Assignment 1

To train and test different models based on the handwritten digits, the data was pseudo-randomly divided into three separate sets of data.

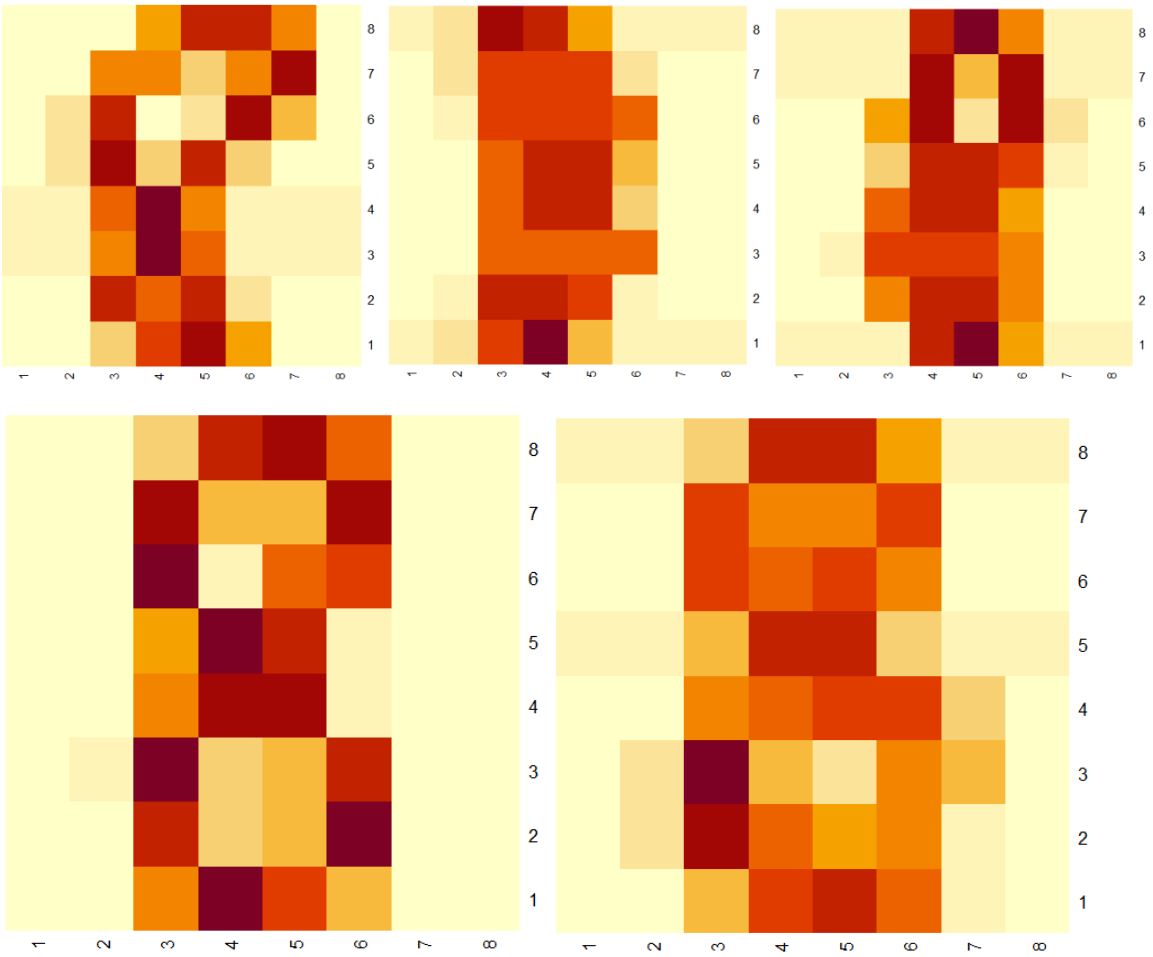
Confusion matrices were created to get an overview of how well the 30-nearest neighbour classifier would work on the handwritten digits data. The figure bellow shows the confusion matrix for the 30-nearest neighbour classifier for the training data compared with itself. Every column is the predicted digit, and every row is the actual digit. Numbers on the diagonal therefore indicates the number of correct predictions. Any other coordinate in the matrix indicates an incorrect prediction.

	0	1	2	3	4	5	6	7	8	9
0	177	0	0	0	1	0	0	0	0	0
1	0	174	9	0	0	0	1	0	1	3
2	0	0	170	0	0	0	0	1	2	0
3	0	0	0	197	0	2	0	1	0	0
4	0	1	0	0	166	0	2	6	2	2
5	0	0	0	0	0	183	1	2	0	11
6	0	0	0	0	0	0	200	0	0	0
7	0	1	0	1	0	1	0	192	0	0
8	0	10	0	1	0	0	2	0	190	2
9	0	3	0	4	2	0	0	2	4	181

There are high values on the diagonal so many predictions were correct. The misclassification rate was 4.239% which is adequate. The 30-nearest neighbour classifier was also applied on the test data set compared with the training data set. The misclassification rate for this set was 4.916% and the confusion matrix is shown in the following figure. This misclassification rate seems reasonable as the classifier should give the best predictions when it is compared with itself.

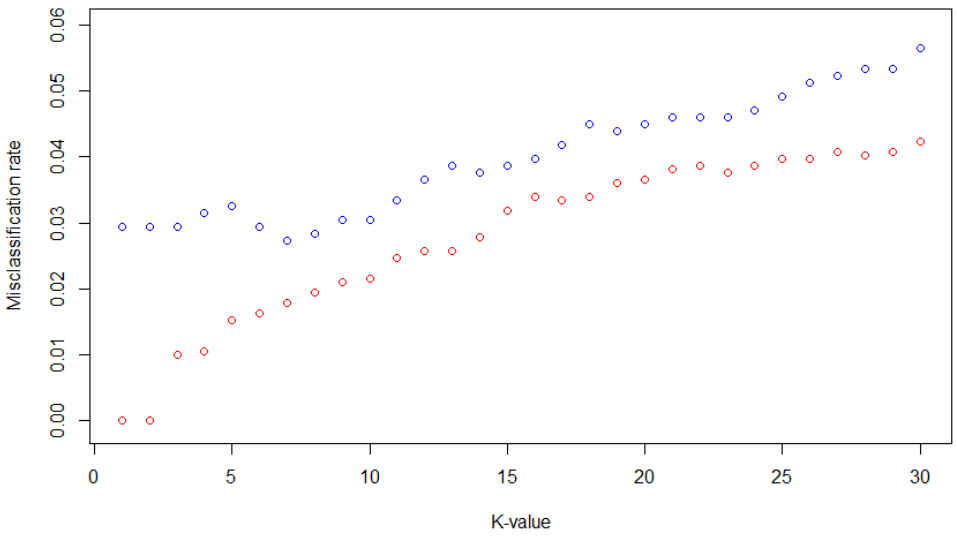
	0	1	2	3	4	5	6	7	8	9
0	97	0	0	0	0	0	1	0	0	0
1	0	91	3	0	0	0	0	0	0	3
2	0	0	93	1	0	0	0	0	1	0
3	0	0	0	95	0	0	0	2	1	0
4	1	0	0	0	89	0	1	5	1	3
5	0	1	0	1	0	79	1	0	0	5
6	0	0	0	0	0	0	94	0	0	0
7	0	2	0	0	0	1	0	91	1	0
8	0	3	0	1	0	0	1	0	86	0
9	0	0	0	4	0	0	0	2	1	94

Some of the handwritten digits were easy to make a correct prediction for and some were more difficult. This is due to the difference between the different handwritings in the data set. To visualize this, heat maps were used to show roughly how the handwriting looks. The three most difficult to predict eights are show bellow together with the two easiest to predict cases.



As the figures indicate, the difficult to predict eights are hard to recognize as being eights. They just lack details. You can distinguish a one (of two) ovals in the leftmost and the rightmost but not both ovals that make up an eight. The easy to predict eights are, in contrast, very easy to see that they are eights even if the resolution is just 8x8 pixels.

To get an idea of what a good K-value for this kind of data would be, the misclassification error was tested for every K-value between 1 and 30. In the figure bellow, the misclassification rate for the k-values are plotted for both the training (red dots) and the validation (blue dots) sets.

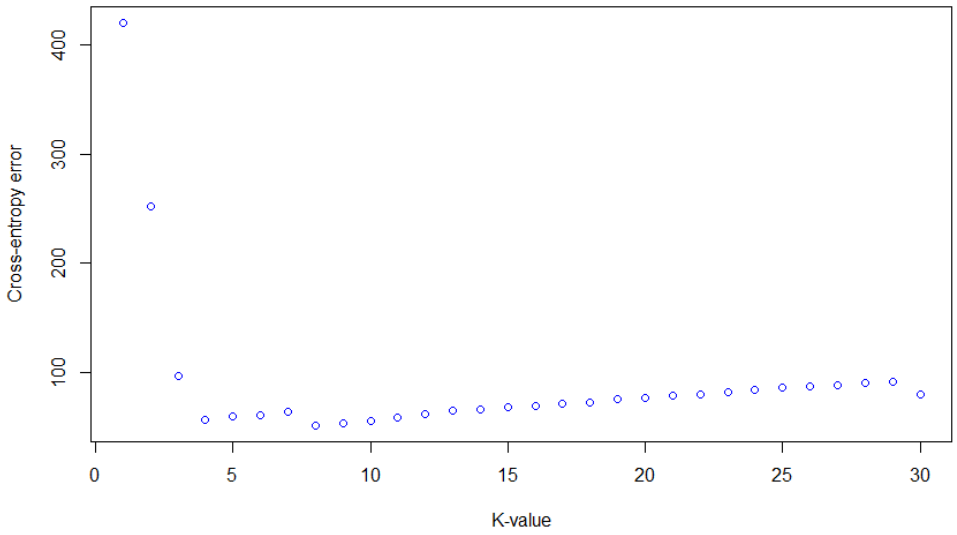


The misclassification rate for the training data is 0 for k=1 and k=2 which makes sense as the nearest neighbour of a data point is always itself. For the validation, these lower K-values are not optimal. The misclassification rate increases slowly up until k=5, then it dips and achieves its' lowest value at k=7. It then increases as the K-value increases. This output makes sense as the predictions slowly get better at first because as the number of neighbours increasing means that outliers in the data does not skew the predictions. This effect only needs a small number of extra neighbours. After that, the predictions get worse as the prediction accounts for data points far away.

7 was the best K-value for this data. Using the test set, a misclassification error of 3.870% was found for the model using k=7 which is about 1 percentage point worse than the error for the validation data.

Cross entropy

As an alternative to misclassification rate, cross-entropy was also used to check the error of the classification. In the figure bellow, the cross-entropy errors are plotted for the K-values between 1 and 30.



Here, the optimal K-value is 8. This is the K-value with the best predictions. If a model is good according to the cross-entropy error, it means that it predicts correctly with a high amount of certainty and when it guesses incorrectly, it is not very certain (the choice with the highest chance is not very high).

The cross-entropy loss function is good as it incorporates the chances of the classification instead of just the predicted class. A prediction with 60% certainty is not treated the same as a prediction with 90% certainty for example.

Assignment 2

The first step is to clean the data by getting rid of the columns that are not of interest and scaling the data. The purpose was to predict the motor_UPDRS from a set of voice characteristics so all other columns were removed, leaving motor_UPDRS as column 1 and then 16 feature variables as column 2:17. The scaling was done with the scale() function which scales the data such that the mean of every column is 0 and the variance is 1. The data now had the data structure "large matrix", and this had to be converted to data frame since that is what has to be sent in to the model. The next step was to split the data into training and test data with a 60/40 split. This was done by first randomly selecting 60 % of the rows for training and then assigning the rest for testing.

A linear regression model was computed with the lm() function. The result from the model was the following:

```
Call:
lm(formula = motor_UPDRS ~ ., data = train)

Residuals:
    Min      1Q  Median      3Q     Max
-3.0162 -0.7340 -0.1084  0.7310  2.1892

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)     0.006789   0.015790   0.430 0.667252
Jitter...       0.180301   0.144277   1.250 0.211496
Jitter.Abs.    -0.169795   0.040856  -4.156 3.32e-05 ***
Jitter.RAP     -5.087728  18.186914  -0.280 0.779688
Jitter.PPQ5    -0.071963   0.084712  -0.850 0.395659
Jitter.DDP      5.068468  18.190294   0.279 0.780541
Shimmer         0.590434   0.205314   2.876 0.004055 **
Shimmer.dB.    -0.172730   0.139396  -1.239 0.215380
Shimmer.APQ3   32.009848  77.023255   0.416 0.677738
Shimmer.APQ5   -0.387249   0.113730  -3.405 0.000669 ***
Shimmer.APQ11   0.310751   0.062288   4.989 6.37e-07 ***
Shimmer.DDA   -32.325644  77.023042  -0.420 0.674739
NHR            -0.186169   0.045766  -4.068 4.85e-05 ***
HNR            -0.239653   0.036570  -6.553 6.45e-11 ***
RPDE            0.004059   0.022615   0.179 0.857576
DFA            -0.276898   0.019893 -13.919  < 2e-16 ***
PPE             0.229100   0.033268   6.886 6.75e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9367 on 3508 degrees of freedom
Multiple R-squared:  0.1212,    Adjusted R-squared:  0.1172
F-statistic: 30.24 on 16 and 3508 DF,  p-value: < 2.2e-16
```

This output can be used to determine which variables contribute significantly to the model. We look at the right-most column of the output, which is the p-values. All variables with p-value < 0.05 can be said to have a significant contribution. In this case it is Jitter.Abs., Shimmer, Shimmer.APQ5, Shimmer.APQ11, NHR, HNR, DFA and PPE. Next we want to calculate the MSE for the training and test data. Beginning with the training data, a function called mse() was created which takes the fitted model as a parameter and returns the means of the square of the residuals from the model. The result from this was mse_train = 0.87. To measure the test MSE we first had to make a prediction with the test data based on the model. This was done with the predict() function, which returns a vector of predicted values. To calculate the MSE from this we then calculated the mean of (the predicted value - the actual value)^2. This resulted in mse_test = 0.93.

The next part was to implement some functions, first the loglikelihood function. When creating a model we want to find a vector **theta** with parameters that fit the data well. A reasonable way of thinking about this is that we want to find a **theta** which makes observing **y** as likely as possible. That is what the *maximum likelihood* is about. However, it is often better to work with the logarithm for numerical reasons. Maximizing the log-likelihood or minimizing the negative log-likelihood is equal to maximizing the likelihood function, because log is a monotonically increasing function. Since motor_UPDRS is normally distributed, we can use the following log-likelihood function:

$$\ln p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = -\frac{n}{2}\ln(2\pi\sigma_{\varepsilon}^2) - \frac{1}{2\sigma_{\varepsilon}^2}\sum_{i=1}^{n}\left(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}_i - y_i\right)^2$$

We implemented this in R in the following way:

```
llh <- function(theta, sigma){
  n = length(train[,1])
  x = train[,-1]
  y = train[,1]
  return(- n / 2 * log(2*pi*sigma^2, exp(1)) - (1 / (2*sigma^2)) * sum((theta %*% t(x) - y)^2))
}
```

A vector **theta** and a scalar sigma are input parameters, n is the number of observations in the training data, x is a matrix of the feature variables and y is the target variable vector. The returned function is the same as the one specified above.

Next we implemented a ridge function. The purpose of this function is to penalize complexity and thus avoid overfitting. We implemented the ridge function in the following way:

```
ridge <- function(theta, sigma, lambda){
  loglik = llh(theta, sigma)
  penalty = lambda * sum(theta^2)
  return (penalty - loglik)
}
```

This function returns the negative log-likelihood with an additional ridge penalty. The penalty is given by the hyperparameter lambda multiplied with the sum of theta squared. This helps keeping the theta parameters close to 0. This is good since small parameters are desirable as long as not too much information is lost in the model. There is a trade-off between keeping **theta** small and fitting the training data well.

Then we have the RidgeOpt funtion which is used to find the optimal **theta** with regard to the ridge function. Our implementation is given below:

```
ridgeOpt <- function(lambda){
  return(optim(par=numeric(length(train[1,-1])), fn=ridge, method="BFGS",
               sigma=sigma(fit1), lambda=lambda))
}
```

The function takes lambda as an input parameter and then uses the optim() function to find the optimal **theta**.

The last function to implement was the DF function, which was implemented according to the following definition:

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

Our implementation of the function is:

```
DF <- function(lambda){
  opt = ridgeOpt(lambda)
  sigma_sqr = var(train[,1])
  y = train[,1]
  n = length(train[,1])
  y_hat = numeric(n)
  for (i in 1:n){
    y_hat[i] = opt$par %*% t(train[i,-1])
  }
  return (1 / sigma_sqr * cov(y_hat, y))
}
```

The function takes lambda as input parameter to pass in to the ridgeOpt() function and get the corresponding parameter vector. Sigma squared is the variance of the training data, y is the correct values from the training data, n is the number of rows in the training data and y_hat is a vector of the predicted values for the given parameter vector.

When using the ridgeOpt() function with lambda = 1, 100, 1000 the returned parameter vectors **theta** were the following:

```
> print(ro1$par)
 [1]  0.172925928 -0.168816137 -0.008761781 -0.067751148 -0.007506097  0.533145735 -0.144788327
 [8] -0.150099591 -0.373649472  0.311085590 -0.150581950 -0.184531493 -0.238334048  0.004998807
[15] -0.276040090  0.228528944
> print(ro100$par)
 [1]  0.04710354 -0.12994314  0.01583158 -0.02610922  0.01590333  0.03755943  0.02145389
 [8] -0.07634392 -0.10051651  0.22455993 -0.07638435 -0.14203382 -0.18294343  0.02774281
[15] -0.24235354  0.21607987
> print(ro1000$par)
 [1]  0.006020365 -0.042161505 -0.003326839 -0.006841769 -0.003319359  0.005608257  0.012932840
 [8] -0.017988550 -0.010265955  0.073743613 -0.017991478 -0.038404615 -0.080388968  0.045516365
[15] -0.128695109  0.107032140
```

One thing to note when looking at this is that the parameter values decrease when lambda increases. This is expected because a larger lambda means a larger penalty for large parameter values. These parameters were then used to make predictions and calculate MSE for these different lambdas for both the train and test data subsets. The results were as follows, where the number in the variable names correspond to the value of lambda:

mse_test	0.929702144083076
mse_test_1	0.929034143105053
mse_test_100	0.926315554698319
mse_test_1000	0.947136708705449
mse_train	0.873147045796779
mse_train_1	0.873277172058579
mse_train_100	0.879049827165216
mse_train_1000	0.914534725431453

If we begin looking at the different MSEs for the training data we can see a pattern that the MSE increases when lambda increases. That is an expected result since the training data is the data used for the model and the larger the lambda, the more we deviate from the fit. The difference in MSE is not that great between lambda = 0, 1, 100 but with lambda = 1000 the difference is notable. When looking at the test data MSE we notice that it is overall higher than that for the training data, because the model is fitted on the training data. However, with increasing lambda, the overfitting decreases and the MSE gets a little better for test with lambda = 1 and 100. With lambda = 1000 the MSE increases a bit, and this means that we have simplified the model too much and lost some information. The conclusion is that lambda = 100

is the best of these options in this case since it gives the least MSE for the test data while not increasing the MSE for the training data significantly.

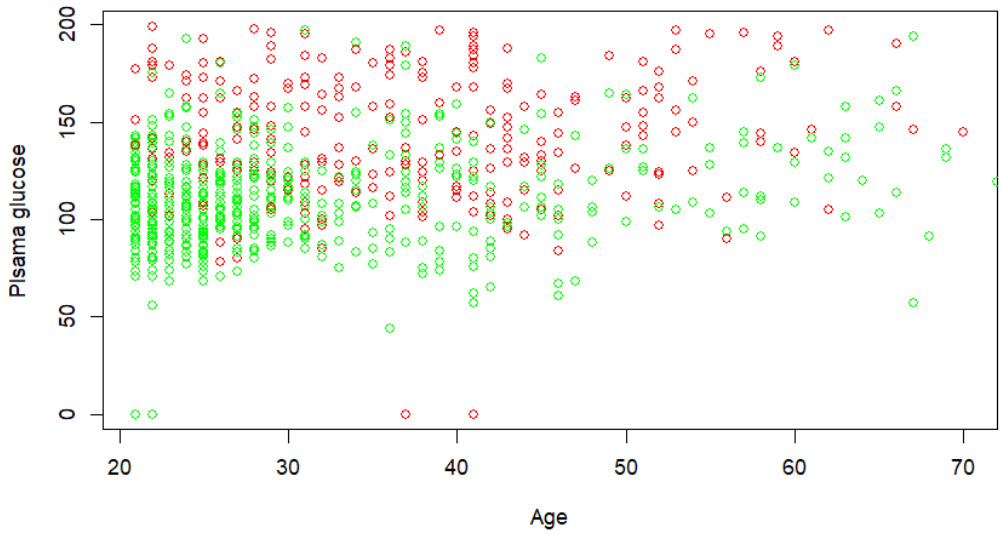
The calculated degrees of freedom of the training data with lambda = 1, 100, 1000 were:

df_1	0.12068379165083
df_100	0.102575378317235
df_1000	0.0564013823447367

The results show that the degrees of freedom decrease when lambda increases.

Assignment 3

When making a scatter plot showing Plasma glucose concentration on Age we get the following



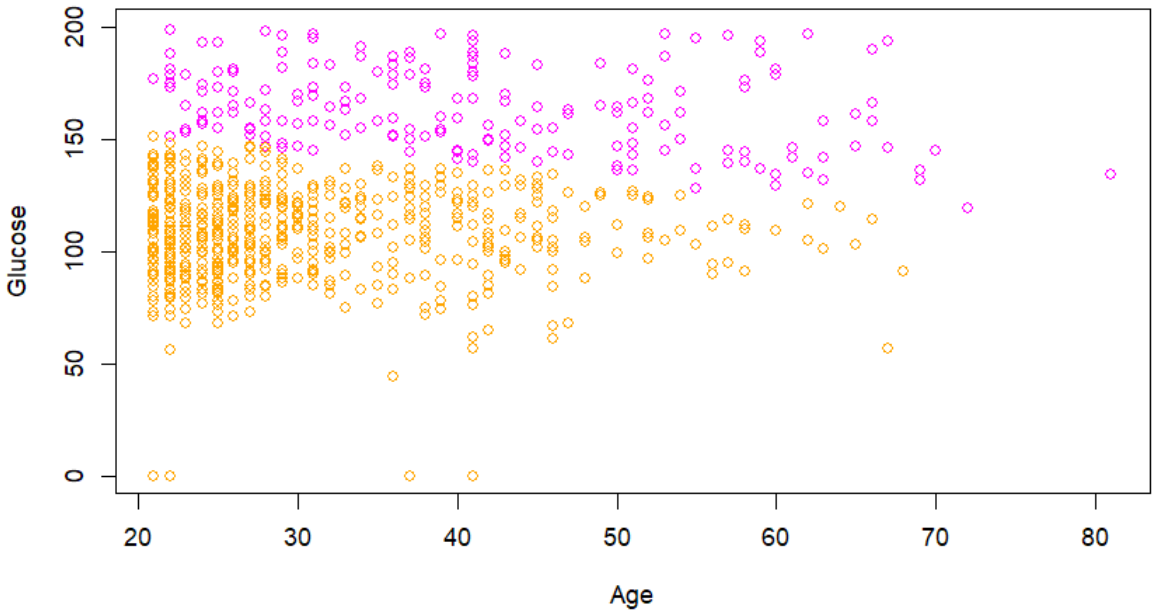
When observing the scatter plot above there is some characteristics that might indicate that one person might have diabetes. When looking at the lower age span, we see that glucose levels of above 130-140 might indicate more towards that the person have diabetes. When looking in the higher ages span it is harder to see that there are any connections between the presences of diabetes and Age & Plasma glucose concentrations. However, it is not obvious that one has or do not have diabetes when looking at the graph, even though that there is indicators.

When constructing a logistic regression model with Diabetes as target and Age & Plasma glucose concentrations as parameters we receive the following probabilistic equation:

```
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -5.897858   0.462450  -12.75  < 2e-16 ***
Age          0.024502   0.007379    3.32 0.000899 ***
Glucose      0.035582   0.003288   10.82  < 2e-16 ***
```

$$-5.897858 + 0.024502 \times Age + 0.035582 \times Glucose$$

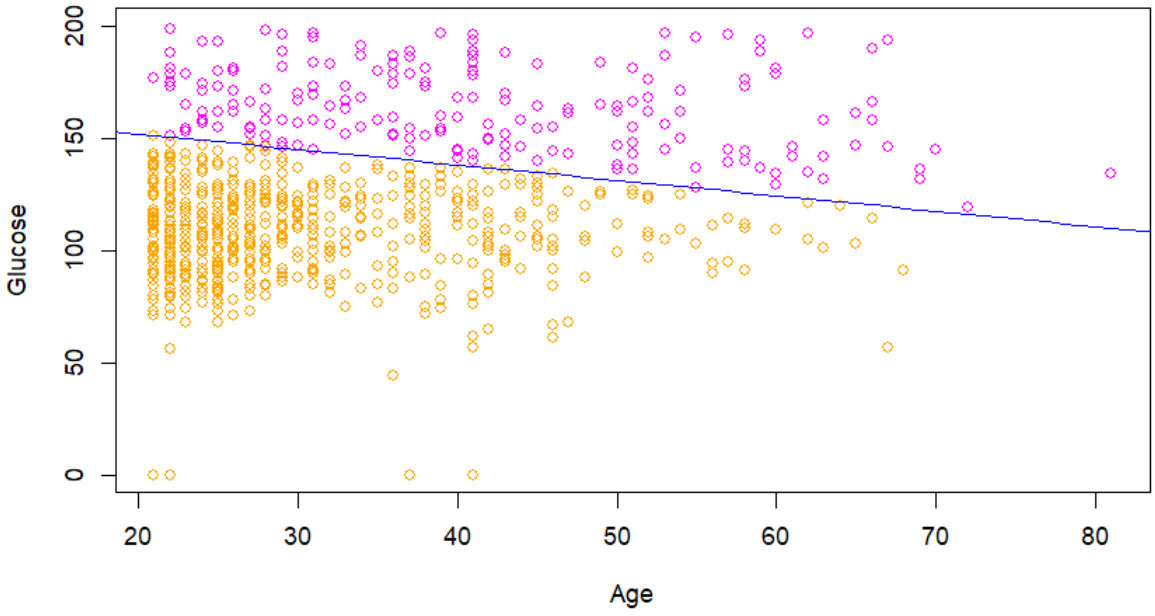
When computing the misclassification error for the model we need to take the average misclassification error by first calculating the misclassification for classifying as Diabetes and then for classifying as not diabetes and taking an average of them. When doing this we get a misclassification error of 0.289067. When plotting the predictions, magenta = predict diabetes, orange = predict not diabetes, we get the following.



When observing the equation, we can extract the following formula, note that we have Age on x-axis and Glucose on y – axis

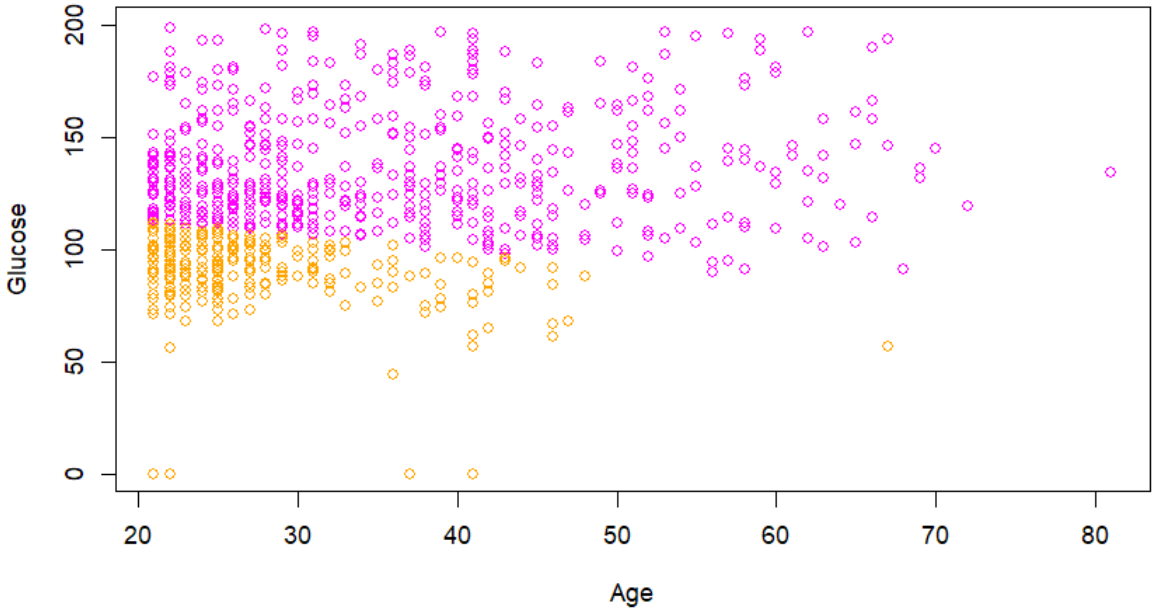
$$Glucose = \frac{5.897858 - 0.024502 \times Age}{0.035582}$$

When plotting this line in the scatterplot we gain the following:

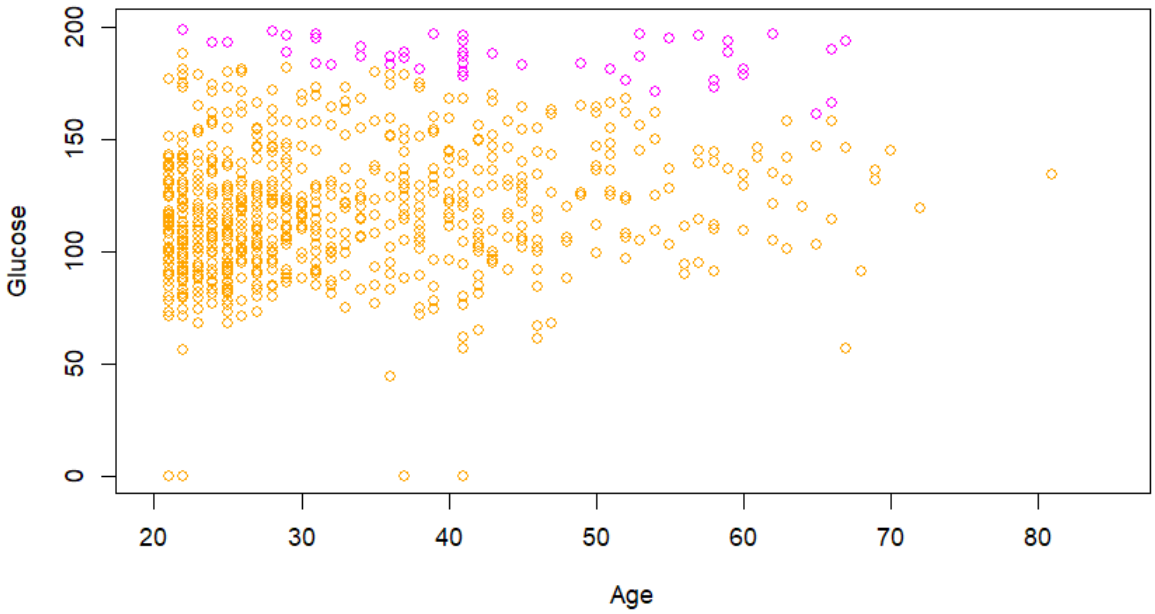


When observing the scatterplot above one see that the data is covered quite well. It matches the decision curve mostly perfect, the only place where it is a bit uncertain is right around the line.

When changing the r value to 0.2 we receive the following graph



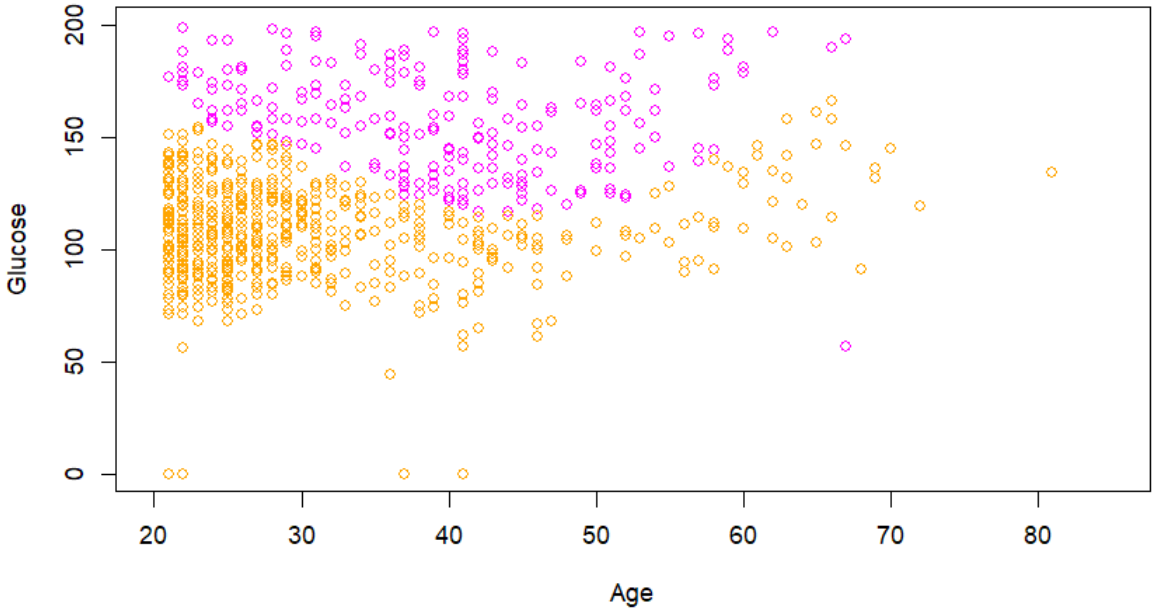
As well as a misclassification error of 0.3074492. With r = 0.8 we gain the following graph



With a misclassification error of 0.2688898

You see that the higher the r value the lower the numbers of cases are classified as diabetes. This is because we set a higher threshold, the same principle in reverse applies when we lower the r value. This is one thing to take into consideration when constructing the model. However, it might be better to classify someone as having diabetes than the reverse, and therefor one might want to have a lower threshold.

When expanding the model with $z_1 = x_1^4, z_2 = x_1^3 x_2 , z_3 = x_1^2 x_2^2, z_4 = x_1 x_2^3, z_5 = x_2^4$ we gain the following graph



With the misclassification error 0.2679288. The quality of this model is a little better than the other (however not by much). However, the shape of the model has changed, we have now gained a more of a convex shape than a straight line. The model does not seem to be overfitting the data, we do get a misclassification error of almost 0.27. The prediction accuracy has improved a bit, but by tweaking r you get a better model, but in this case, there is risk of overfitting.

Appendix 1

```
# Import library for k-NN nearest neighbors
library(kknn)

# Data set
data = read.csv("optdigits.csv")
colnames(data)[65] = "digit"

# Number of observations
n = dim(data)[1]

# Dividing the data into different samples
set.seed(12345)
learning_sample     = sample(1:n, floor(n*0.5))
not_learning_sample = setdiff(1:n, learning_sample)
# Should not need to set a seed again but we followed the example in the lecture
# slides
set.seed(12345)
valid_sample        = sample(not_learning_sample, floor(n*0.25))
test_sample         = setdiff(not_learning_sample, valid_sample)

learning = data[learning_sample,]
valid    = data[   valid_sample,]
test     = data[    test_sample,]

# Creating a model where the digit depends on the 64 values
# The model uses the learning set to train and test the model
# The model is based on the 30 nearest neighbors
model1 <- kknn(as.factor(digit) ~ .,
               train  = learning,
               test   = learning,
               k      = 30,
```

```
            kernel = "rectangular")

# Prints the confusion matrix showing the difference between the predicted and
# the actual values
print(table(learning[,65], model1$fitted.values))

# Misclassification function that returns the rate of incorrect predictions
misclass=function(X1,X2){
  n = length(X1)
  return(1 - sum(diag(table(X1,X2))) / n)
}

# Prints the misclassification rate of the model
print(misclass(learning[,65], model1$fitted.values))

# Another model that used the test data to test the model instead
model2 <- kknn(as.factor(digit) ~ .,
            train  =learning,
            test   = test,
            k      = 30,
            kernel = "rectangular")

# Prints the confusion matrix showing the difference between the predicted and
# the actual values
print(table(test[,65], model2$fitted.values))

# Prints the misclassification rate of the model
print(misclass(test[,65], model2$fitted.values))

# The pictures of eights in the learning data
actual_eight    = learning[learning$digit==8,]
# The pictures of eights in ascending order based on the probability that the
# digit was eight
```

```
predicted_eight = order(model1$prob[learning$digit==8,9])

# The 8x8 matrix of values of the 3 pictures that were the most difficult to
# predict and the 2 easiest to predict
hardest1 = matrix(unlist(actual_eight[predicted_eight[1],   -65]), 8, 8, TRUE)
hardest2 = matrix(unlist(actual_eight[predicted_eight[2],   -65]), 8, 8, TRUE)
hardest3 = matrix(unlist(actual_eight[predicted_eight[3],   -65]), 8, 8, TRUE)
easiest1 = matrix(unlist(actual_eight[predicted_eight[204], -65]), 8, 8, TRUE)
easiest2 = matrix(unlist(actual_eight[predicted_eight[205], -65]), 8, 8, TRUE)

# Shows the heatmap of the 8x8 matrices
heatmap(hardest1, Colv = NA, Rowv = NA)
heatmap(hardest2, Colv = NA, Rowv = NA)
heatmap(hardest3, Colv = NA, Rowv = NA)
heatmap(easiest1, Colv = NA, Rowv = NA)
heatmap(easiest2, Colv = NA, Rowv = NA)

# Vectors containing the misclassification rate each of the models
misclass_learn = numeric(30)
misclass_valid = numeric(30)

# Creates one model that uses learning to test and one that used valid to test
# for each k value between 1 and 30
for (k in 1:30) {
  model_learn      <- kknn(as.factor(digit) ~ .,
                           train  = learning,
                           test   = learning,
                           k      = k,
                           kernel = "rectangular")
  model_valid      <- kknn(as.factor(digit) ~ .,
                           train  = learning,
                           test   = valid,
                           k      = k,
```

```
                        kernel = "rectangular")
  misclass_learn[k] = misclass(learning[,65], model_learn$fitted.values)
  misclass_valid[k] = misclass(   valid[,65], model_valid$fitted.values)
}

# Plots the two vectors of misclassification rates against eachother
plot(  1:30, misclass_valid, col = "blue", ylim = c(-0.001,0.06),
       xlab = "K-value", ylab = "Misclassification rate")
points(1:30, misclass_learn, col="red")

# 6 was the k-value with the lowest misclassification rate
# This is the model for k=6 and test as test data
model_test <- kknn(as.factor(digit) ~ .,
                   train  = learning,
                   test   = test,
                   k      = 7,
                   kernel = "rectangular")
# Misclassification rate for model_test
misclass_test = misclass(test[, 65], model_test$fitted.values)

# Add the misclassification rate to the plot
points(7, misclass_test, col="Brown")

# Use cross entropy to find the error of the models
cross_entropy = double(30)
# Loop over the k-values
for (k in 1:30){
  model_validation <- kknn(as.factor(digit) ~ .,
                           train  = learning,
                           test   = valid,
                           k      = k,
                           kernel = "rectangular")
  cross_entropy_sum = 0.000000000
```

```
  # Loop over the possible values
  for (i in 0:9) {
    # Add up all cross entropy values for every value
    cross_entropy_sum = cross_entropy_sum -
      sum(log10(model_validation$prob[valid$digit==i, i+1] + 1e-15))
  }
  cross_entropy[k] = cross_entropy_sum
}

# Plots the cross entropy values for the different models
plot(1:30, cross_entropy, col = "blue",
     xlab = "K-value", ylab = "Cross-entropy error")
```

Appendix 2

```
#Part 1

data = read.csv('parkinsons.csv')

#Scale the target data and the features and convert to data frame
data_scaled = scale(data[,5:22])
data_scaled = data_scaled[,-2]
data_scaled = data.frame(data_scaled)

#Divides scaled data into training and testing sets
set.seed(12345)
n = dim(data_scaled)[1]
id1 = sample(1:n, floor(n*0.6))
train = data_scaled[id1,]
test = data_scaled[-id1,]

#Part 2

#Function to calculate MSE
mse <- function(x){
  mean(x$residuals^2)
}

#Create linear regression model and calculate training MSE
fit1 = lm(motor_UPDRS~., data=train)
summary(fit1)
mse_train = mse(fit1)

#Make predictions with test data and calculate test MSE
fit2 = predict(fit1, test[-1])
mse_test = mean((fit2-test[,1])^2)
```

```
#Part 3

#Loglikelyhood function
llh <- function(theta, sigma){
  n = length(train[,1])
  x = train[,-1]
  y = train[,1]
  return(- n / 2 * log(2*pi*sigma^2, exp(1)) - (1 / (2*sigma^2)) * sum((theta %*% t(x) - y)^2))
}

#Ridge function
ridge <- function(theta, sigma, lambda){
  loglik = llh(theta, sigma)
  penalty = lambda * sum(theta^2)
  return (penalty - loglik)
}

#Optimal Ridge function
ridgeOpt <- function(lambda){
  return(optim(par=numeric(length(train[1,-1])), fn=ridge, method="BFGS",
               sigma=sigma(fit1), lambda=lambda))
}

#Degrees of freedom function
DF <- function(lambda){
  opt = ridgeOpt(lambda)
  sigma_sqr = var(train[,1])
  y = train[,1]
  n = length(train[,1])
  y_hat = numeric(n)
  for (i in 1:n){
    y_hat[i] = opt$par %*% t(train[i,-1])
```

```
  }
  return (1 / sigma_sqr * cov(y_hat, y))
}

#Part 4

ro1 = ridgeOpt(1)
ro100 = ridgeOpt(100)
ro1000 = ridgeOpt(1000)

#Function for calculating MSE for different parameters and subsets of data
mse2 <- function(theta, subset) {
  x = subset[,-1]
  y = subset[,1]
  return (mean((theta %*% t(x) - y)^2))
}

mse_train_1 = mse2(ro1$par, train)
mse_train_100 = mse2(ro100$par, train)
mse_train_1000 = mse2(ro1000$par, train)
mse_test_1 = mse2(ro1$par, test)
mse_test_100 = mse2(ro100$par, test)
mse_test_1000 = mse2(ro1000$par, test)

#Calculated degrees of freedom for the training data with different lambdas
df_1 = DF(1)
df_100 = DF(100)
df_1000 = DF(1000)
```

Appendix 3

```
# Data set
data = read.csv('pima-indians-diabetes.csv')

# Make a new variable, containing the data if patient have diabetes
Have_diabetes = data[data$X1 == 1,]
# Make a new variable, containing the data if patient does not diabetes
Do_not_have_diabetes = data[data$X1 == 0,]

#Plotting the people who have diabetes as red dots and green points as not having diabetes
plot(Have_diabetes$X50, Have_diabetes$X148, xlab = "Age",
    ylab = "Plsama glucose",  col="red", xlim = c(18,85))
points(Do_not_have_diabetes$X50, Do_not_have_diabetes$X148, col = "Green")

refined_data = data[,c(2,8:9)] #Refine data
names(refined_data)[names(refined_data) == 'X1'] <- 'Diabetes'
names(refined_data)[names(refined_data) == 'X50'] <- 'Age'
names(refined_data)[names(refined_data) == 'X148'] <- 'Glucose'


logi = glm(Diabetes ~ Age+Glucose,refined_data,family="binomial") #Make a model
predictions = predict(logi, type = "response") #Make predictions
max(predictions)
r = 0.5

#Take out the people who the model classifies as having diabetes
predict_diabetes = data[predictions>r,]
#Take out the people who the model classifies as not having diabetes
predict_not_diabetes = data[predictions<= r,]

#Constructing a functions for missclasification
```

```
missclass_average=function(X,X1){
  n=length(X)
  m = length(X1)
  return(((1 - sum(X) / n) + sum(X1) / m)/2)
}
#Calculate the missclassification error
print(missclass_average(predict_diabetes[,9], predict_not_diabetes[,9]))


#Plotting the predictions of people having diabetes as magenta and orange for predections of people not having diabetes
plot(predict_diabetes$X50, predict_diabetes$X148, xlab = "Age", ylab = "Glucose",
     col = "magenta", ylim = c(0,200))
points(predict_not_diabetes$X50, predict_not_diabetes$X148, col = "orange")
#Making the desicion line
x = 0:85
y = -(logi$coefficients["(Intercept)"] + logi$coefficients["Age"]*x)/logi$coefficients["Glucose"]
lines(x, y, col = "blue")



#Testing for r = 0.8
r = 0.8
#Separating predictions
predict_diabetes = data[predictions>r,]
predict_not_diabetes = data[predictions<= r,]
#Calculating the missclassification error
print(missclass_average(predict_diabetes[,9], predict_not_diabetes[,9]))
#Plotting
plot(predict_diabetes$X50, predict_diabetes$X148, xlab = "Age", ylab = "Glucose", col = "magenta", ylim = c(0,200), xlim = c(20,85))
points(predict_not_diabetes$X50, predict_not_diabetes$X148, col = "orange")


#Testing for r = 0.2
r = 0.2
```

```
#Separating predictions

predict_diabetes = data[predictions>r,]

predict_not_diabetes = data[predictions<= r,]

#Calculating the missclassification error

print(missclass_average(predict_diabetes[,9], predict_not_diabetes[,9]))

#Plotting

plot(predict_diabetes$X50, predict_diabetes$X148, xlab = "Age", ylab = "Glucose", col = "magenta",
ylim = c(0,200))

points(predict_not_diabetes$X50, predict_not_diabetes$X148, col = "orange")



x1 = refined_data$Glucose

x2 = refined_data$Age

# Make new features

z1 = x1^4

z2 = x1^3*x2

z3 = x1^2*x2^2

z4 = x1*x2^3

z5 = x2^4

#making new model using all the features

logi = glm(Diabetes ~ Age+Glucose+z1+z2+z3+z4+z5,refined_data,family="binomial")

#making predictions

predictions = predict(logi, type = "response")

r = 0.5

#Separating predictions

predict_diabetes = data[predictions>r,]

predict_not_diabetes = data[predictions<= r,]

#Calculating the missclassification error

print(missclass_average(predict_diabetes[,9], predict_not_diabetes[,9]))

#Plotting

plot(predict_diabetes$X50, predict_diabetes$X148, xlab = "Age", ylab = "Glucose", col = "magenta",
ylim = c(0,200), xlim = c(20,85))

points(predict_not_diabetes$X50, predict_not_diabetes$X148, col = "orange")
```