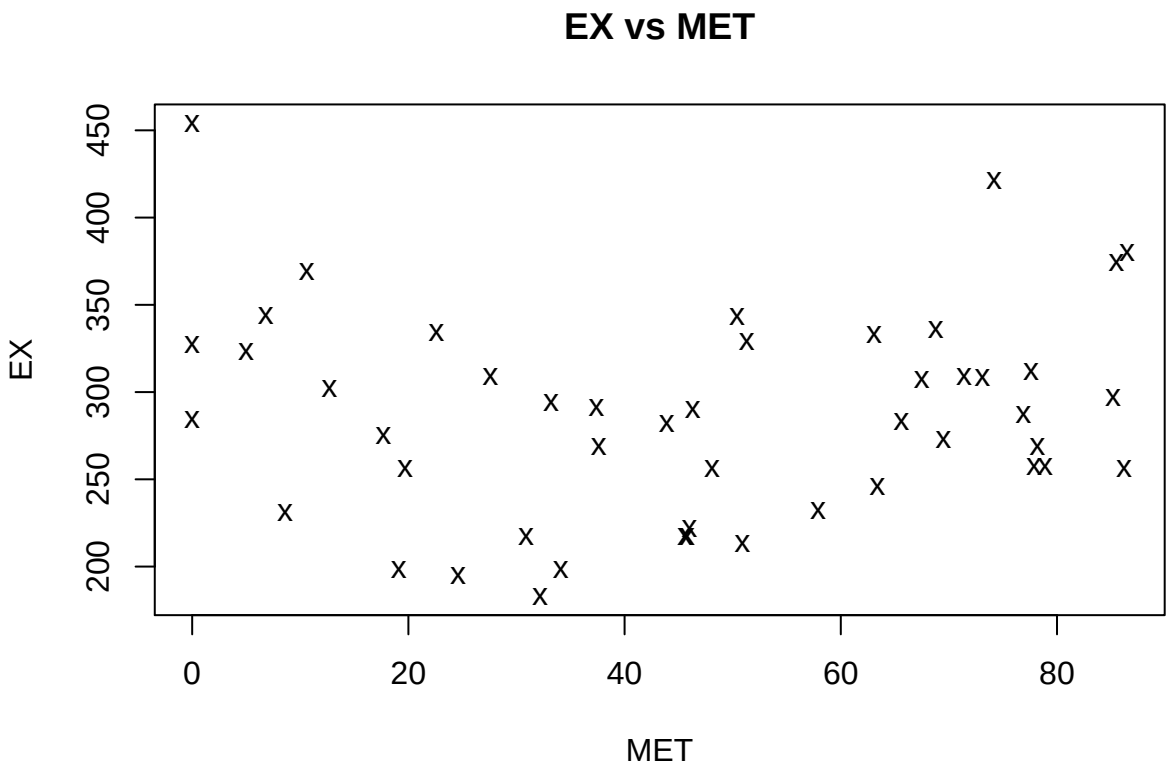
We see that we obtain a significantly worse result, with few bad classified as good, but a lot of good predicted as bad, since the loss function tends towards this.

21.3 Assignment 3 - Uncertainty estimation using Bootstrap for a regression tree

21.3.1 Task 1 - reorder data and plot

In task 1 - we simply reorder the data and plot it. It looks like it could be fitted with a polynomial model.

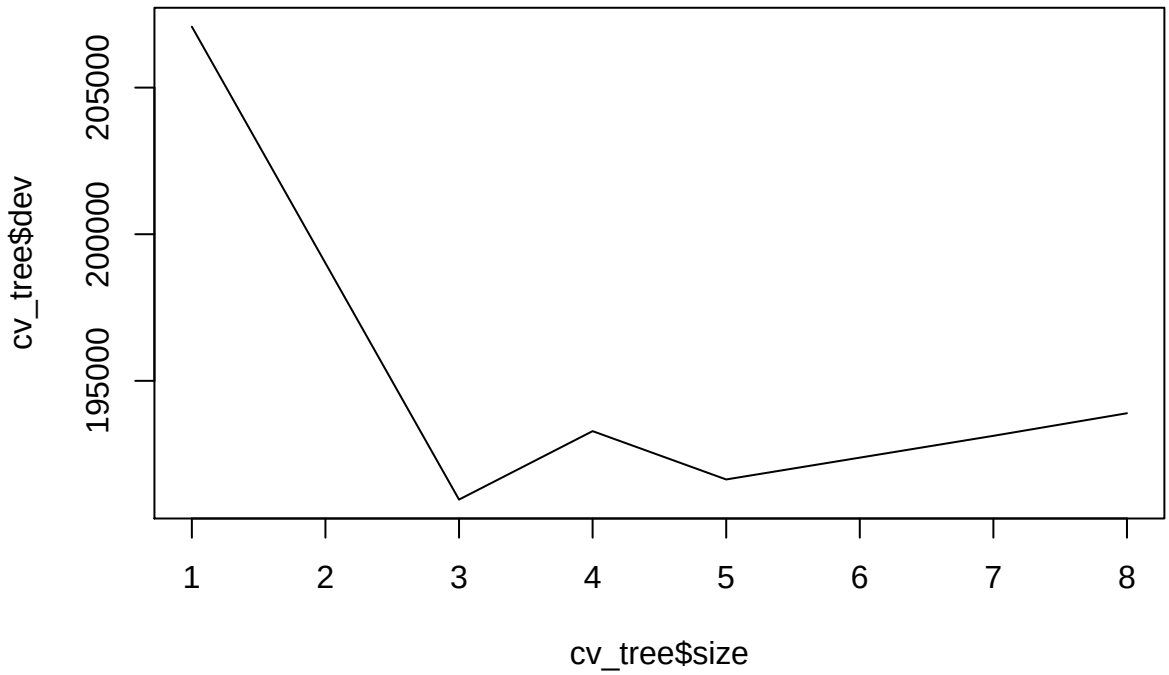
```
## Task 1

data =
  read.csv(
    "~/Desktop/Plugg_Lkpg/HT18/TDDE01/Tentaplugg/Lab_repetitions/Lab2/State.csv",
    header = T,
    sep = ";",
    dec = ",")

data = data[order(data$MET),]

plot(x=data$MET, y = data$EX, main ="EX vs MET", xlab = "MET", ylab = "EX", pch="x")
```



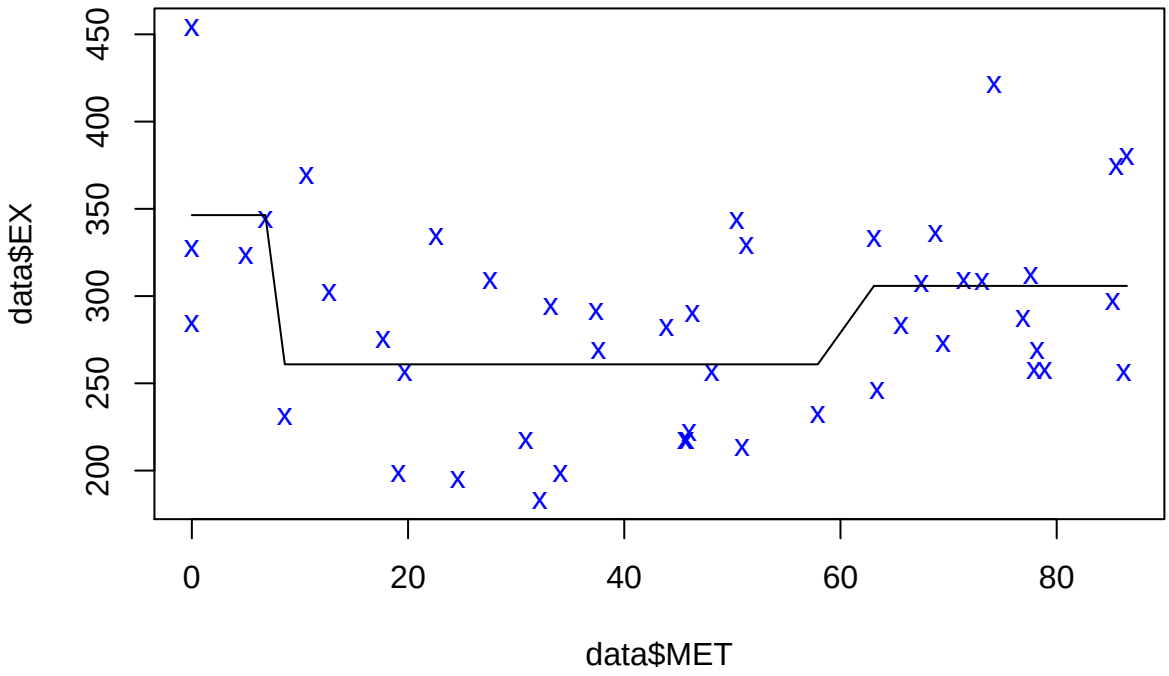
21.3.2 Task 2 - fitting a regression tree

Many of the residuals seem to be around -50, and it is mainly negative.

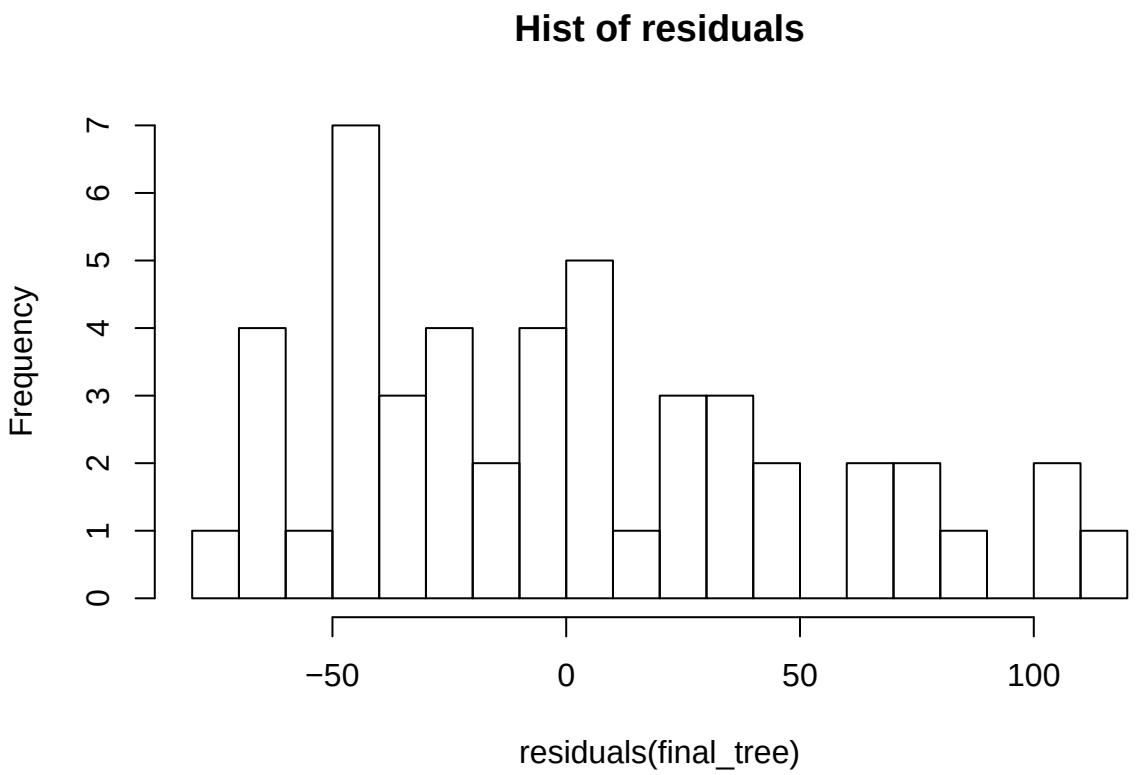
```
## Task 2

library(tree)
set.seed(12345)
tree_model = tree(EX~MET,data=data, control = tree.control(nobs = nrow(data),minsize = 8))
cv_tree = cv.tree(tree_model)
best_size = cv_tree$size[which.min(cv_tree$dev)]

plot(x=cv_tree$size, y= cv_tree$dev, type = "l")
```




```
final_tree = prune.tree(tree_model, best=best_size)

preds = predict(final_tree, data)



plot(x=data$MET, y = data$EX, col = "blue", pch = "x")
lines(x=data$MET, y = preds)
```



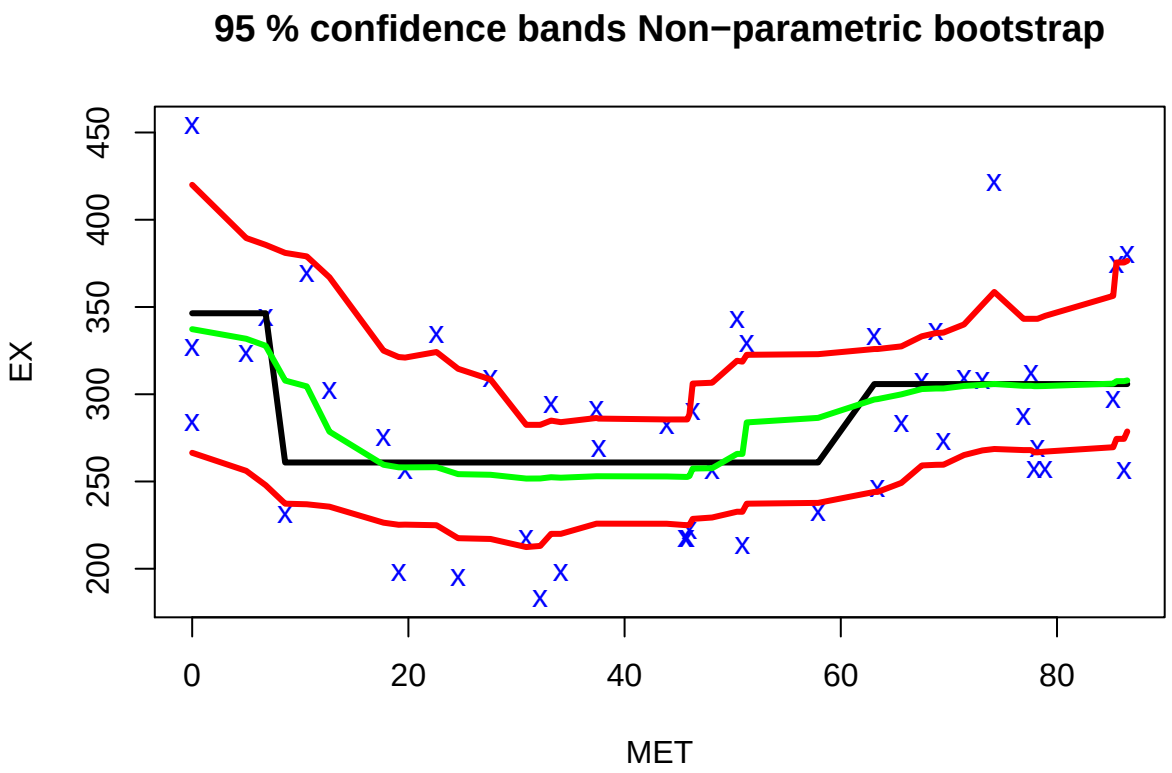
```r
hist(residuals(final_tree), breaks = 20, main = "Hist of residuals")
```



\subsubsection{Task 3 - Non-parametric bootstrap, computing and plotting 95 % confidence bands}

```r
# Compute and plot the 95 % confidence bands
# using non-parametric bootstrap for the tree regression model.
# Do non-parametric bootstrap
set.seed(12345)
B = 1000
all_preds = matrix(NA, nrow=nrow(data),ncol=B)
for (i in 1:B) {
  # Sample indices to pick out
  samples = sample(seq(1,nrow(data),1),size = nrow(data), replace=TRUE)
  data_sampled = data[samples,]
  tree_model = tree(EX~MET,
                    data=data_sampled,
                    control = tree.control(nobs = nrow(data),
                                           minsize = 8)
                    )
  final_tree = prune.tree(tree_model, best=best_size)
  all_preds[,i] = predict(final_tree, data)
}
confidence_bands = apply(all_preds,1,function(row) {
  return(quantile(row, probs = c(0.025,0.5,0.975)))
})
plot(x=data$MET, y = data$EX,
     col = "blue",
     pch = "x",
     xlab = "MET", ylab = "EX",
     main ="95 % confidence bands Non-parametric bootstrap")
```

```
lines(x=data$MET, y = preds, lwd=3)
lines(x=data$MET, y = confidence_bands[1,], lwd=3, col = "red")
lines(x=data$MET, y = confidence_bands[2,], lwd=3, col = "green")
lines(x=data$MET, y = confidence_bands[3,], lwd=3, col = "red")
```



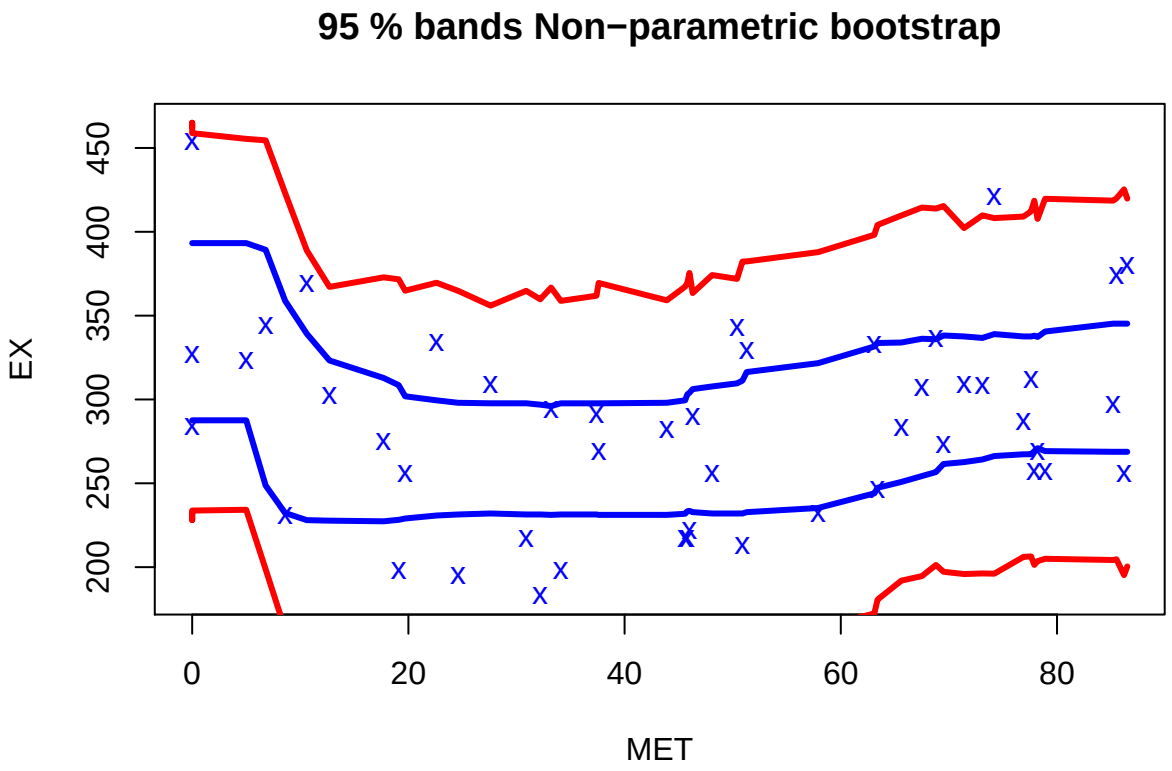

```
# Use parametric bootstrap
library(boot)
# Parametric bootstrap: Fit model to D, use this model all the time.

# Step 1: Fit model to D
tree_model = tree(EX~MET,
                  data=data,
                  control = tree.control(nobs = nrow(data),
                                         minsize = 8)
)

# Function to randomly generate the data points for non-param
random_gen = function(data, tree_model) {

  y_hat = predict(tree_model, data)
  y = data$EX
  residuals = y - y_hat
  sigma = sd(residuals)
  data$EX = rnorm(nrow(data), mean = y_hat, sd = sigma)
  return(data)
}

ci_tree = function(data) {
  tree_model = tree(EX~MET,
```

```
                    data=data,
                    control = tree.control(nobs = nrow(data),
                                           minsize = 8)
  )

  final_tree = prune.tree(tree_model, best=best_size)
  preds = predict(final_tree, data)

  return(preds)
}

pred_band_tree = function(data) {
  tree_model = tree(EX~MET,
                    data=data,
                    control = tree.control(nobs = nrow(data),
                                           minsize = 8)
  )

  final_tree = prune.tree(tree_model, best=best_size)
  preds = predict(final_tree,data)
  y = orig_data$EX
  residuals = y - predict(final_orig_tree, orig_data)
  preds = rnorm(nrow(data), mean = preds, sd = sd(residuals))
  return(preds)
}

final_orig_tree = prune.tree(tree_model, best=best_size)

#sigma2 = sd(summary(final_tree)$residuals)^2
orig_data = data

# Do
ci_estimates = boot(orig_data,
                    statistic = ci_tree,
                    mle=final_orig_tree,
                    R=1000,sim="parametric", ran.gen = random_gen)

ci_res = envelope(ci_estimates)

pred_bands_estimates = boot(orig_data,
                            mle = final_orig_tree, statistic = pred_band_tree, R = 1000, sim = "paramet

pred_res = envelope(pred_bands_estimates)
```

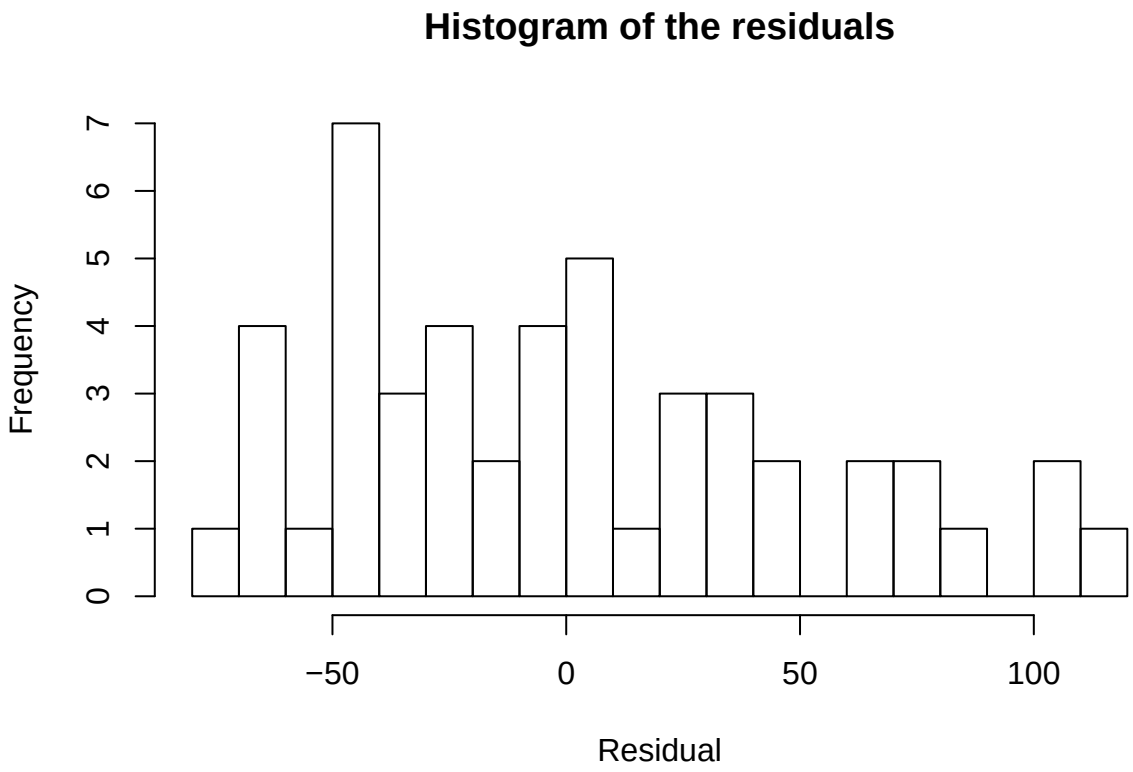
```
## Warning in envelope(pred_bands_estimates): unable to achieve requested
## overall error rate
```

```
plot(x=data$MET, y = data$EX,
     col = "blue", pch = "x", xlab = "MET",
     ylab = "EX",
     main ="95 % bands Non-parametric bootstrap",
     ylim=c(min(ci_res$point[1,], pred_res$point[1,],data$EX),max(ci_res$point[1,], pred_res$point[1,],
lines(x=data$MET, y = pred_res$point[1,], lwd = 3, col = "red")
lines(x=data$MET, y = pred_res$point[2,], lwd = 3, col = "red")
```

```
lines(x=data$MET, y = ci_res$point[1,], lwd = 3, col = "blue")
lines(x=data$MET, y = ci_res$point[2,], lwd = 3, col = "blue")
```




```
## Task 5
tree_model = tree(EX~MET,data=data, control = tree.control(nobs = nrow(data),minsize = 8))
final_tree = prune.tree(tree_model, best=best_size)

preds = predict(final_tree, data)
hist(residuals(final_tree),
     breaks = 20, main ="Histogram of the residuals",
     xlab = "Residual")
```



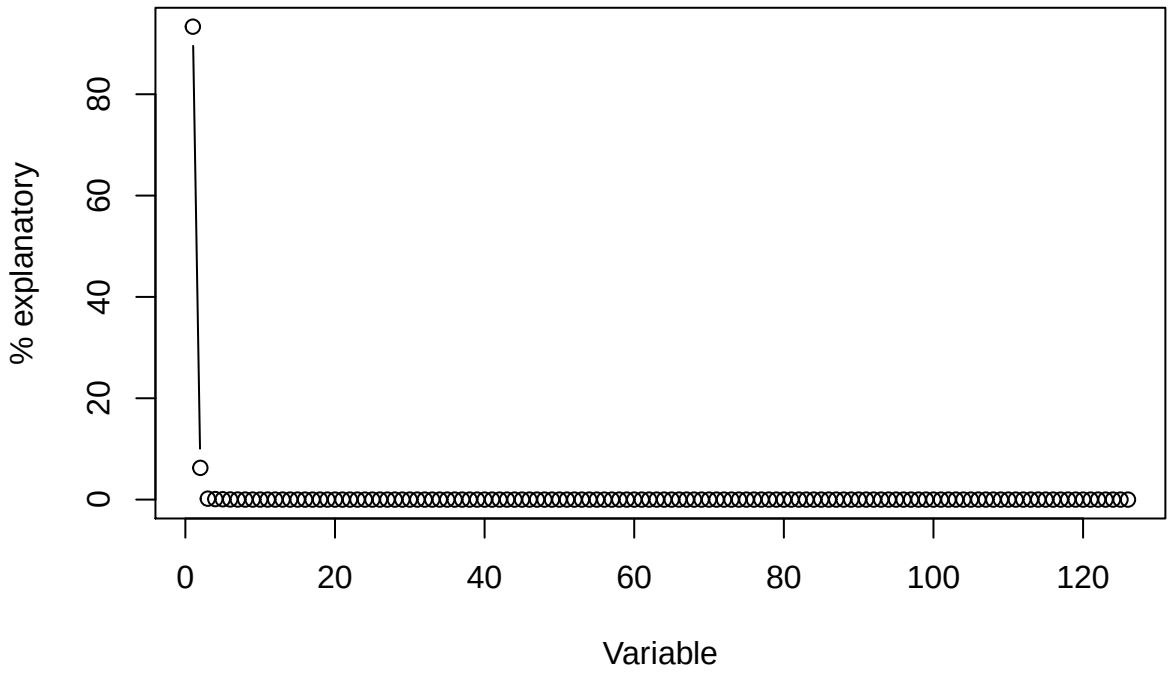

```
# Comment: a chi-square model would be able to handle this nicely.
```

21.4 Assignment 4 - PCA

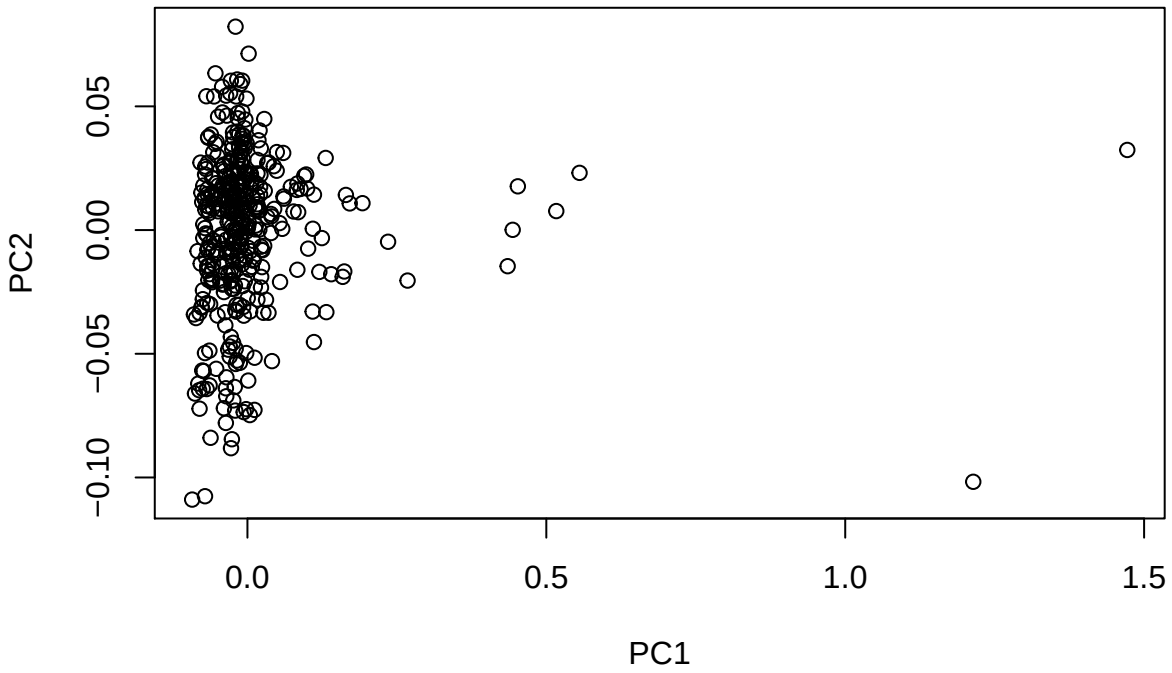
21.4.1 Task 1 - Standard PCA, choose features

Here, the task was to study how the near-infrared spectra of diesel fuels affects the viscosity. Performing a standard PCA, we can clearly see how 93 % is explained by the hidden factor, and 6 % by the second one, PC2.

```
############################
####### ASSIGNMENT 4 #######
############################

data =
  read.csv(
    "~/Desktop/Plugg_Lkpg/HT18/TDDE01/Tentaplugg/Lab_repetitions/Lab2/NIRSpectra.csv",
    sep = ";",
    dec = ",")
featSpace = data[,-c(ncol(data))]
result = prcomp(featSpace)

# the sdev squared will give us the variation.
# Normalizing the variance vector will give us how much variation is explained by each principal compon
lambda=result$sdev^2
explanationFactor = lambda/sum(lambda)*100

names(explanationFactor) = seq(1,length(lambda),1)
plot(explanationFactor, type = "b", xlab = "Variable", ylab = "% explanatory")
```




```
explanationFactor = sort(explanationFactor, decreasing = T)

sumP = 0
i = 0
while (sumP < 99) {
  i = i + 1
  sumP = sumP + explanationFactor[i]


}

nrVarsToInvolve = i


plot(result$x[,1], result$x[,2], xlab = "PC1", ylab = "PC2")
```

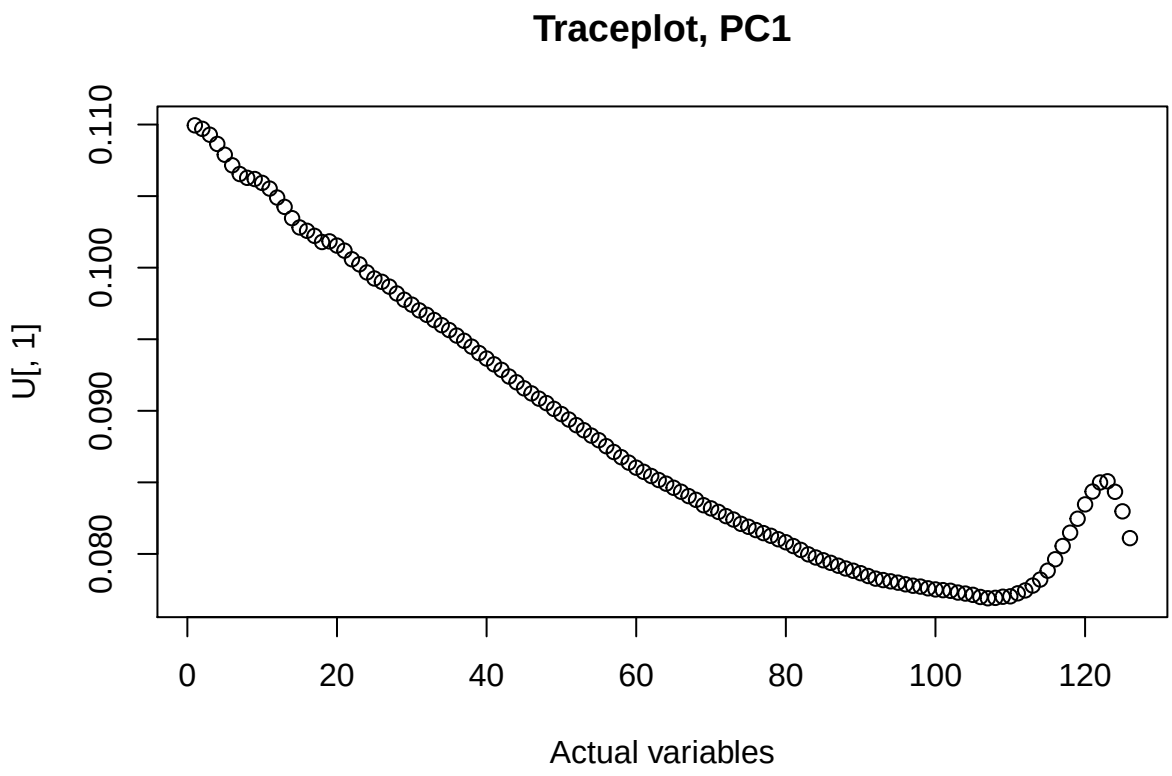


There are a few outliers as can be seen in the plot, although they are not many.

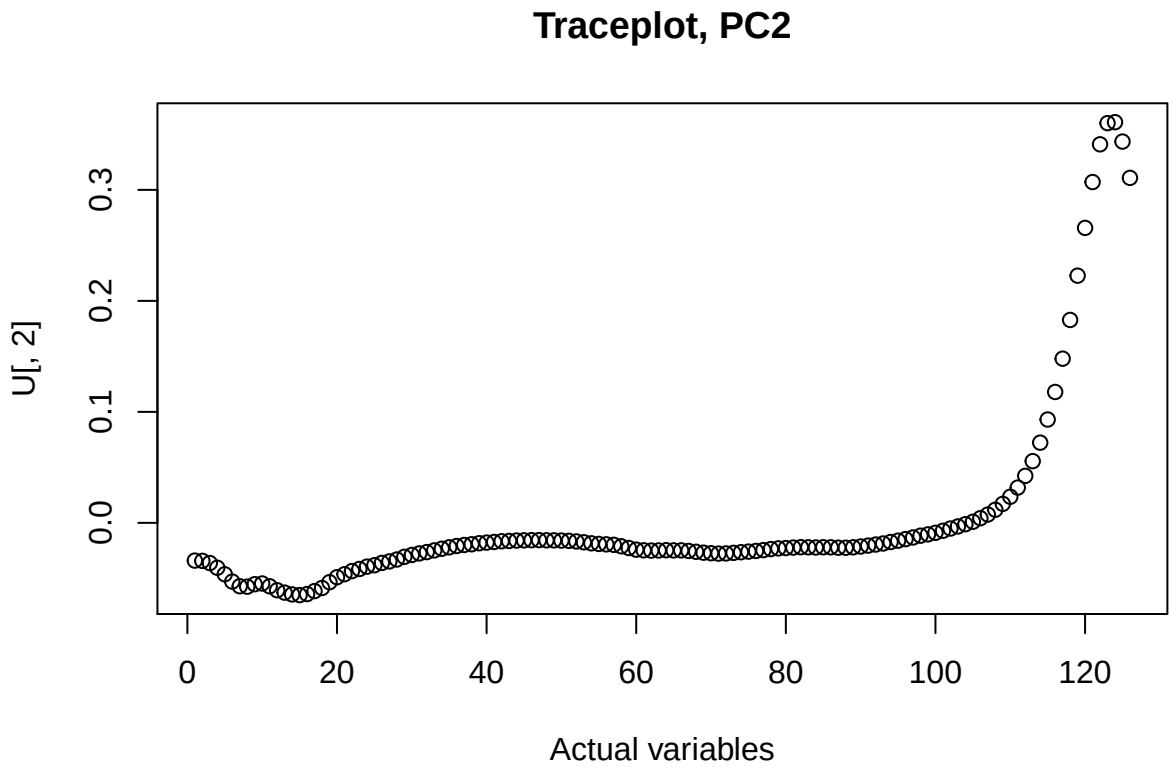
21.4.2 Task 2 - Trace plots

Investigating the trace plots of PC1 and PC2, we can clearly see that PC2 is mainly explained by the last variables. For PC1, we see a decreasing degree of explaining as the variable index increases, followed by a slight tip on the end.

```
# Task 2: Make trace plots

U = result$rotation
plot(U[,1], main="Traceplot, PC1", xlab = "Actual variables")
```



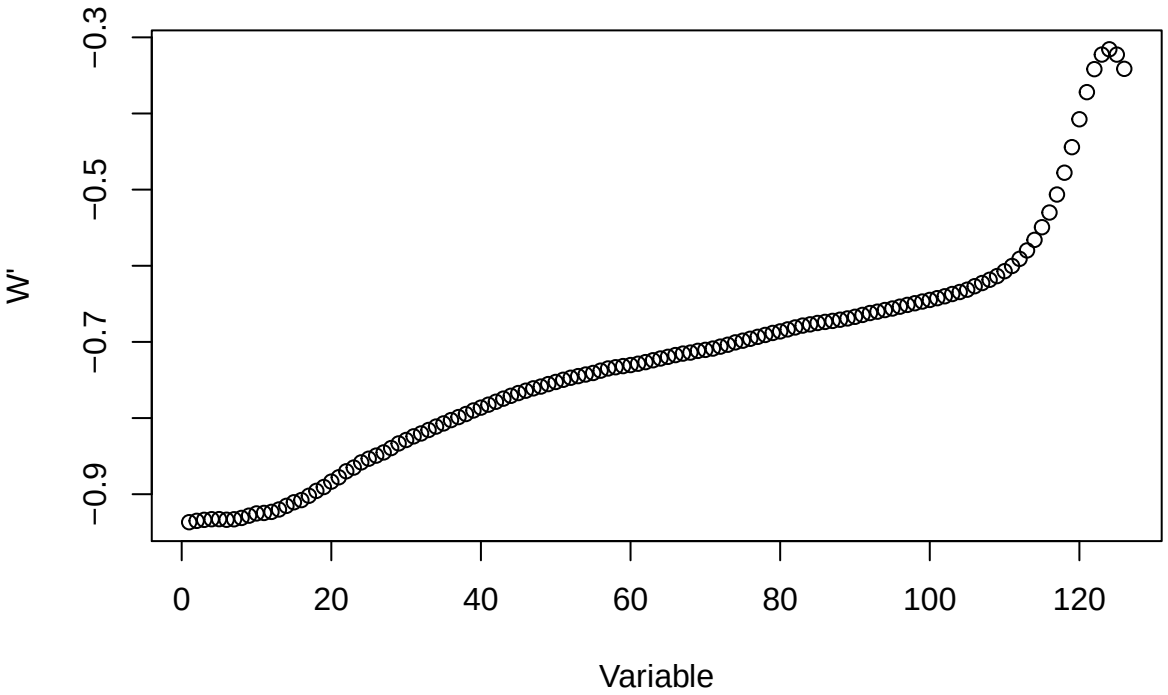

```
plot(U[,2],main="Traceplot, PC2", xlab = "Actual variables")
```



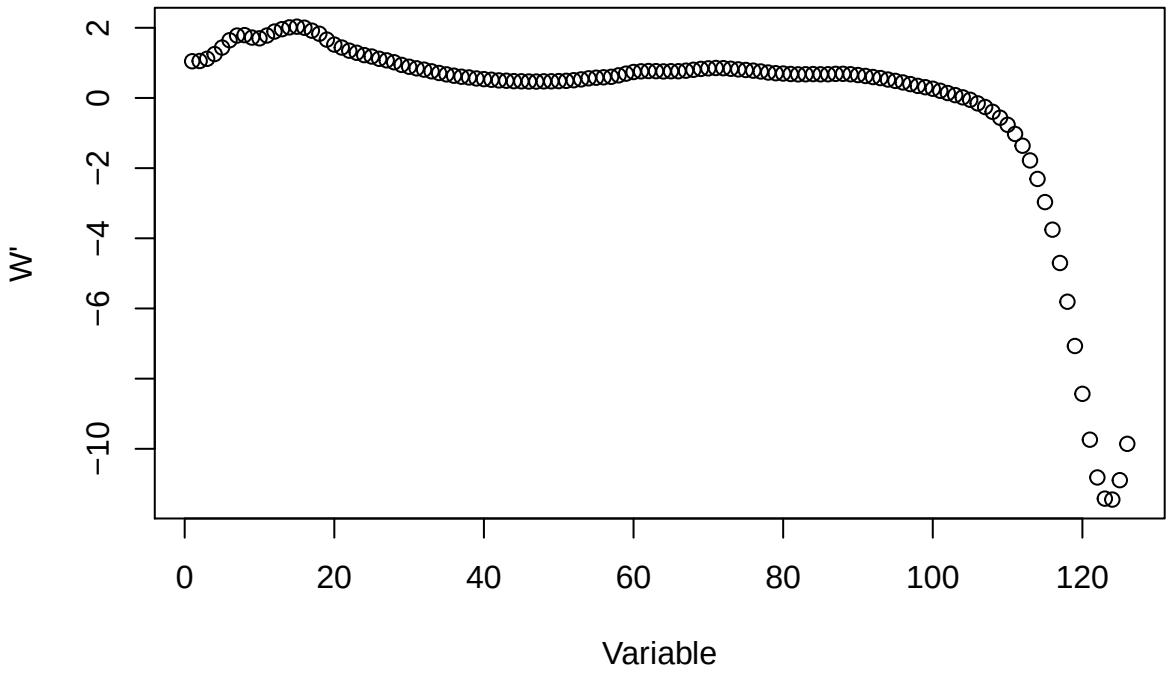
21.4.3 Task 3 - something

Performing the last task, we can see that fastICA yields the same information, but becomes kind of the "inverse" to the PCA. Since they are their own inversions, it means that the variables are statistically independent and also non-gaussian.

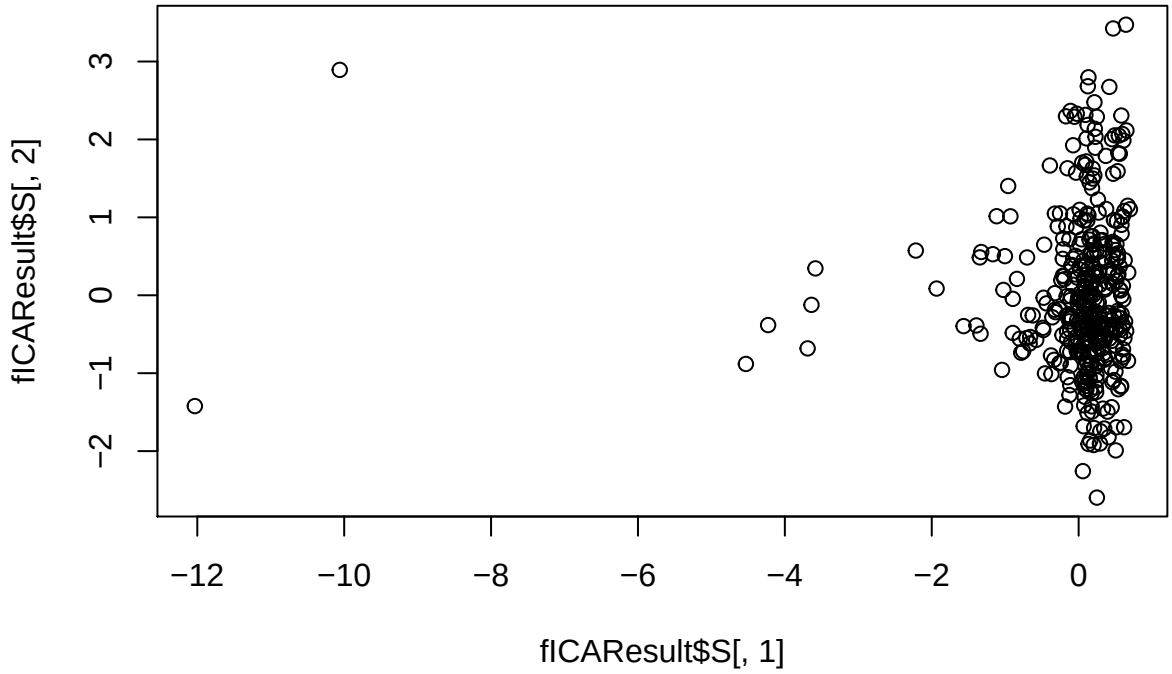
```
# Task 3: Perform independent component analysis
#install.packages("fastICA")
library(fastICA)
#fICA = fastICA(as.matrix(featSpace),)
#X = as.matrix()

# a. Compute W' = K * W
set.seed(12345)
fICAResult = fastICA(as.matrix(featSpace), n.comp = nrVarsToInvolve)
W_prime = fICAResult$K%*%fICAResult$W

plot(W_prime[,1], xlab = "Variable", ylab = "W'")
```




```
plot(W_prime[,2], xlab = "Variable", ylab = "W'")
```




```
plot(x=fICAResult$S[,1],fICAResult$S[,2])
```



22 Lab 3

22.1 Assignment 1 - The kernel trick

In this task, we are asked to perform predictions for a certain location in Sweden between the hours 04:00:00 and 00:00:00 on a specific date. I chose Norrkoping on the 11th of November 2013.

We were asked to to do this using three kernels; one for the distance between the cities, one for the day and one for the time of the day. The first task was to set the width parameters. I chose the parameters as follows.

```
h_distance <- 100000 # reasonable that it is the same within 200km more or less
h_date <- 10 # If the year was extra cold. Reasonable to have within one year.
h_time <- 2 # About an hour ish. Reasonable.
```

I consider it reasonable that we have more or less the same climate within a radius of 100 km, which is why I chose `h_distance` as above. I chose `h_date` as 180, one unit is one day, and I believe that same years are colder than others, thus yielding a colder temperature overall, which I want to yield signifiance then. The parameter`h_time` I chose to be 4, as one unit is about 15 minutes.

After this, we initialize all the necessary functions and commence the problem.

```
set.seed(1234567890)
library(geosphere)
stations <-
  read.csv(
    "~/Desktop/Plugg_Lkpg/HT18/TDDE01/Tentaplugg/Lab_repetitions/Lab3/stations.csv",
    stringsAsFactors=FALSE,
    fileEncoding="latin1")
temps <-
  read.csv(
    "~/Desktop/Plugg_Lkpg/HT18/TDDE01/Tentaplugg/Lab_repetitions/Lab3/temps50k.csv")
st <- merge(stations,temps,by="station_number")

h_distance <- 100000
  h_date <- 40
  h_time <- 2
a <- 58.4274 # The point to predict (up to the students)
b <- 14.826
date <- "2013-11-04" # The date to predict (up to the students)
times <- c("04:00:00", "06:00:00", "08:00:00", "10:00:00",
           "12:00:00", "14:00:00", "16:00:00", "18:00:00",
           "20:00:00", "22:00:00", "00:00:00")
temp <- vector(length=length(times))
# My own code


gauss_kern = function(u) {
  return(exp(-(u^2)))
}

convert_to_mins = function(time) {

  return(as.numeric(substr(time,1,2))*60 + as.numeric(substr(time,4,5)))
}

st$time = convert_to_mins(st$time)
st$day_num = as.numeric(factor(substr(as.character(temps$date),
                                      start = 6,
                                      stop = 10)))
# point1 is the longitude and latitude
# If distance_calc = True, the distance is sent
dist_kernel = function(point1, point2, distance = NULL) {
  if (is.null(distance)) {
```

```
    distance = distHaversine(p1=point1, p2 = point2)
  }

  return(exp(-(distance/h_distance)^2))
}

time_kernel = function(time1, time2) {
  timeDiff = abs((time1 - time2)/60)


  if (length(timeDiff) > 1) {
    timeDiff = apply(as.matrix(timeDiff), 1, function(row) {
      if (row > 12) {
        return(12 - row %% 12)
      } else {
        return(row)
      }
    })
  }

  return(exp(-(timeDiff^2)/(h_time^2)))
}

date_kernel = function(date1, date2) {
  if (length(date2 > 1)) {
    daysBetween = apply(as.matrix(date2), 1, function(d2) {
      min(366 - abs(date1 - d2), abs(date1 - d2))
    })
  } else {
    daysBetween = min(366 - abs(date1 - date2), abs(date1 - date2))
  }
  return(exp(-(daysBetween^2)/(h_date^2)))
}

# Kernels created. Now start the prediciton

# Now, time to predict the temperature for point a,b
# pred_point is the longitude and latitude as c(latitude, longitude)
# data is st sent.
# Is assumed to have time
# converted to minutes and day_num, which is the day of the year.
sum_kernels = function(data, pred_point, times, date) {
  times = convert_to_mins(times)
  longitude = pred_point[2]
  latitude = pred_point[1]
  # Convert date to its number on the year
  date = which(
    substr(
      as.character(date),
      start=6,
      stop=10) == levels(
        factor(
          substr(as.character(data$date),
```

```
                        start = 6, stop = 10)
          )
        )
    )

  # Now sum up the kernels!!!!

  k_date = date_kernel(date,data$day_num)
  k_distance = dist_kernel(c(longitude,latitude),
                           matrix(c(data$longitude,data$latitude), ncol=2, byrow=F))
  k_time = apply(as.matrix(times),1,time_kernel,data$time)

  # Sum up the kernels
  kern = apply(k_time, 2, function(time_k, k_date, k_distance) {
    return(time_k + k_date + k_distance)
  },k_date,k_distance)

  kern_t = apply(kern,2,function(col) {
    return(col*data$air_temperature)
  })

  # Now calculate the temperature for each time period
  n_time_periods = (dim(kern_t)[2])

  temps = as.numeric()

  for (i in 1:n_time_periods) {
    temps[i] = sum(kern_t[,i])/sum(kern[,i])
  }
  print(temps)
  return(temps)
}

prod_kernels = function(data, pred_point, times, date) {
  times = convert_to_mins(times)
  longitude = pred_point[2]
  latitude = pred_point[1]
  # Convert date to its number on the year
  date = which(substr(
    as.character(date),
    start=6, stop=10) ==
      levels(factor(substr(as.character(data$date),
                           start = 6, stop = 10)))
    )
  # Now sum up the kernels!!!!
  k_date = date_kernel(date,data$day_num)
  k_distance = dist_kernel(c(longitude,latitude),
                           matrix(c(data$longitude,data$latitude),
                                  ncol=2,
                                  byrow=F))

  k_time = apply(as.matrix(times),1,time_kernel,data$time)
```

```
  # Sum up the kernels
  kern = apply(k_time, 2, function(time_k, k_date, k_distance) {
    return(time_k*k_date*k_distance)
  },k_date,k_distance)

  kern_t = apply(kern,2,function(col) {
    return(col*data$air_temperature)
  })

  # Now calculate the temperature for each time period
  n_time_periods = (dim(kern_t)[2])

  temps = as.numeric()
  for (i in 1:n_time_periods) {
    temps[i] = sum(kern_t[,i])/sum(kern[,i])
  }
  return(temps)
}
```

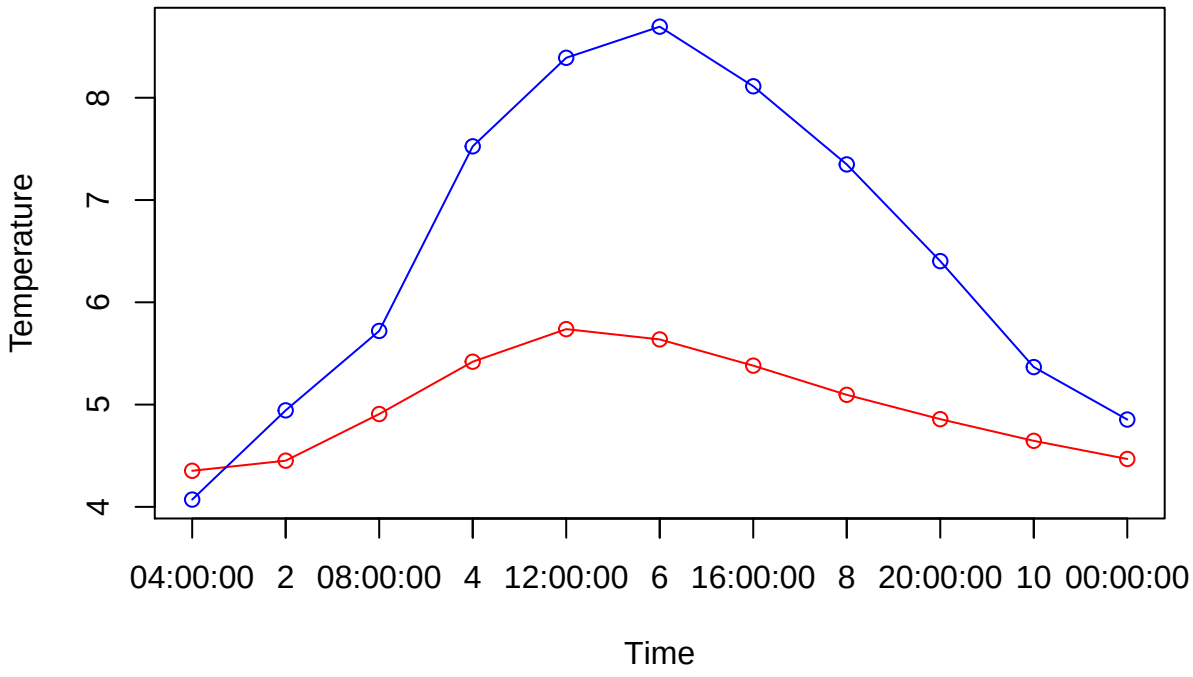
22.1.1 Predicting temperature for the point and date in question

The code for predicting can be seen below.

```
temps = sum_kernels(data = st, pred_point = c(a,b), times = times, date = date)
```

```
##  [1] 4.352925 4.451577 4.907520 5.419191 5.737782 5.637129 5.380954
##  [8] 5.095227 4.857640 4.645672 4.468328
```

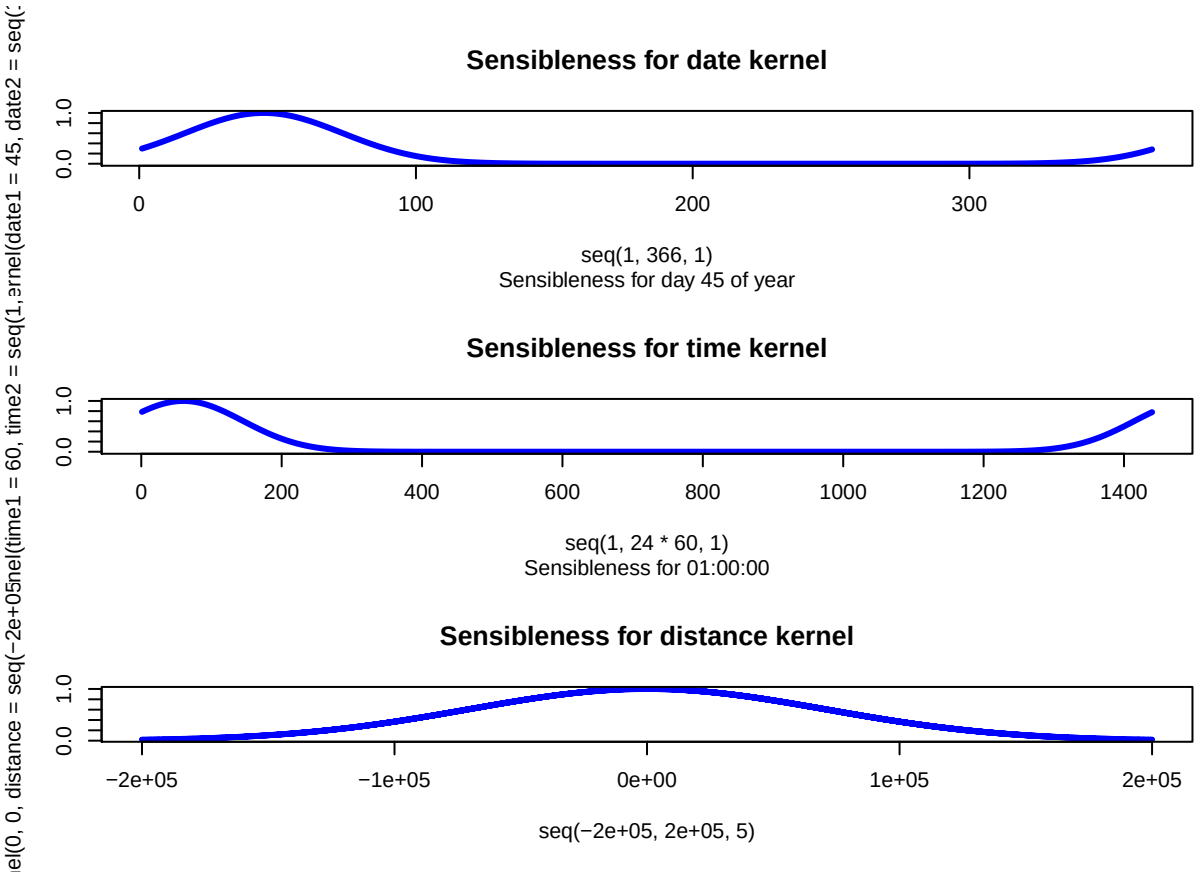
```
temps_prod = prod_kernels(data = st, pred_point = c(a,b), times = times, date = date)
plot(temps, type="o",
     ylim = c(min(temps_prod, temps),
              max(temps_prod, temps)),
     col = "red", xlab = "Time",
     ylab = "Temperature")
lines(temps_prod, type = "o", col = "blue")
axis(1, at=1:11, labels=times)
```



22.1.2 Showing sensitivity

We were asked to show that our kernels are sensitive. Thus, I here generate a few plots to show how each of the kernels individually are sensitive.

```
# Show sensibleness
# Show date
par(mfrow=c(3,1))
plot(x=seq(1,366,1), y=date_kernel(date1 = 45,
                                   date2 = seq(1,366,1)),
     type="l",
     lwd=3,
     col = "blue",
     main = "Sensibleness for date kernel",
     sub ="Sensibleness for day 45 of year")
# Show sensibleness for time kernel for 06:00:00
plot(x=seq(1,24*60,1),
     y = time_kernel(time1 = 60, time2 = seq(1,24*60,1)),
     type="l", lwd=3, col = "blue",
     main ="Sensibleness for time kernel",
     sub = "Sensibleness for 01:00:00")
# Show sensibleness for distance
plot(x=seq(-200000,200000,5),
     y=dist_kernel(0,0,distance = seq(-200000,200000,5)),
     type = "l", lwd=3, col ="blue",
     main="Sensibleness for distance kernel")
```




```
par(mfrow=c(1,1))
```

We can clearly see that the kernels are less sensitive the longer the distance from their actual values. Thus, we have shown their sensitivity.

22.2 Task 2 - SVM

```
##### Assignment 2 #####

# Task 1

library(kernlab)

data(spam)
data = spam
gauss_width = 0.05
C = c(0.5,1,5)

n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.7))
train=data[id,]
val=data[-id,]
```

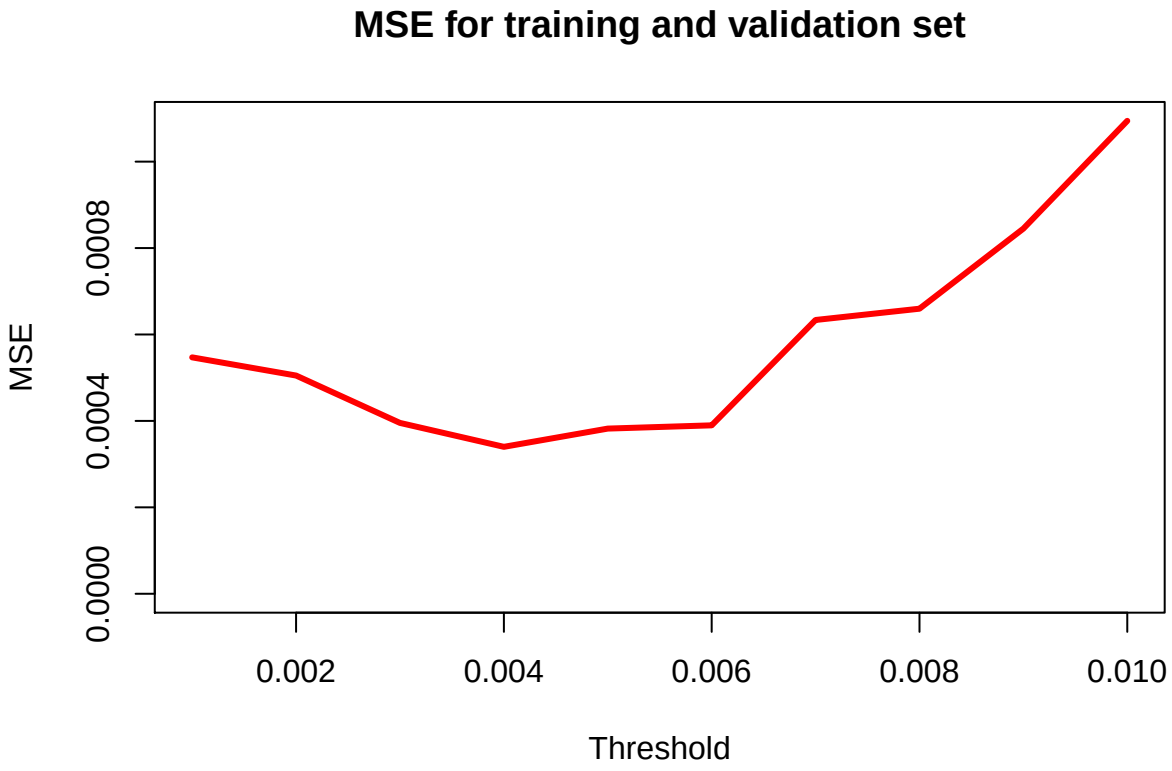
```
for (c in C) {
  svm_model = ksvm(type~., data = train, C=c, kernel="rbfdot", kpar=list(sigma=gauss_width))
  pred_vals = predict(svm_model, val)
  print("Misclassification rate")
  print((1 - sum(diag(table(Actual = val$type, Predicted = pred_vals)))/sum(table(Actual = val$type, Pr
}

# Best model is with C = 5

# Comment: C has the purpose of regularizing and avoiding overfitting by reducing the coefficients in t
```

22.3 Neural networks - training a NN to learn the Sin function

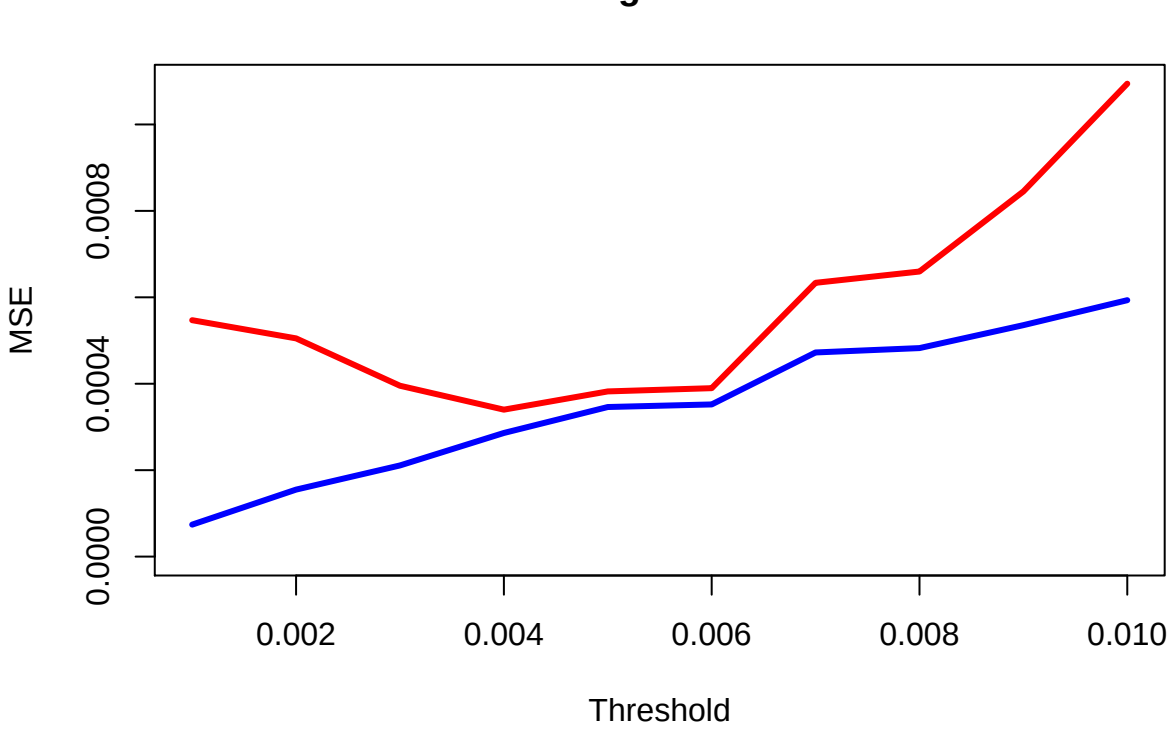
In this task, we were asked to implement a neural network with 10 hidden layers. To obtain the optimal neural network, we try with different thresholds and use the optimal neural network with the threshold that yields the smallest MSE on the predictions. We arrive at the conclusion the threshold 0.004 is optimal on the validation, and we can see in the plot the it approximates the function very nicely.

```
# Reuse function from lab 1
MSE = function(y, y_hat) {
  n = length(y)
  return((1/n)*sum((y - y_hat)^2))
}

library(neuralnet)
set.seed(1234567890)
Var <- runif(50, 0, 10)
trva <- data.frame(Var, Sin=sin(Var))
tr <- trva[1:25,] # Training
va <- trva[26:50,] # Validation
# Random initialization of the weights in the interval [-1, 1]
winit <- runif(50,-1,1)

MSE_val = as.numeric()
MSE_train = as.numeric()
for(i in 1:10) {
    nn <- neuralnet(Sin ~ Var, data = tr,
                    threshold = (i/1000), startweights = winit, hidden = 10)
    tr_preds = compute(nn, tr$Var)$net.result
    val_preds = compute(nn, va$Var)$net.result
    MSE_val[i] = MSE(va$Sin, val_preds[,1])
    MSE_train[i] = MSE(tr$Sin, tr_preds[,1])
}

# Which error is the best?
plot(x=(seq(1,10,1)/1000),
     y=MSE_val,
     main = "MSE for training and validation set",
     xlab = "Threshold",
     ylab = "MSE",
     type = "l",
     lwd = 3,
     col = "red",
```

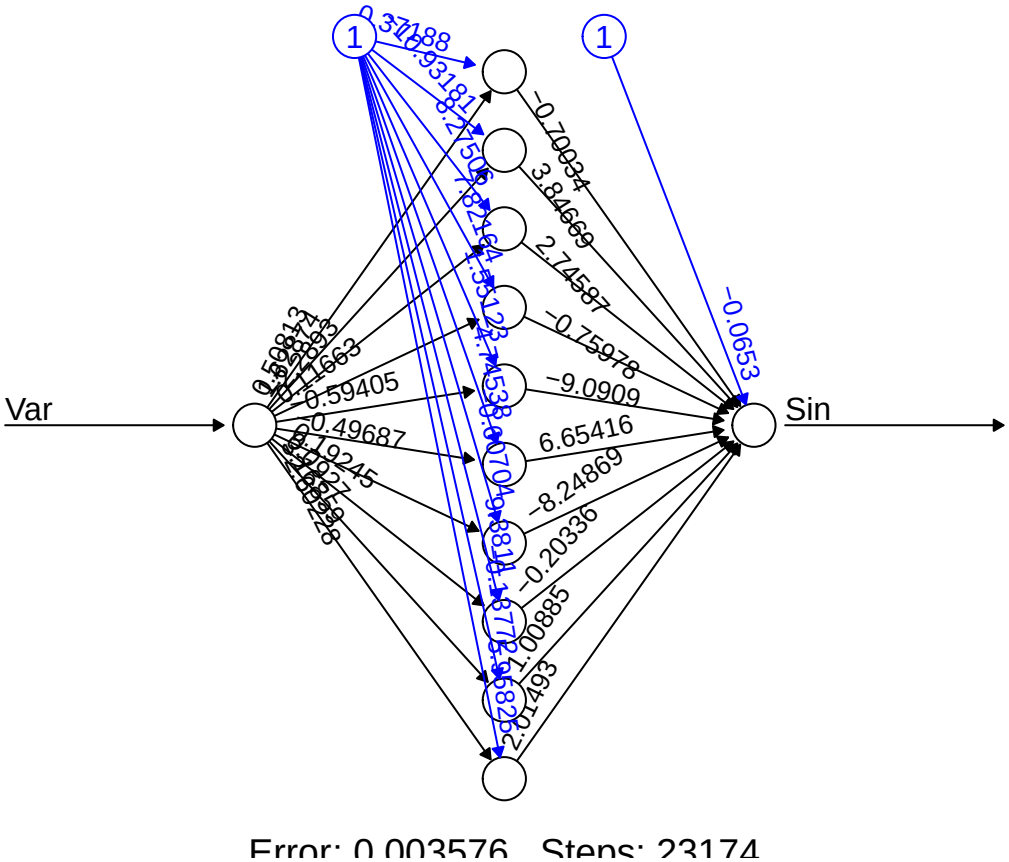
```
     ylim=c(0,max(MSE_train, MSE_val)))
```



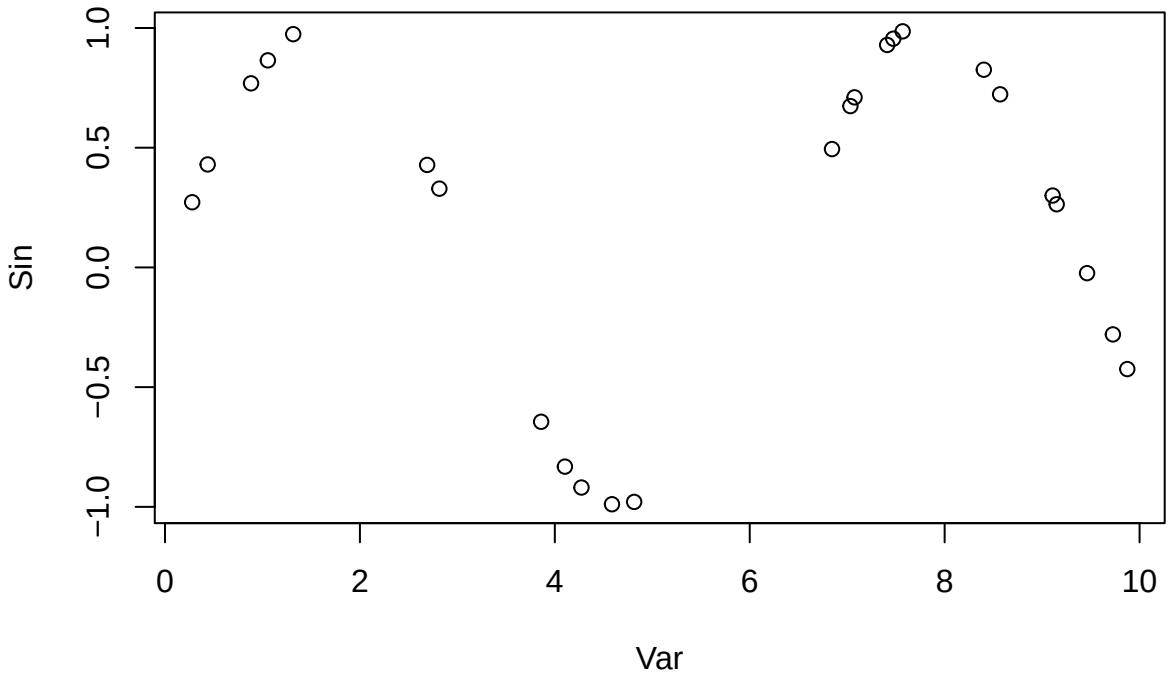

```
lines(x=(seq(1,10,1)/1000), y = MSE_train, lwd = 3, col = "blue")
```



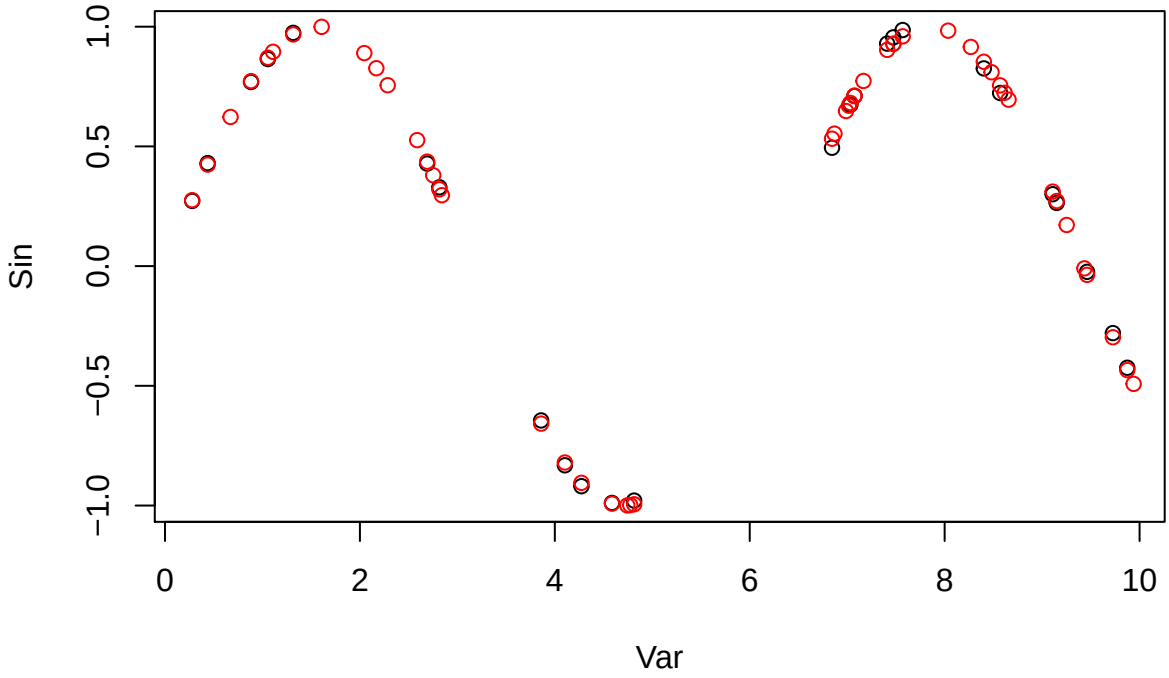
```r
# Generate thresholds again
bestThr = (seq(1,10,1)/1000)[which.min(MSE_val)]

# Plot the best one
plot(nn <- neuralnet(Sin ~ Var,
                     data = tr,
                     threshold = bestThr,
                     startweights = winit,
                     hidden = 10),
     rep = "best")
```




```r
  # Plot of the predictions (black dots) and the data (red dots)
plot(prediction(nn)$rep1)
```




```
points(trva, col = "red")
```



23 Special tasks

23.1 Special task 1 lab 1

To implement the kNN classifier is done below.

```r
library(readxl)
data = read_excel("~/Desktop/Plugg_Lkpg/HT18/TDDE01/Tentaplugg/Lab_repetitions/Lab1/spambase.xlsx")

knearest = function(data, K, newData) {
  #data is the training data,
  #newData returns the predicted class probabilities
  # for newData by using K-nearest neighbour approach.
  responsesX = as.matrix(data[,ncol(data)])
  responsesY = as.matrix(data[,ncol(newData)])
  X = as.matrix(data[,-c(ncol(data))])
  Y = as.matrix(newData[,-c(ncol(newData))])
  Xhat = X/sqrt(rowSums(X^2))
  Yhat = Y/sqrt(rowSums(Y^2))
  C_matrix = Xhat%*%t(Yhat)
  D_matrix = 1 - C_matrix
  #For the data test set,
  # take the distance to the training data points,
  # take the k nearest neighbours and classify them
  # as the average of the training set's labels of the k points

  # return the indices of the k nearest neighbours
  # for all different new data points, generating a matrix with the k nearest neighbou
  indicesKnearest = apply(D_matrix, 2, function(col, k){
    names(col) = seq(1, length(col), 1)
    col = sort(col)
    return(strtoi(names(col[1:k])))
  }, K)

  classifications = apply(indicesKnearest, 2, function(row) {
    # Classify according to the mean of the training data
    avg = mean(responsesX[row,])
    return(round(avg))
  })
  return(classifications)
}

# Use same seed and partition as before.
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]
test=data[-id,]

labelsTest = knearest(train, 30, test)
labelsTrain = knearest(train, 30, train)
#table(train$Spam, labelsTrain)
#table(test$Spam, labelsTest)

#print("misclassification rate train")
#1 - sum(diag(table(train$Spam, labelsTrain)))/sum(table(train$Spam, labelsTrain))
#print("misclassification rate test")
#1 - sum(diag(table(test$Spam, labelsTest)))/sum(table(test$Spam, labelsTest))
```

These results can be compared to the task in lab 1, where

```r
classificationRate = function(confMatrix) {
  return(sum(diag(confMatrix))/sum(confMatrix))
}
# Task 4
library("kknn")

data$Spam = factor(data$Spam)
# Use same seed and partition as before.
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]
test=data[-id,]

kkn.fit = kknn(Spam ~., train = train, test = train, k = 30)
#print("misclassification rate train")
#table(train$Spam, kkn.fit$fitted.values)
#1 - classificationRate(table(train$Spam, kkn.fit$fitted.values))
kkn.fit = kknn(Spam ~., train = train, test = test, k = 30)
#print("misclassification rate test")
#table(test$Spam, kkn.fit$fitted.values)
#1 - classificationRate(table(test$Spam, kkn.fit$fitted.values))
```

So my implementation produces worse results on the training set, but slightly better on the test set for this particular seed.

23.2 Neural network - implementing backpropagation

```r
set.seed(1234567890)
Var <- runif(50, 0, 10)
trva <- data.frame(Var, Sin=sin(Var))
tr <- trva[1:25,] # Training
va <- trva[26:50,] # Validation

w_j = runif(10, -1,1)

b_j = runif(10,-1,1)
w_k = runif(10,-1,1)
b_k = runif(10,-1,1)

learning_rate = 1/(nrow(tr)^2)

n_iter = 5000
error = rep(0, n_iter)
error_va = rep(0,n_iter)

# Implementing stochastic gradient descent

for (i in 1:n_iter) {

  # Looping through the training set to evaluate the current error. Calculating the current prediction
  for (n in 1:nrow(tr)) {
```

```
    z_j = tanh(w_j* tr[n,]$Var + b_j)
    y_k = sum(w_k * z_j) + b_k
    error[i] = error[i] + (y_k - tr[n,]$Sin)^2
  }

  for (n in 1:nrow(va)) {
    z_j = tanh(w_j* va[n,]$Var + b_j)
    y_k = sum(w_k * z_j) + b_k
    error_va[i] = error_va[i] + (y_k - va[n,]$Sin)^2
  }

  #cat("i: ", i, ", error: ", error[i]/2, ", error_va: ", error_va[i]/2, "\n")
  #flush.console()

  # Now train the network on all data points!

  for (n in 1:nrow(tr)) {
    # First do forward propagation to calculate the current derivates and partial derivates
    z_j = tanh(w_j* tr[n,]$Var + b_j)
    y_k = sum(w_k * z_j) + b_k

    # Now do backpropagation to change everything
    d_k = y_k - tr[n,]$Sin
    d_j = (1 - z_j^2)*w_k*d_k
    partial_w_k = d_k*z_j
    partial_b_k = d_k
    partial_w_j = d_j * tr[n,]$Var
    partial_b_j = d_j
    w_k = w_k - learning_rate*partial_w_k
    b_k = b_k - learning_rate*partial_b_k
    w_j = w_j - learning_rate*partial_w_j
    b_j = b_j - learning_rate*partial_b_j
  }
}

w_j
b_j
w_k
b_k

plot(error/2, ylim=c(0, 5))
points(error_va/2, col = "red")

# prediction on training data
pred <- matrix(nrow=nrow(tr), ncol=2)
for(n in 1:nrow(tr)) {
  z_j <- tanh(w_j * tr[n,]$Var + b_j)
  y_k <- sum(w_k * z_j) + b_k
  pred[n,] <- c(tr[n,]$Var, y_k[1])
}
plot(pred)
points(tr, col = "red")
# prediction on validation data
```

```
pred <- matrix(nrow=nrow(tr), ncol=2)
for(n in 1:nrow(va)) {
  z_j <- tanh(w_j * va[n,]$Var + b_j)
  y_k <- sum(w_k * z_j) + b_k
  pred[n,] <- c(va[n,]$Var, y_k[1])
}
plot(pred)
points(va, col = "red")
```

23.3 Task 2 lab 1 - KNN density estimation

In task 2, we were to produce k-nearest neighbour density estimation for the variable speed from the cars data set. the result can be found below. Since we are working in one dimension, we cannot really, and do not really need either, to work with cosine similarities. It is important to note that we need

$$K > max_{i \in D} \quad Count(i)$$

where D is the the domain of the data. Otherwise, we will get null values from my algorithm, as the distance radius R will be zero. I would be glad to hear about how this could be worked around, apart from choosing to map against values not matching any data points in the data set. Now I did a linear extrapolation of the radius if it happens to be 0. That does not happen for $K = 6$, so it does not matter in this case, but I am curious whether K-nearest neighbour density estimation can work for $K \leq max_{i \in D} Count(i)$

```
attach(mtcars)

cars = cars
detach(mtcars)
k = 6

volume_N_sphere = function(nDim, radius) {
  n = nDim
  R = radius
  return(((pi^(n/2))/gamma(n/2 + 1))*R^n)
}

kNearestNeighDensEstimation = function(data, k = 6, newData) {
  # data is the training data
  # newData is the data
  N = nrow(data)
  nDim = ncol(data)
  X = as.matrix(data) # We do not assume responses to exist
  Y = as.matrix(newData) # We do not assume responses to exist
  D_matrix = abs(apply(Y, 1, function(row) {
    return(abs(row - X))
  }))
  # return the indices of the k+1 nearest neighbours
  # for all different new data points, generating a matrix with the k nearest neighbours
  # Since we include the point itself, we return k+1 nearest radiuses.
  radiuses = apply(D_matrix, 2, function(col, k) {
    sortedCol = sort(col)
    return(sortedCol[k])
  }, k+1)
```

```r
  # Making sure no radius is 0
  for (i in 1:length(radiuses)) {
    if (radiuses[i] == 0) {
      #If it is 0, do linear extrapolation
      radiuses[i] = (radiuses[i-1] + radiuses[i+1])/2
    }
  }
  #Density estimation
  densities = apply(as.matrix(radiuses), 1, function(R, nDim) {
    return(k/(N*volume_N_sphere(nDim, R)))
  }, nDim)
  return(densities)
}

plot_dens = seq(0,max(cars$speed), by = 0.01)

densities = kNearestNeighDensEstimation(data = as.matrix(cars$speed),
                                        k=6, newData = plot_dens)
hist(cars$speed,
     breaks = 20,
     freq = FALSE,
     ylim=c(0, max(densities)),
     xlab = "Speed",
     main = "Histogram vs k-nearest density, K= 6")
lines(x=plot_dens, y = densities, lwd = 3, col = "red")
```