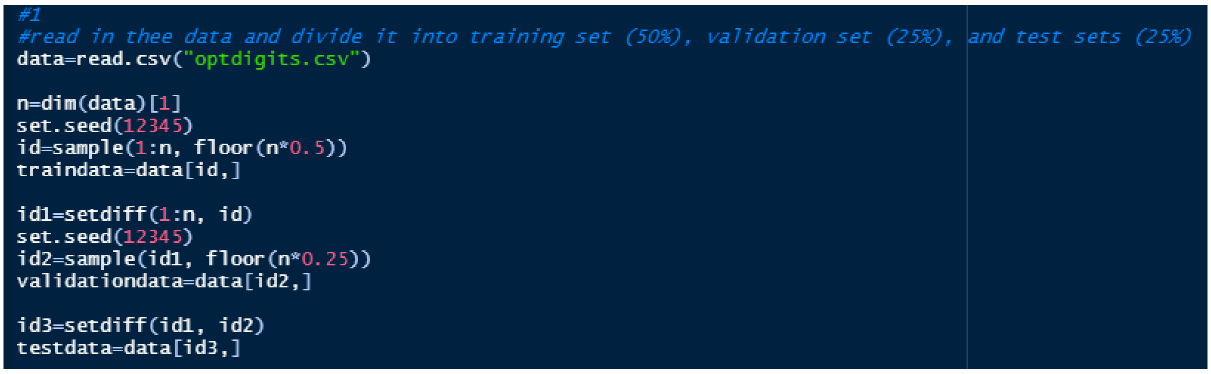
Assignment 1

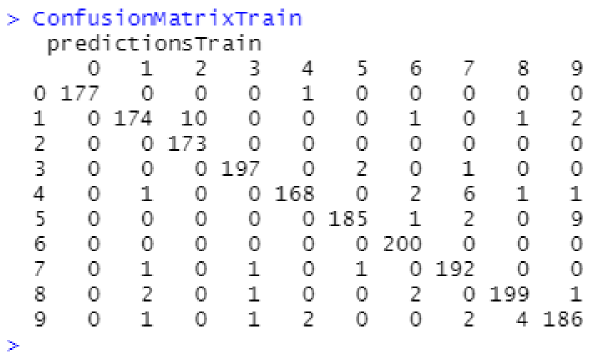
1. Read and split data



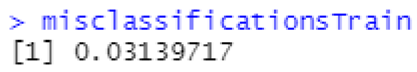
```
#1
#read in thee data and divide it into training set (50%), validation set (25%), and test sets (25%)
data=read.csv("optdigits.csv")

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

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

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

2. Fitting 30-NN classifier model using training set, and evaluating the model using training set and validation set respectively

Confusion matrix and misclassification error for training data to test the model



```
> confusionMatrixTrain
   predictionsTrain
      0   1   2   3   4   5   6   7   8   9
  0 177   0   0   0   1   0   0   0   0   0
  1   0 174  10   0   0   0   1   0   1   2
  2   0   0 173   0   0   0   0   0   0   0
  3   0   0   0 197   0   2   0   1   0   0
  4   0   1   0   0 168   0   2   6   1   1
  5   0   0   0   0   0 185   1   2   0   9
  6   0   0   0   0   0   0 200   0   0   0
  7   0   1   0   1   0   1   0 192   0   0
  8   0   2   0   1   0   0   2   0 199   1
  9   0   1   0   1   2   0   0   2   4 186
>
```

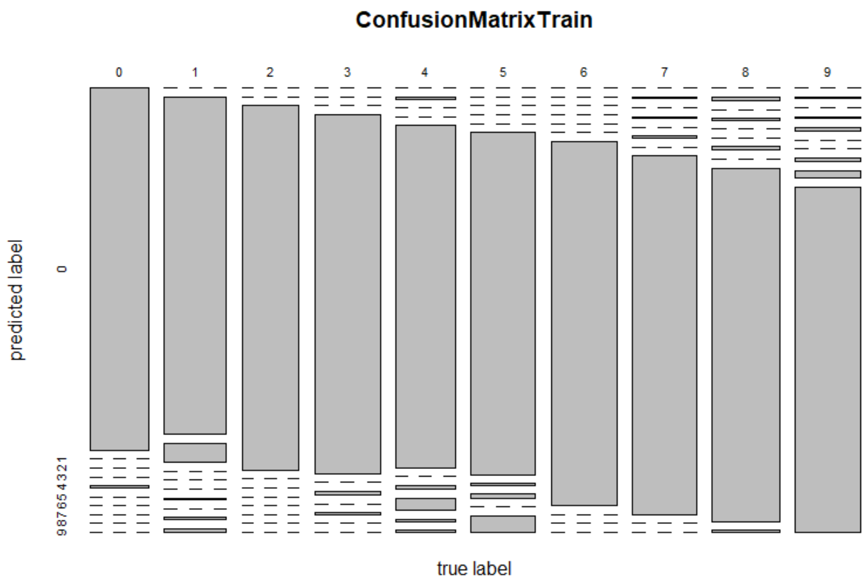


```
> misclassificationsTrain
[1] 0.03139717
```

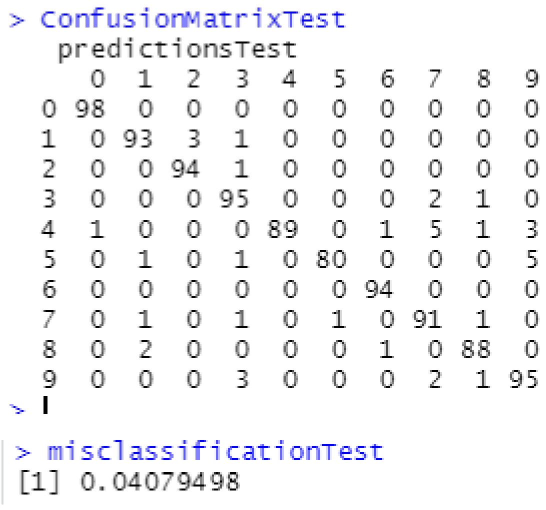
The misclassification error using our model trained on the training set, to predict the labels of the same training set for evaluation, is 3,14%. This means that the model has a high accuracy, at least when making predictions on data it has already seen. If we look at the confusion matrix, the x-axis corresponds to the true labels, and the y-axis to the predicted labels, thus the elements on the diagonal correspond to correct classifications.

We can see that for some numbers, e.g. 0, 2, 3, 6 and 7 the model performs great and barely has any wrong predictions (0-3). Whereas for some other numbers the model has a slightly worse prediction quality, e.g. when predicting 1’s, which the model often wrongfully classified as 2’s (10 cases). The same can be seen for 4’s, which the model often wrongfully predicted as 7’s, and 5’s, which the model often wrongfully predicted as 9’s.

A visual representation of the models prediction quality can also be seen in the below plot o the confusion matrix.



Confusion matrix and misclassification error for test data to test the model

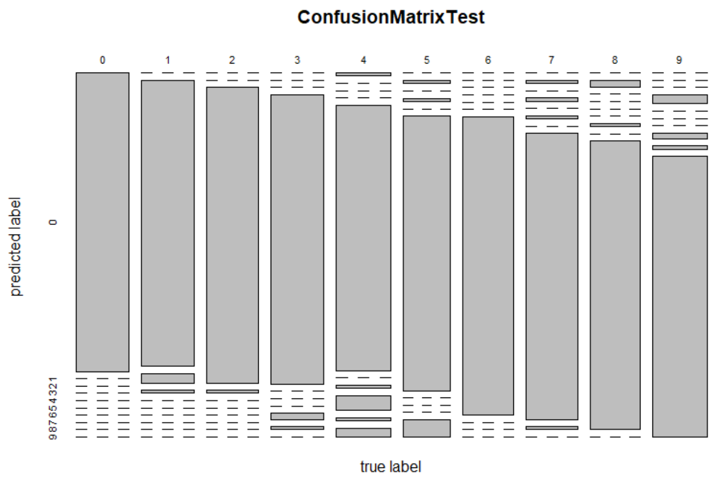


```
> ConfusionMatrixTest
   predictionsTest
     0  1  2  3  4  5  6  7  8  9
  0 98  0  0  0  0  0  0  0  0  0
  1  0 93  3  1  0  0  0  0  0  0
  2  0  0 94  1  0  0  0  0  0  0
  3  0  0  0 95  0  0  0  2  1  0
  4  1  0  0  0 89  0  1  5  1  3
  5  0  1  0  1  0 80  0  0  0  5
  6  0  0  0  0  0  0 94  0  0  0
  7  0  1  0  1  0  1  0 91  1  0
  8  0  2  0  0  0  0  1  0 88  0
  9  0  0  0  3  0  0  0  2  1 95
> 
> misclassificationTest
[1] 0.04079498
```

The misclassification error using our model trained on the training set, to predict the labels of the test set for evaluation, is 4,08%. This means that the model has a high accuracy also when using new data, but of course the misclassification error is larger here than when making predictions on the same data the model was trained on. But an increase from 3,14% accuracy to 4,08% going from training to test data should not be anything to worry about regarding the models accuracy, since we can still expect it to perform classifications on new data only with a 4,08% misclassification rate.

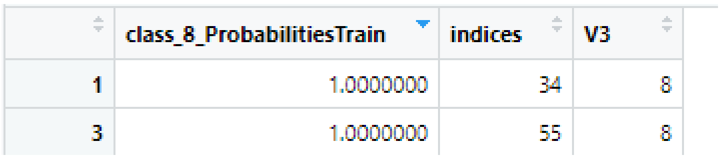
We can see that for some numbers, e.g. 0, 2, 3, and 6 the model performs great and barely has any wrong predictions (0-3). Whereas for some other numbers the model has a slightly worse prediction quality, e.g. when predicting 4's, which the model often wrongfully classified as 7's (5 cases), and when predicting 5's, which the model often wrongfully classified as 9's (5 cases).

A visual representation of the models prediction quality can also be seen in the below plot of the confusion matrix.

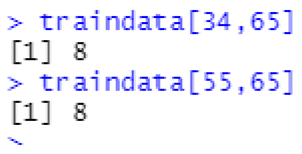


3.

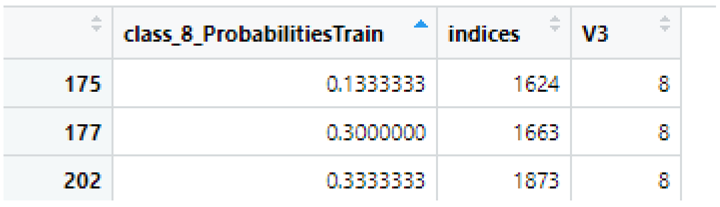
The indices and probabilities for the easiest and hardest 8’s are saved in two matrices. Also testing that the number on the corresponding indices are in fact 8’s.



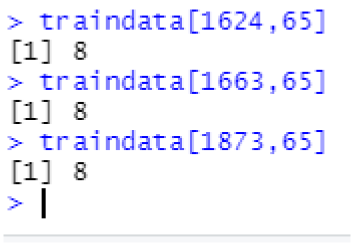
	class_8_ProbabilitiesTrain	indices	V3
1	1.0000000	34	8
3	1.0000000	55	8



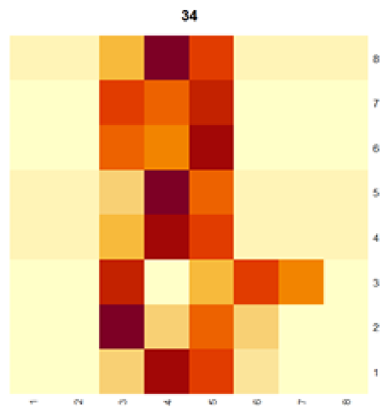
```
> traindata[34,65]
[1] 8
> traindata[55,65]
[1] 8
>
```

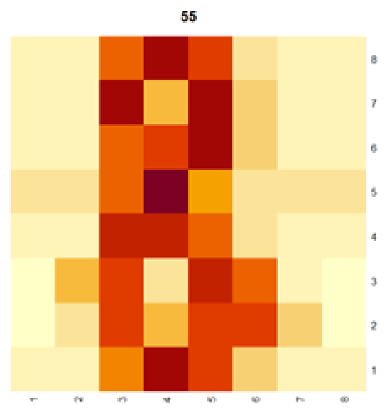


	class_8_ProbabilitiesTrain	indices	V3
175	0.1333333	1624	8
177	0.3000000	1663	8
202	0.3333333	1873	8

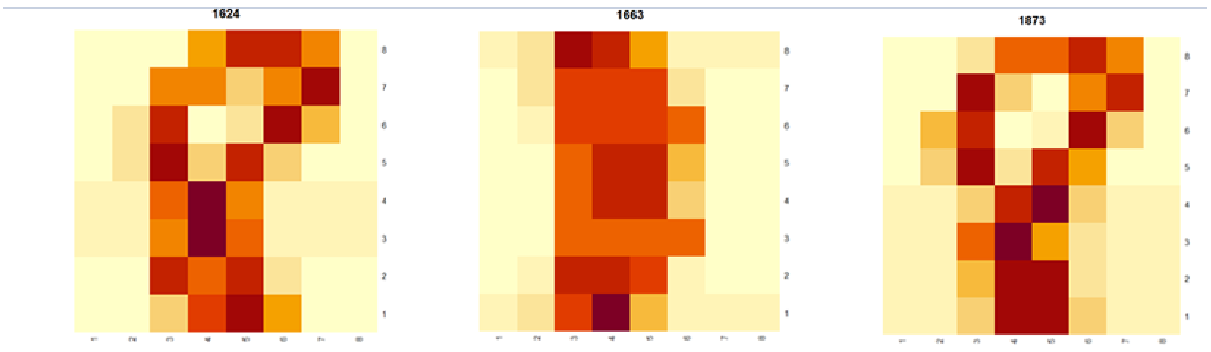


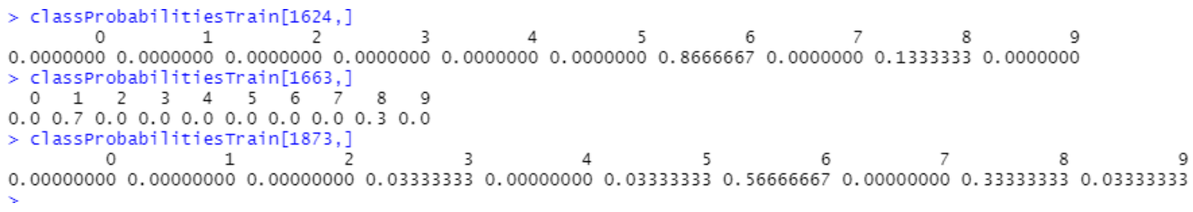
```
> traindata[1624,65]
[1] 8
> traindata[1663,65]
[1] 8
> traindata[1873,65]
[1] 8
> 
```





The 2 easiest 8's to classify were the 8's on row 34 and 55. The probability to classify the number as an 8 was 100% in both cases. In the above case, you can quite clearly see that the heatmap resembles an 8.



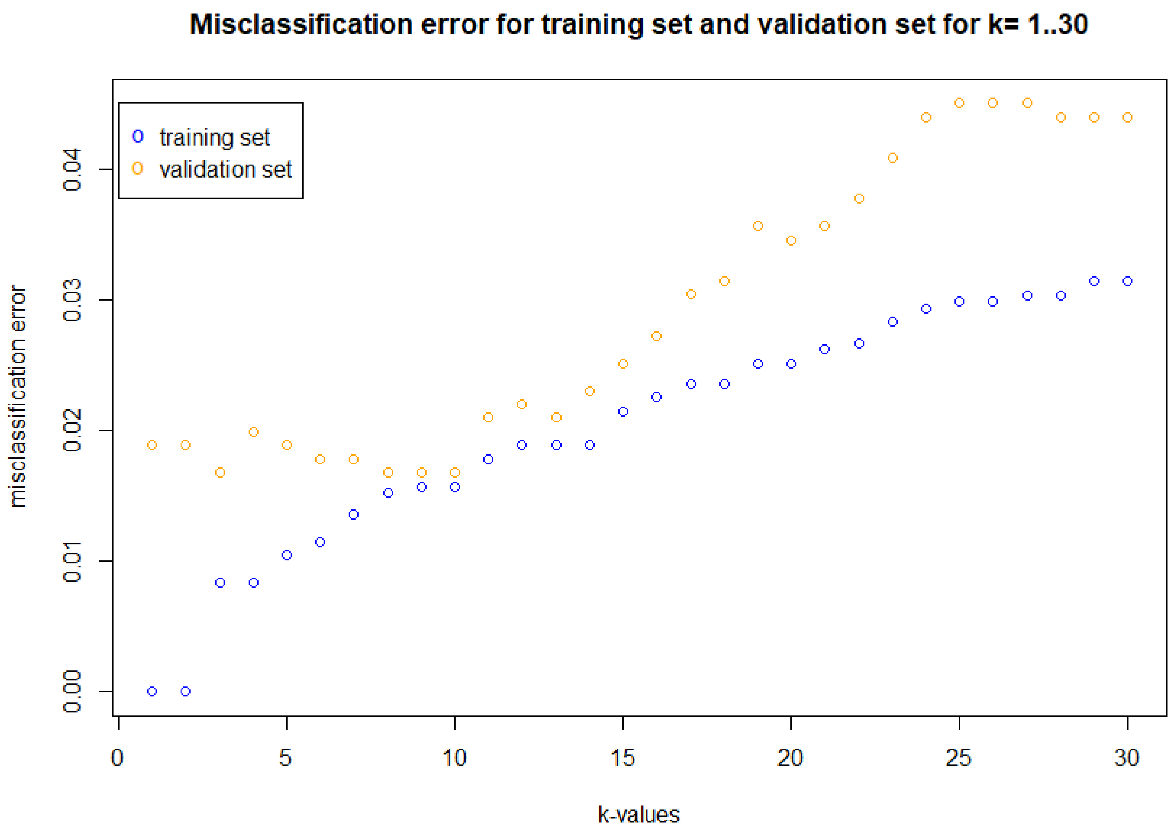


```
> classProbabilitiesTrain[1624,]
        0         1         2         3         4         5         6         7         8         9 
0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.8666667 0.0000000 0.1333333 0.0000000 
> classProbabilitiesTrain[1663,]
  0   1   2   3   4   5   6   7   8   9 
0.0 0.7 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.0 
> classProbabilitiesTrain[1873,]
         0          1          2          3          4          5          6          7          8          9 
0.00000000 0.00000000 0.00000000 0.03333333 0.00000000 0.03333333 0.56666667 0.00000000 0.33333333 0.03333333 
> 
```

The 3 hardest 8's to classify were the 8's on row 1624, 1663 and 1873. The probability to classify the number as an 8 was 13,3%, 30% and 33% respectively. The predicted number of above 3 cases was 6, 1 and 6. Although in my opinion the above images do not clearly resemble a 6,1 or 6 you can see why the model made the predictions it made, and I would say that none of them clearly resemble an 8. For the first and third case, the 8 lacks the bottom hole which together with the top hole is a distinct feature for the 8, and for the second case the 8 is lacking any hole at all, thus making it appear as a thick 1.

^Missuppfattning av frågan. Se koden I lab1 assignment1 för korrekt.

4.

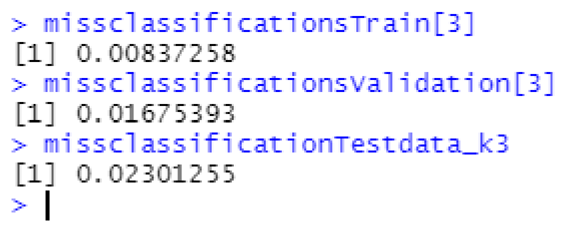


In the figure we can see the various misclassification errors for k=1,..,30 for training data (orange), and validation data (blue).

As we have a low k-value, we have a high variance and low bias thus an over-fitting model, which indicates that the model is complex. Thus as we increase k, we move towards a lower variance and a higher bias and our models complexity decreases, and if we decrease the complexity too much we could get an underfitting model. Meaning that the model complexity decreases as we increase our k.

```
> which.min(missclassificationsvalidation)
[1] 3
```

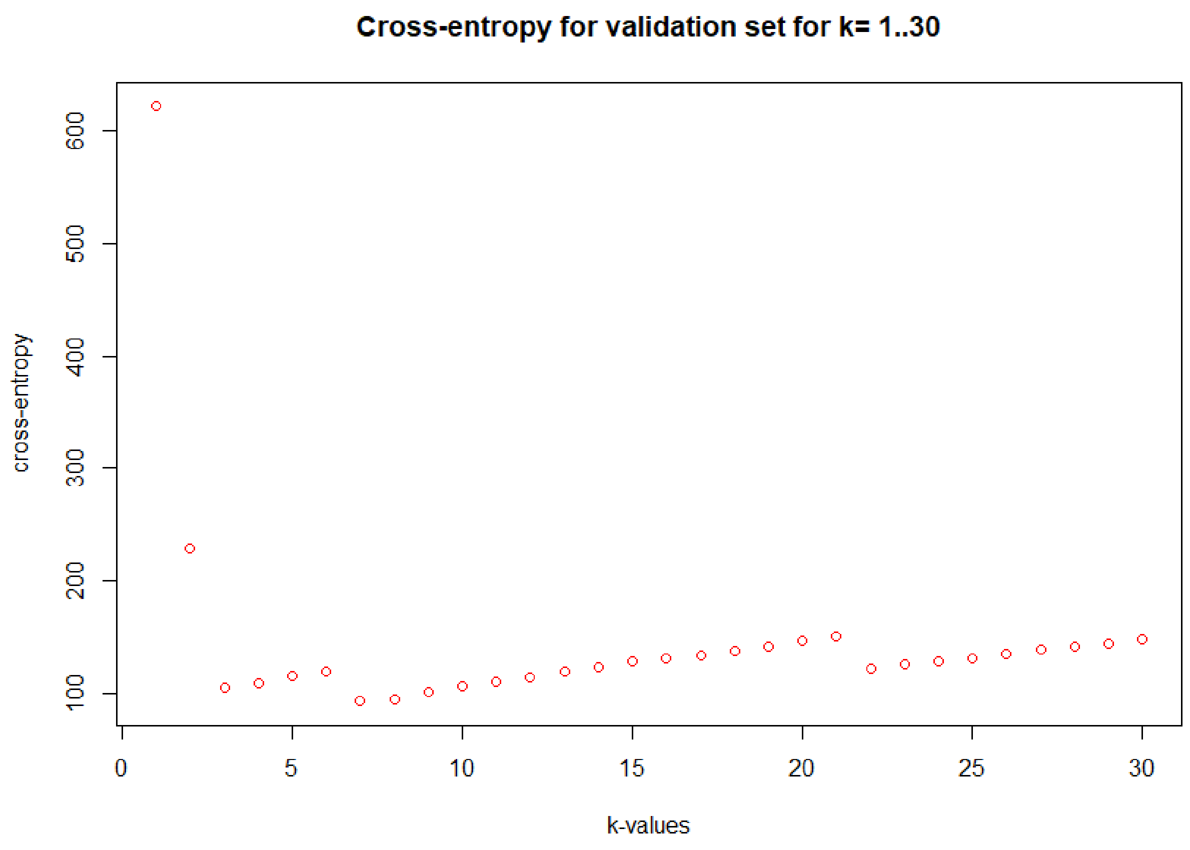
By looking at the plot, we see that for the training set we have no misclassification at all for k=1 and k=2 (since we are overfitting), and for k=3 there is a big increase in the misclassification rate and from here the misclassification steadily increases as k increases. For the validation data, which is more interesting to look at since this is new data which has not been used for training to predict for our model, we can see that the optimal k for the validation set is k=3. Comparing to the training sets misclassifications, assuming that k=1 and k=2 are not optimal due to overfitting, k=3 is also one of the better values for the training set. Thus I would argue that k=3 is the optimal value for k here.

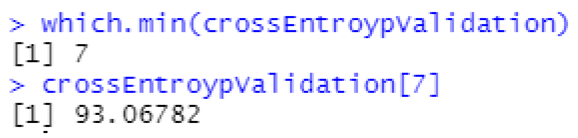


```
> missclassificationsTrain[3]
[1] 0.00837258
> missclassificationsValidation[3]
[1] 0.01675393
> missclassificationTestdata_k3
[1] 0.02301255
> 
```

Using our optimal k, k=3, we can derive the following misclassification rates. For the test data the misclassification rate is 2,30%, compared to 1,67% for the validation data and 0,84% for the training data. It is most relevant to look at test set here, since validation data can also to some extent overfit similarly to training data since we selected optimal hyperparameter k by using the validation data, i.e. the validation data is also no lunger “new” data to the model, whereas the test data has never been used for training nor hyperparameter selection. But we can see that for the test data the misclassification error is 2,30%, which I would consider to be good in this problem and its domain (as we saw with the heatmaps, some numbers would be hard to label even for a human), and thus I would say that the model quality is good.

5.





```
> which.min(crossEntroypValidation)
[1] 7
> crossEntroypValidation[7]
[1] 93.06782
```

Looking at the plot, we can also see that the cross-entropy is minimized for k=7, compared to k=3 which we previously found to be optimal using misclassification rate. K=3 is however still has a reasonable value here, with a cross-entropy pf 105,60.

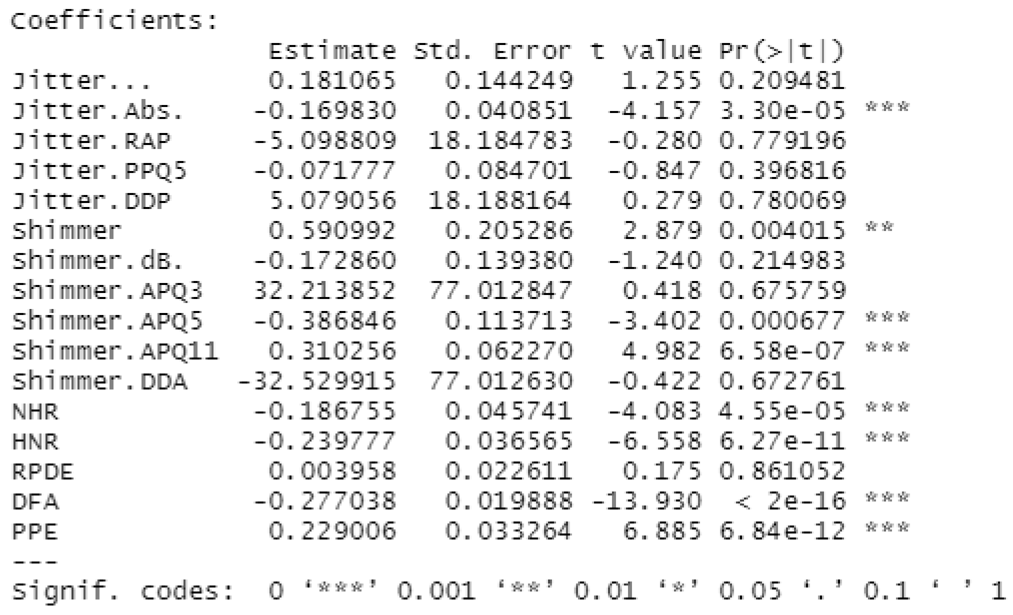
Since we have a multinomial distribution of 10 classes here, the cross-entropy is a more reasonable choice for error function rather than the misclassification error for this problem, due to the fact that the cross-entropy looks at probability that each prediction is indeed the correct class label, i.e. it is also taking into account how probable a certain prediction is. Compared to the misclassification rate, which simply takes into account whether the prediction is correct or not. Thus cross-entropy will give us a more accurate estimation of the error, especially as our data set grows.

Assignment 2

1. See code for scaling and data split

2.

Significant variables:



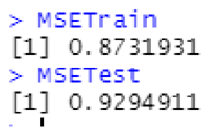
```
Coefficients:
               Estimate Std. Error t value Pr(>|t|)    
Jitter...      0.181065   0.144249   1.255 0.209481    
Jitter.Abs.   -0.169830   0.040851  -4.157 3.30e-05 ***
Jitter.RAP    -5.098809  18.184783  -0.280 0.779196    
Jitter.PPQ5   -0.071777   0.084701  -0.847 0.396816    
Jitter.DDP     5.079056  18.188164   0.279 0.780069    
Shimmer        0.590992   0.205286   2.879 0.004015 ** 
Shimmer.dB.   -0.172860   0.139380  -1.240 0.214983    
Shimmer.APQ3  32.213852  77.012847   0.418 0.675759    
Shimmer.APQ5  -0.386846   0.113713  -3.402 0.000677 ***
Shimmer.APQ11  0.310256   0.062270   4.982 6.58e-07 ***
Shimmer.DDA  -32.529915  77.012630  -0.422 0.672761    
NHR           -0.186755   0.045741  -4.083 4.55e-05 ***
HNR           -0.239777   0.036565  -6.558 6.27e-11 ***
RPDE           0.003958   0.022611   0.175 0.861052    
DFA           -0.277038   0.019888 -13.930  < 2e-16 ***
PPE            0.229006   0.033264   6.885 6.84e-12 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

We look at the p-values, which tells us how significant a variable is to the model. Smaller is more significant to the model, which is what we want. The standard threshold is normally p < 0.05, and otherwise a feature is not significant enough to keep in the model.

In our model: Jitter... Jitter.RAP Jitter.PPQ5 Jitter.DPP Shimmer.dB. Shimmer.APQ3 Shimmer.DDA RPDE do not contriubte significantly (since p-value > 0.05).

In the model summary, a significant feature with a p-value < 0.05 is denoted by *** and **. So significant variables: Jitter.Abs. ; Shimmer ; Shimmer.APQ5 ; Shimmer.APQ11 ; NHR ; HNR ; DFA ; PPE

MSE for training data and test data:

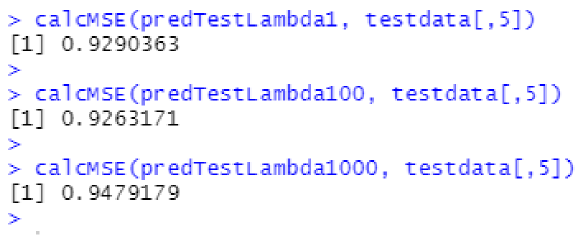


```
> MSETrain
[1] 0.8731931
> MSETest
[1] 0.9294911
```

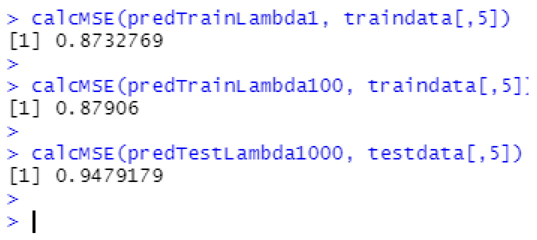
As expected, MSE for test data is slightly higher than for training data (due to the model overfitting for training data).

3. See code for the functions.

4.

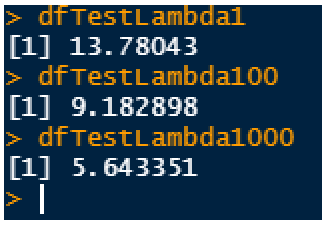


```
> calcMSE(predTestLambda1, testdata[,5])
[1] 0.9290363
>
> calcMSE(predTestLambda100, testdata[,5])
[1] 0.9263171
>
> calcMSE(predTestLambda1000, testdata[,5])
[1] 0.9479179
>
```



```
> calcMSE(predTrainLambda1, traindata[,5])
[1] 0.8732769
>
> calcMSE(predTrainLambda100, traindata[,5]]
[1] 0.87906
>
> calcMSE(predTestLambda1000, testdata[,5])
[1] 0.9479179
>
> |
```

lambda=100 seems to be best if we look at the test data. lambda=1 seems to be best if we look at training data, however we care more about the test data. So lambda = 100 seems to be the optimal value of the parameter which gives a moderate amount of punishment of large coefficients when optimizing our theta. Although, the difference between MSE for lambda = 1 and 100 is minimal when looking at the test data.

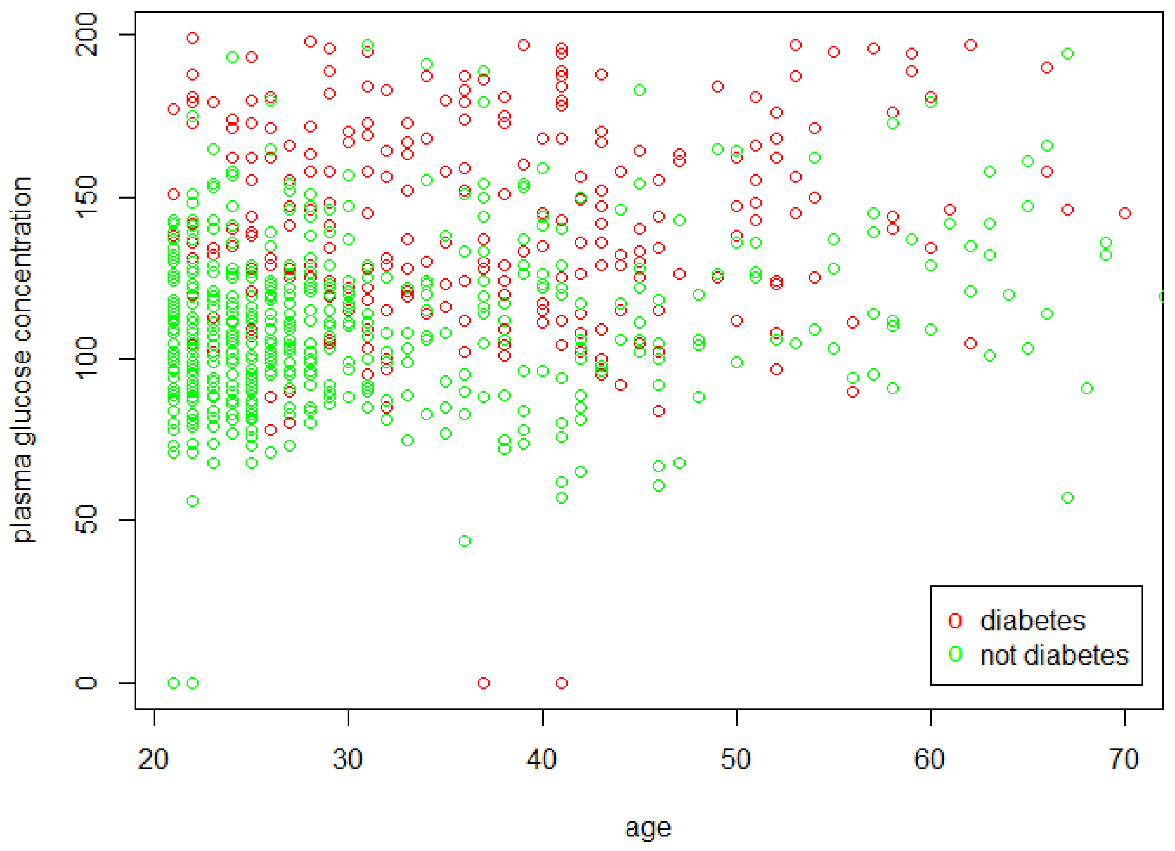


```
> dfTestLambda1
[1] 13.78043
> dfTestLambda100
[1] 9.182898
> dfTestLambda1000
[1] 5.643351
> |
```

The degrees of freedom are decreasing as lambda increases since this makes the model less complex, and the degrees of freedom tell us how well the model is able to follow changes in y for various samples, and a large value of degrees of freedom tells us the model is able to do this very well, i.e. it is very flexible and complex and probably overfitting. For lambda = 0, we have df = p, i.e. the model is complex compared to when we let lambda grow which then decreases the df (since for lambda = 0 we don't regularize at all, so this is as complex the model gets). But as lambda grows, we regularize more, and thus we get lower df, meaning a less complex model.

In statistics, the number of degrees of freedom is the number of values in the final calculation of a statistic that are free to vary. The number of independent ways by which a dynamic system can move, without violating any constraint imposed on it, is called number of degrees of freedom

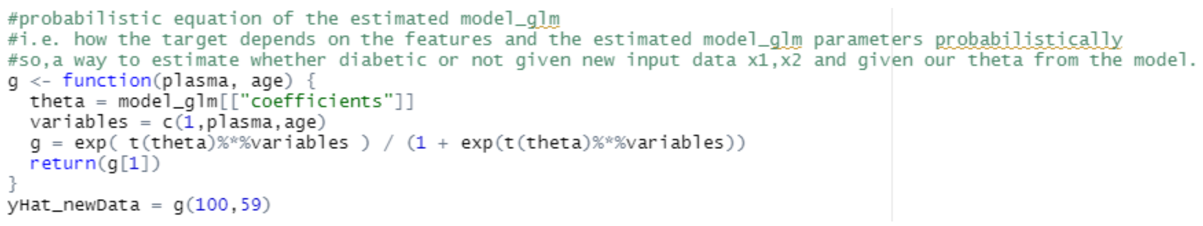
Assignment 3



1. From the plot, it is not that apparent that diabetes is easily classified by a logistic regression model. However, a lot of the data belong to diabetes cases are quite clustered around low age and a glucose concentration of around 100. It also appears as if a high plasma glucose concentration is corelated with having diabetes. So you could in the plot see that there may exist some decision boundary which would be able to separate and thus classify the data quite okay. However the data points are quite scattered outside of the obvious green cluster and the red cluster at the top of the plot, thus it might be hard to achieve a good logistic regression model and we will most likely have a quite large misclassification rate.
   But more concretely, it is hard for the linear logistic regression to separate overlapping datapoints with hard linear boundaries, thus we can draw the conclusion that the logistic regression model will not be able to perform well here.

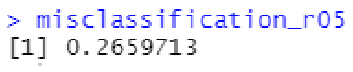
2.

Probabilistic equation of the estimated model

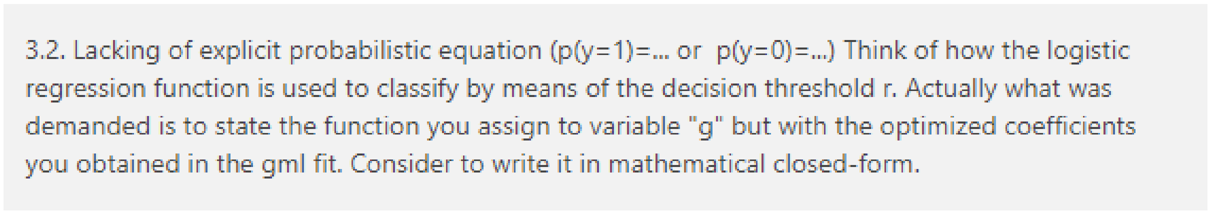


```r
#probabilistic equation of the estimated model_glm
#i.e. how the target depends on the features and the estimated model_glm parameters probabilistically
#so,a way to estimate whether diabetic or not given new input data x1,x2 and given our theta from the model.
g <- function(plasma, age) {
  theta = model_glm[["coefficients"]]
  variables = c(1,plasma,age)
  g = exp( t(theta)%*%variables ) / (1 + exp(t(theta)%*%variables))
  return(g[1])
}
yHat_newData = g(100,59)
```

Misclassification for r = 0,5



```
> misclassification_r05
[1] 0.2659713
```



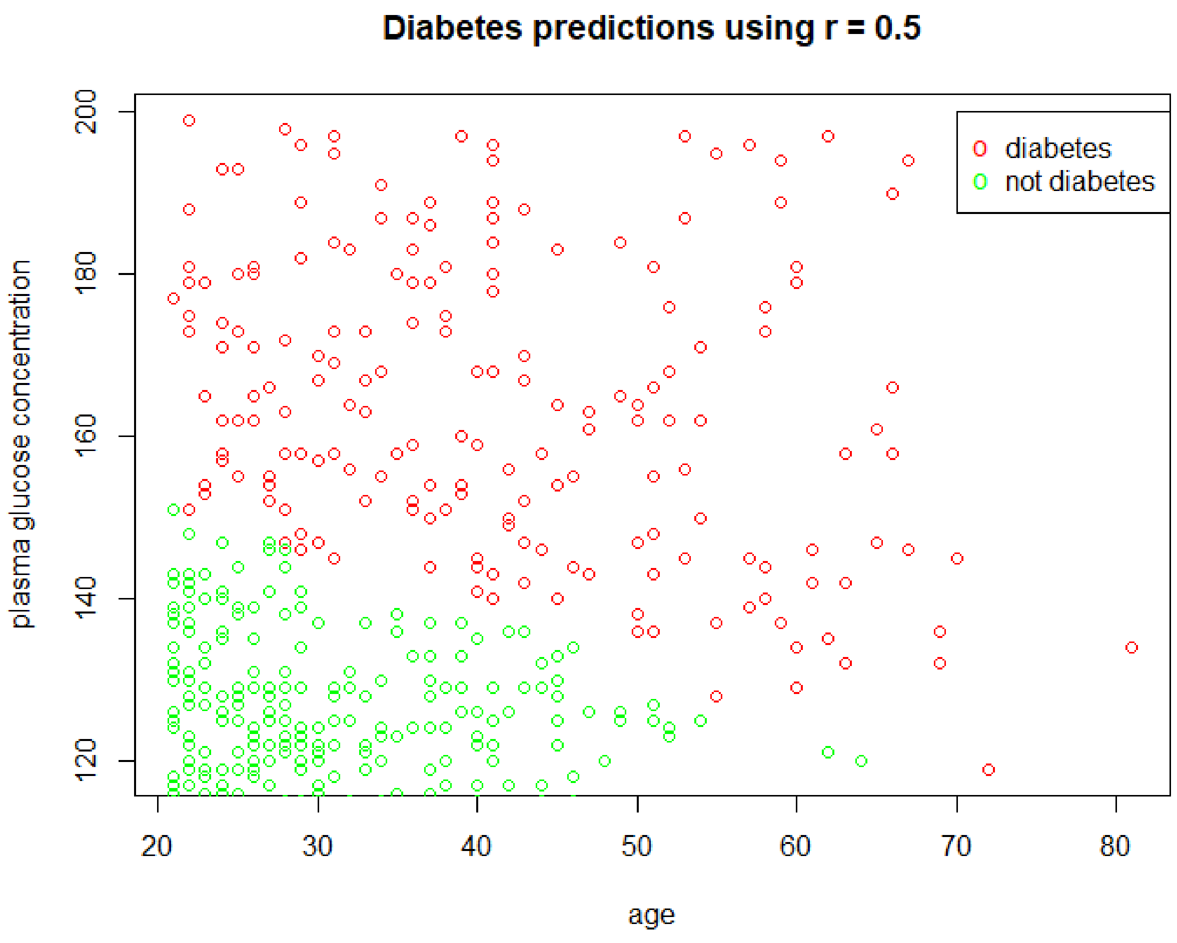
3.2. Lacking of explicit probabilistic equation (p(y=1)=... or p(y=0)=...) Think of how the logistic regression function is used to classify by means of the decision threshold r. Actually what was demanded is to state the function you assign to variable "g" but with the optimized coefficients you obtained in the gml fit. Consider to write it in mathematical closed-form.

C = {C_1 = 1, C_0 = 0}, where C_1 is diabetes, C_0 is not diabetes.

P(y = C_1|x) = g(**x**) = 1 / $(1 + e^{-\theta^T x})$, where θ^T = (-5.897858, 0.035582, 0.024502) and x = (1, plasmaGlucose, age)

So: g(x) = 1 / (1 + exp(-(-5. 897858 + 0.035582x1 + 0.024502x2)))

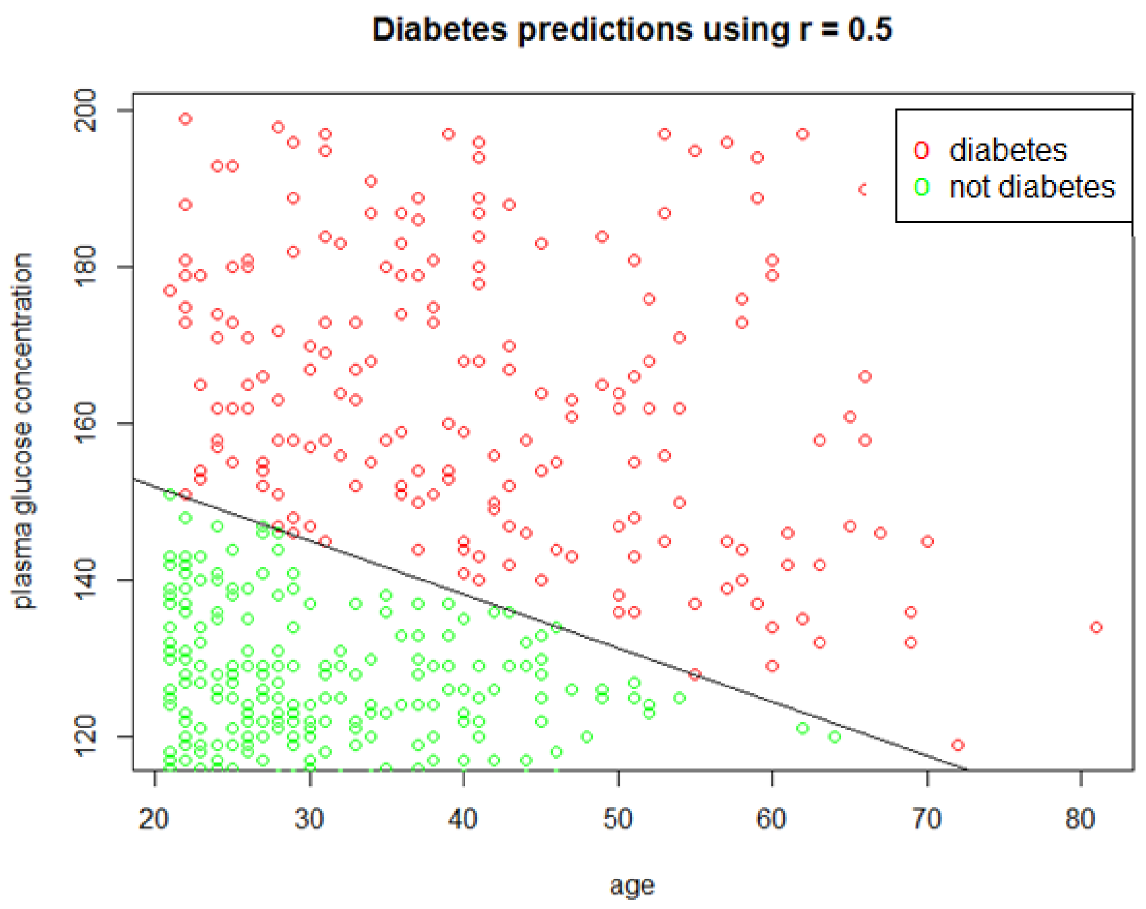
And p(y=C|x) = {g(x) if C = 1, 1-g(x) if C = 0}, i.e. p(y=1|x) = g(x), p(y=0|x) = 1 – g(x)



The misclassification error is quite high for our model with above 2 variables and r= 0.5 at 26,6%. Thus our model does not classify the data very well.

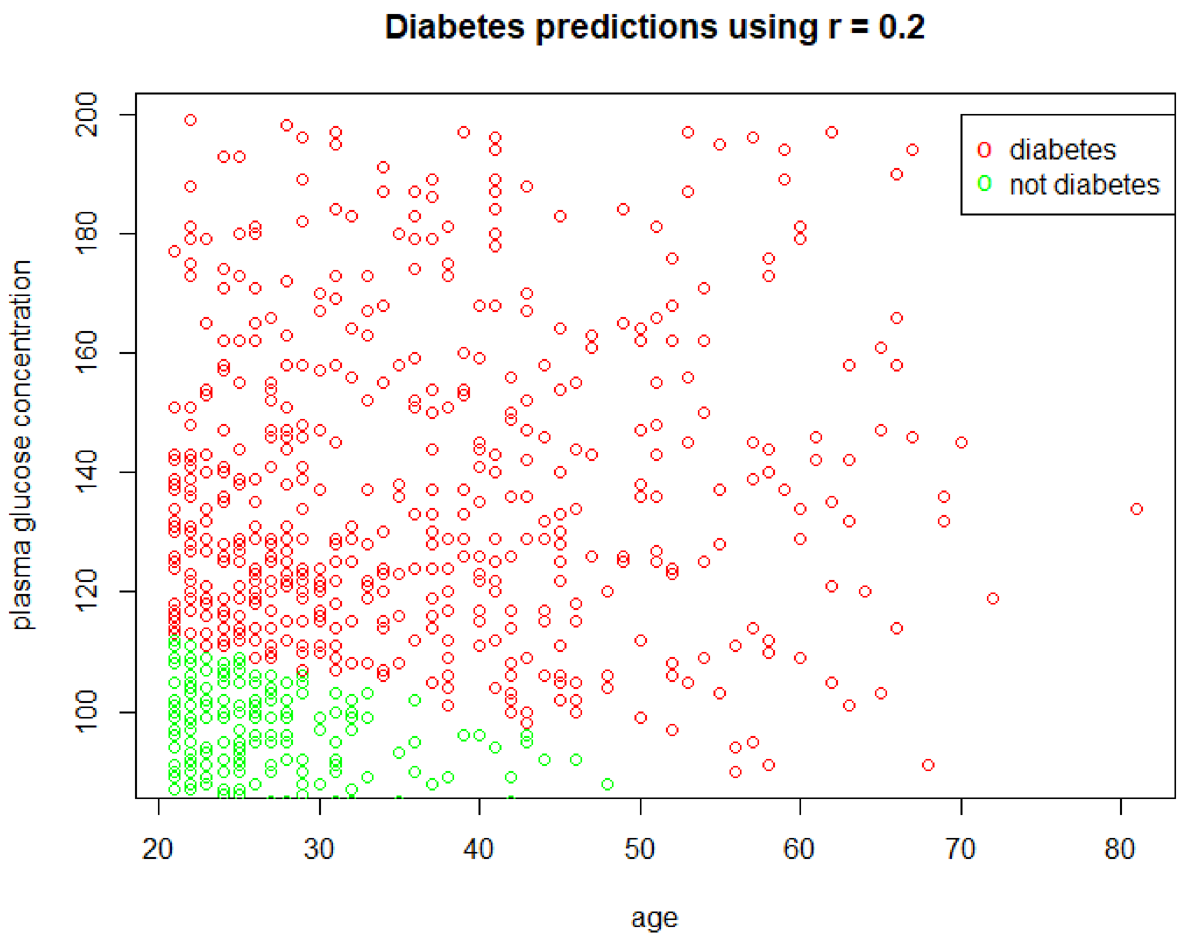
3.

```
#To calculate the decision boundary: t(theta) %*% X = 0
#This gives us: theta0 + theta1x1 + theta2x2 = 0 <=> x2 = -theta0/theta2 -theta1/theta2 * x1
slope = - coef(model_glm)[3] / coef(model_glm)[2]
intercept = - coef(model_glm)[1] / coef(model_glm)[]
abline(intercept, slope)
```



Adding this curve i.e. the decision boundary to the previous plot where we used threshold value r=0.5 we get the above plot. This does not mean that we are catching the data distribution pattern well, since the decision boundary depends on the r-value and thus what prediction we make, in the above plot it will always separate the 2 classes perfectly since we are predicting based on our decision boundary. But comparing to our underlying data distribution, we are not catching this pattern well, which we can see if we look where the decision boundary would go in the original plot. We need a model that is more flexible to be able to catch the underlying distribution pattern well.

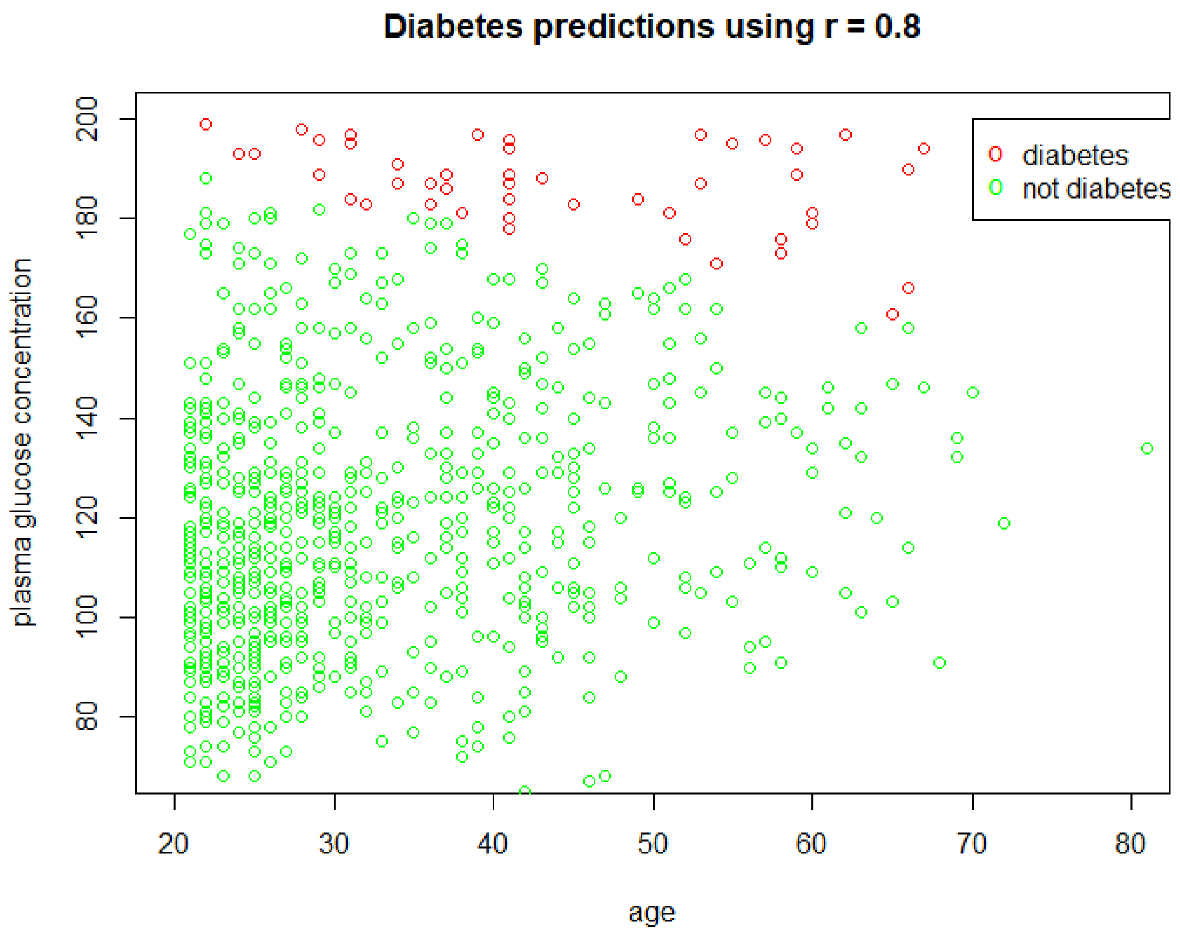
4.



```
> misclassification_r02
[1] 0.3741851
```

As we can see, when we change r from 0.5 to 0.2, more data points will be classified as having diabetes, since now when a person has a probability of diabetes > 0.2 the person will be classified as having diabetes, thus resulting in more red points in the plot. We get a classifier that more correctly classifies predicted diabetes, but also more wrongfully classifies no predicted diabetes.

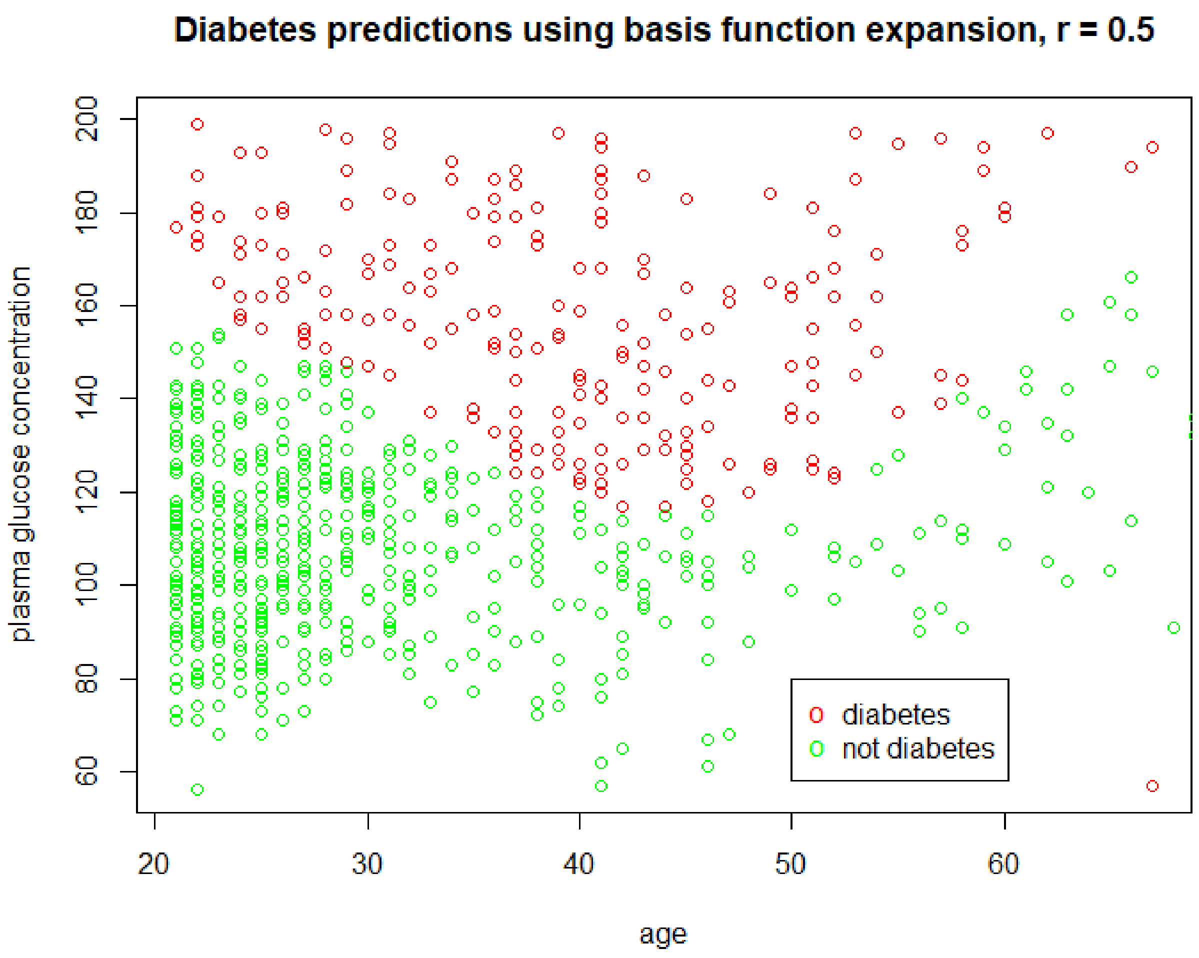
Furthermore, we can also see that the misclassification rate has worsened as compared to using r=0,5, i.e. using a threshold value of r=0,2 makes our model perform worse.



```
> misclassification_r08
[1] 0.3142112
```

As we can see, when we change r from 0.5 to 0.8, less data points will be classified as having diabetes, since now when a person has a probability of diabetes > 0.8 the person will be classified as having diabetes, thus resulting in fewer red points in the plot. We get a classifier that more correctly classifies no predicted diabetes, but also more wrongfully classifies predicted diabetes. I.e. we have a chance of classifying more sick patients as healthy, which we of course do not want to do. In this case, this is far worse than what happens when we classify more patients as sick rather than healthy as we did with decision boundary 0.2.

Furthermore, we can also see that the misclassification rate has worsened as compared to using r=0,5, i.e. using a threshold value of r=0,8 makes our model perform worse.



```
> misclassification_basis_f_exp
[1] 0.2464146
```

Upon doing the basis function expansion trick, our model now consists of 7 features rather than 2, whereas the expansion is supposed to help us better classify our data using our two variables plasma glucose and age, i.e. all 7 features still depend on the 2 input variables plasma glucose and age. Upon fitting a new glm model and making predictions with a threshold of r=0,5, we get the above plot and a misclassification of 24,6%.

Although the model using the basis function expansion trick performs slightly better then previous model, it still has a high misclassification rate of 24,6% for r=0,5, compared to 26,6% for r=0,5 with of the previous model. However, it is an improvement at the cost of increased complexity, and by introducing nonlinear terms like this we can improve our model. We should further on evaluate this model on unseen testing data to see how it performs, since there is a risk that this increased complexity causes the model to overfit.

As can be seen in the plot, the basis expansion trick has made the shape of the decision boundary to go from being a line (linear) to non-linear (looking like it behaves as a positive quadratic equation, but it should be a fourth degree curve, which we can see by the one outlier at age=65, plasma=50 which is classified as diabetes). This makes sense, since when we introduce the new variables, we also introduce x1 and x2 terms to the power of 2,3 and 4. Looking at the original data distribution, this new decision boundary does indeed seem to capture the underlying distribution slightly better, which we concluded above from the misclassification rates.

To calculate it we could use the formula:

t(theta)X = 0

This gives us: t0 + t1x1 + t2x2 + t3z1 +t4z2 + t5z3 + t6z4 + t7z5 = 0 <=>

t0 + t1x1 + t2x2 + t3x1^4 + t4x1^3x2 + t5x1^2x2^2+t6x1x2^3+t7x2^4 = 0

Appendix Code Assignment 1

```
#1
#read in thee data and divide it into training set (50%), validation set
(25%), and test sets (25%)
data=read.csv("optdigits.csv")

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

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

id3=setdiff(id1, id2)
testdata=data[id3,]

#2
#fitting 30-NN classifier using the *training* set for training, and using
the same *training* set for testing the models accuracy
kknnModelTrain <- kknn(formula=as.factor(traindata[,65]) ~ . ,
train=traindata, test=traindata, k=30, kernel="rectangular")
#the predicted classification from using the model to predict the classes
of the training set
predictionsTrain <- kknnModelTrain[["fitted.values"]]

#Confusion matrix, original labels on the x-axis, predicted labels on the
y-axis
ConfusionMatrixTrain = table(traindata[,65], predictionsTrain)
#plot(ConfusionMatrixTrain, xlab="true label", ylab="predicted label")
misclassificationsTrain = ( 1-
sum(diag(ConfusionMatrixTrain))/length(traindata[,65]))


#fitting 30-NN classifier using the *training* set for training, and using
the *test* set for testing the models accuracy
kknn_testdata <- kknn(formula=as.factor(traindata[,65]) ~ . ,
train=traindata, test=testdata, k=30, kernel="rectangular")
#the predicted classification from using the model to predict the classes
of the test set
predictionsTest <- kknn_testdata[["fitted.values"]]

ConfusionMatrixTest = table(testdata[,65], predictionsTest)
#plot(ConfusionMatrixTest, xlab="true label", ylab="predicted label")
```

```r
misclassificationTest = ( 1-
sum(diag(ConfusionMatrixTest))/length(testdata[,65]))


#3
#extract class probabilities for each row, and extract the class
probability for only class 8
classProbabilitiesTrain = kknnModelTrain[["prob"]]
class_8_ProbabilitiesTrain = classProbabilitiesTrain[,9]

#for indexing our data
indices = c(1:nrow(traindata))

#combine class probabilities for class 8, with index, and with actual label
class_8_probTrain_Indexed = cbind(class_8_ProbabilitiesTrain, indices,
traindata[,65])
#remove all rows where the the actual label is not 8,
#resulting in a matrix with probability + index of row in training data
class_8_probTrain_Indexed_only8s = subset(class_8_probTrain_Indexed,
class_8_probTrain_Indexed[,3]==8)

#we can now easily check the class_8_probTrain_Indexed_only8s-matrix and
see that
#the 2 easiest cases to classify are the numbers on row 34 and 55 of the
training data
#and the 3 hardest are row 1624, 1663 and 1873 in the training data


#for plotting heatmap for a certain row of our data
heatmapPlot <- function(row) {
  bitmapdata = as.numeric(traindata[row,1:64])
  bitmapMatrix = matrix(bitmapdata, nrow=8, byrow=TRUE)
  heatmap(bitmapMatrix, Rowv = NA, Colv = NA, main = row)
}

heatmapPlot(34)
heatmapPlot(55)
heatmapPlot(1624)
heatmapPlot(1663)
heatmapPlot(1873)


#4
#function for calculating the misclassification for a data set for
k=1...30. parameter data is data to test against,
#i.e. can be traindata, validationdata, or testdata
calc_Missclassification_k_1_to_30 <- function(data) {
  missclassifications = numeric(30)
  for (i in 1:30) {
    kknn_model <- kknn(formula=as.factor(traindata[,65])~.,
train=traindata, test=data, k=i, kernel="rectangular")
    predictions <- kknn_model[["fitted.values"]]
    confusionMatrix = table(data[,65], predictions)
    missclass = ( 1-sum(diag(confusionMatrix))/length(data[,65]))
    missclassifications[i] = missclass
  }
  return(missclassifications)
}

kvalues = c(1:30)
missclassificationsTrain = calc_Missclassification_k_1_to_30(traindata)
```

```r
plot(kvalues, missclassificationsTrain, col="blue", ylim=c(0,0.045), main =
"Misclassification error for training set and validation set for k= 1..30",
     xlab="k-values", ylab="misclassification error" )

missclassificationsValidation =
calc_Missclassification_k_1_to_30(validationdata)
points(kvalues, missclassificationsValidation, col="orange")

legend(0,0.045,legend=c("training set","validation set"),
col=c("blue","orange"),pch=c("o","o"))


#### not part of the assignment ####
#below 2 lines just for testing if our assumtion of k=3 is optimal is
reasonable by comparing with test data aswell.
missclassificationsTest = calc_Missclassification_k_1_to_30(testdata)
points(kvalues, missclassificationsTest, col="green")
#### not part of the assignment ####


#missclassifcation for test data when k=3.
kknn_testdata <- kknn(formula=as.factor(traindata[,65])~.,
train=traindata, test=testdata, k=3, kernel="rectangular")
predictionsTest <- kknn_testdata[["fitted.values"]]
confusionMatrixTest = table(testdata[,65], predictionsTest)
missclassificationTestdata_k3 = ( 1-
sum(diag(confusionMatrixTest))/length(testdata[,65]))


#5
#calculate cross entropy for k=1..30
crossEntroypValidation = numeric(30)
#loop over k=1,...,30
for (j in 1:30) {
  #set crossEntropy to 0 and fit new model using current k value, and
extract the class probabilities
  crossEntropy = 0
  kknn_validationdata <- kknn(formula=as.factor(traindata[,65])~.,
train=traindata, test=validationdata, k=j, kernel="rectangular")
  classProbabilitiesValidation = kknn_validationdata[["prob"]]

  #for each row in our validation set, and for each class in our data, i.e.
number 0 to 9
  for (i in 1:nrow(validationdata)) {
    for (m in 0:9) {
      #if the actual class label of validationdata[i] == m
      #then calculate probability that row i of validation data belongs to
class m
      if (validationdata[i,65] == m) {
        probability_datai_classm =
as.numeric(classProbabilitiesValidation[i,m+1])
        log_prob = log(probability_datai_classm+1e-15)
        crossEntropy = crossEntropy + log_prob
      }
    }
  }
  #since we generally for computational purposes minimize the log
likelihood
  crossEntroypValidation[j] = -crossEntropy
}
```

```
plot(kvalues,crossEntroypValidation, type="b", col="red", main = "Cross-
entropy for validation set for k= 1..30",
     xlab="k-values", ylab="cross-entropy" )
```

Appendix Code Assignment 2

```
#uppg 2

#1,
data=read.csv("parkinsons.csv")
data = as.data.frame(scale(data))
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.6))
traindata=data[id,]
testdata=data[-id,]

#2,
result = lm(motor_UPDRS ~. -age -sex -subject. -test_time -motor_UPDRS -
total_UPDRS -1, data = as.data.frame(traindata)) #linjärregression som ger
parametrar framför features

summary_result <- summary(result)

fit_train = predict(result)
fit_test = predict(result, testdata)

MSE_train = mean((fit_train - traindata$motor_UPDRS)^2)
MSE_test = mean((fit_test - testdata$motor_UPDRS)^2)


#3a,
Loglikelihood = function(theta, sigma) {

  y = as.matrix(traindata$motor_UPDRS)
  x = as.matrix(traindata[,7:22])
  theta_temp = as.matrix(theta)

  n = dim(traindata)[1]

  loglikelihood = - n*log(sigma*sqrt(2*pi)) - (1/(2*sigma^2))* sum((y -
t(t(theta_temp) %*% t(x)))^2)
  return(loglikelihood)
}

#3b,
Ridge = function(par = c(theta, sigma), lambda) {

    theta_temp = as.matrix(par[1:16])
    sigma_temp = as.matrix(par[17])
    lambda_temp = as.matrix(lambda)
    ll = Loglikelihood(theta_temp, sigma_temp)
    ridge = -ll + lambda*sum(theta_temp^2)
    return(ridge)
    }

#3c,
RidgeOpt = function(theta_init, sigma_init, lambda) {

  init_par <- c(theta_init, sigma_init)
  ridge_opt = optim(init_par, fn = Ridge, gr = NULL, lambda = lambda,
method = "BFGS")
  return(ridge_opt)
  }

#3d,
DF = function(data, lambda) {

  x = as.matrix(data[,7:22])

  hat_matrix =  x %*% solve(( t(x) %*% x + lambda*diag(dim(x)[2]))) %*%
t(x)
  df = sum(diag(hat_matrix))
  return(df)
  }

#4,
ridge_opt_1 = RidgeOpt(coefficients(result), summary_result$sigma, 1)
opt_par_1 = ridge_opt_1$par[1:16]
print(ridge_opt_1)

ridge_opt_100 = RidgeOpt(coefficients(result), summary_result$sigma, 100)
opt_par_100 = ridge_opt_100$par[1:16]
print(ridge_opt_100)

ridge_opt_1000 = RidgeOpt(coefficients(result), summary_result$sigma, 1000)
opt_par_1000 = ridge_opt_1000$par[1:16]
print(ridge_opt_100)

#Training data
y = as.matrix(traindata$motor_UPDRS)

#Prediction train & lambda == 1
pred_train_1 = t(as.matrix(opt_par_1)) %*% t(as.matrix(traindata[,7:22]))
MSE_train_1 = sum((pred_train_1-t(y))^2)/length(pred_train_1)
DF_train_1 = DF(traindata, 1)

#Prediction train & lambda == 100
pred_train_100 = t(as.matrix(opt_par_100)) %*%
t(as.matrix(traindata[,7:22]))
MSE_train_100 = sum((pred_train_100-t(y))^2)/length(pred_train_100)
DF_train_100 = DF(traindata, 100)

#Prediction train & lambda == 1000
pred_train_1000 = t(as.matrix(opt_par_1000)) %*%
t(as.matrix(traindata[,7:22]))
MSE_train_1000 = sum((pred_train_1000-t(y))^2)/length(pred_train_1000)
DF_train_1000 = DF(traindata, 1000)

#Test data
y = as.matrix(testdata$motor_UPDRS)

#Prediction test & lambda == 1
pred_test_1 = t(as.matrix(opt_par_1)) %*% t(as.matrix(testdata[,7:22]))
MSE_test_1 = sum((pred_test_1-t(y))^2)/length(pred_test_1)
```

```
DF_test_1 = DF(testdata, 1)

#Prediction test & lambda == 100
pred_test_100 = t(as.matrix(opt_par_100)) %*% t(as.matrix(testdata[,7:22]))
MSE_test_100 = sum((pred_test_100-t(y))^2)/length(pred_test_100)
DF_test_100 = DF(testdata, 100)

#Prediction test & lambda == 1000
pred_test_1000 = t(as.matrix(opt_par_1000)) %*%
t(as.matrix(testdata[,7:22]))
MSE_test_1000 = sum((pred_test_1000-t(y))^2)/length(pred_test_1000)
DF_test_1000 = DF(testdata, 1000)

print("MSE trainingdata (lambda = 1, 100, 1000)")
print(c(MSE_train_1, MSE_train_100, MSE_train_1000))

print("MSE testdata (lambda = 1, 100, 1000)")
print(c(MSE_test_1, MSE_test_100, MSE_test_1000))

print("DF trainingdata (lambda = 1, 100, 1000)")
print(c(DF_train_1, DF_train_100, DF_train_1000))

print("DF testdata (lambda = 1, 100, 1000)")
print(c(DF_test_1, DF_test_100, DF_test_1000))
```

Appendix Code Assignment 3

```
library(dplyr)

data=read.csv("pima-indians-diabetes.csv")

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

#1
#extracting the relevant columns from our data frame
plasma_age_diabetes = data[,c(2,8,9)]

#splitting into cases of diabetes and non diabetes
diabetes = plasma_age_diabetes %>% filter(X1 == 1)
notDiabetes = plasma_age_diabetes %>% filter(X1 == 0)

plot(diabetes[,2], diabetes[,1], col="red", xlab="age", ylab="plasma
glucose concentration")
points(notDiabetes[,2], notDiabetes[,1], col="green",)
legend(60,30,legend=c("diabetes","not diabetes"), col=c("red","green"),
pch=c("o","o"))


#2
#training our model_glm
model_glm <- glm(plasma_age_diabetes[,3] ~ plasma_age_diabetes[,1] +
plasma_age_diabetes[,2],
              family="binomial", data = data[,c(2,8,9)])
predictions = model_glm[["fitted.values"]]
```

```
for (i in 1:length(predictions)) {
  if (predictions[i] > 0.5) {
    predictions[i] = 1
  }
  else {
    predictions[i] = 0
  }
}


#probabilistic equation of the estimated model_glm
#i.e. how the target depends on the features and the estimated model_glm
parameters probabilistically
#so,a way to estimate whether diabetic or not given new input data x1,x2
and given our theta from the model.
g <- function(plasma, age) {
  theta = model_glm[["coefficients"]]
  variables = c(1,plasma,age)
  g = exp( t(theta)%*%variables ) / (1 + exp(t(theta)%*%variables))
  return(g[1])
}
yHat_newData = g(100,59)

confusionMatrix_r05 = table(plasma_age_diabetes[,3], predictions)
misclassification_r05 = ( 1-
sum(diag(confusionMatrix_r05))/nrow(plasma_age_diabetes))

#combine plasma glucose and age data with our predictions, and split it
into diabetes and non-diabetes cases like earlier.
#then create a plot of this
plasma_age_predictedDiabetes <- cbind(plasma_age_diabetes[,1:2],
predictions)
predictedDiabetes = plasma_age_predictedDiabetes %>% filter(predictions ==
1)
predictedNotDiabetes = plasma_age_predictedDiabetes %>% filter(predictions
== 0)
plot(predictedDiabetes[,2], predictedDiabetes[,1], col="red",
main="Diabetes predictions using r = 0.5", xlab="age", ylab="plasma glucose
concentration")
points(predictedNotDiabetes[,2], predictedNotDiabetes[,1], col="green")
legend(70,200,legend=c("diabetes","not diabetes"), col=c("red","green"),
pch=c("o","o"))


#3
#To calculate the decision boundary: t(theta) %*% X = 0
#This gives us: theta0 + theta1x1 + theta2x2 = 0 <=> x2 = -theta0/theta2 -
theta1/theta2 * x1
slope = - coef(model_glm)[3] / coef(model_glm)[2]
intercept = - coef(model_glm)[1] / coef(model_glm)[]
abline(intercept, slope)


#4
#reset our predictions vector to its original values
predictions = model_glm[["fitted.values"]]
for (i in 1:length(predictions)) {
  if (predictions[i] > 0.2) {
    predictions[i] = 1
  }
  else {
    predictions[i] = 0
  }
}

#combine plasma glucose and age data with our new predictions, and split it
into diabetes and non-diabetes cases like earlier.
#then create a plot of this
plasma_age_predictedDiabetes <- cbind(plasma_age_diabetes[,1:2],
predictions)
predictedDiabetes = plasma_age_predictedDiabetes %>% filter(predictions ==
1)
predictedNotDiabetes = plasma_age_predictedDiabetes %>% filter(predictions
== 0)
plot(predictedDiabetes[,2], predictedDiabetes[,1], col="red",
main="Diabetes predictions using r = 0.2", xlab="age", ylab="plasma glucose
concentration")
points(predictedNotDiabetes[,2], predictedNotDiabetes[,1], col="green")
legend(70,200,legend=c("diabetes","not diabetes"), col=c("red","green"),
pch=c("o","o"))

confusionMatrix_r02 = table(plasma_age_diabetes[,3], predictions)
misclassification_r02 = ( 1-
sum(diag(confusionMatrix_r02))/nrow(plasma_age_diabetes))
print(misclassification_r02)


#reset our predictions vector to its original values
predictions = model_glm[["fitted.values"]]
for (i in 1:length(predictions)) {
  if (predictions[i] > 0.8) {
    predictions[i] = 1
  }
  else {
    predictions[i] = 0
  }
}

#combine plasma glucose and age data with our new predictions, and split it
into diabetes and non-diabetes cases like earlier.
#then create a plot of this
plasma_age_predictedDiabetes <- cbind(plasma_age_diabetes[,1:2],
predictions)
predictedDiabetes = plasma_age_predictedDiabetes %>% filter(predictions ==
1)
predictedNotDiabetes = plasma_age_predictedDiabetes %>% filter(predictions
== 0)
plot(predictedDiabetes[,2], predictedDiabetes[,1], col="red",
     main="Diabetes predictions using r = 0.8", xlab="age", ylab="plasma
glucose concentration", xlim=c(20, 80), ylim=c(70,200) )
points(predictedNotDiabetes[,2], predictedNotDiabetes[,1], col="green")
legend(70,200,legend=c("diabetes","not diabetes"), col=c("red","green"),
pch=c("o","o"))

confusionMatrix_r08 = table(plasma_age_diabetes[,3], predictions)
misclassification_r08 = ( 1-
sum(diag(confusionMatrix_r08))/nrow(plasma_age_diabetes))
print(misclassification_r08)


#5
```

```r
#Basis function expansion trick
inputs = plasma_age_diabetes[,1:2]
inputs <- inputs %>%  mutate(z1 = (inputs[,1]^4))
inputs <- inputs %>%  mutate(z2 = (inputs[,1]^3*inputs[,2]))
inputs <- inputs %>%  mutate(z3 = (inputs[,1]^2*inputs[,2]^2))
inputs <- inputs %>%  mutate(z4 = (inputs[,1]*inputs[,2]^3))
inputs <- inputs %>%  mutate(z5 = (inputs[,2]^4))

model_glm_expanded <- glm(plasma_age_diabetes[,3] ~.,
                 family="binomial", data = inputs)

predictions = model_glm_expanded[["fitted.values"]]
for (i in 1:length(predictions)) {
  if (predictions[i] > 0.5) {
    predictions[i] = 1
  }
  else {
    predictions[i] = 0
  }
}

#combine plasma glucose and age data with our new predictions, and split it
into diabetes and non-diabetes cases like earlier.
#then create a plot of this
plasma_age_predictedDiabetes <- cbind(inputs[,1:2], predictions)
predictedDiabetes = plasma_age_predictedDiabetes %>% filter(predictions ==
1)
predictedNotDiabetes = plasma_age_predictedDiabetes %>% filter(predictions
== 0)
plot(predictedDiabetes[,2], predictedDiabetes[,1], col="red",
     main="Diabetes predictions using basis function expansion, r = 0.5",
xlab="age", ylab="plasma glucose concentration",  )
points(predictedNotDiabetes[,2], predictedNotDiabetes[,1], col="green")
legend(50,80,legend=c("diabetes","not diabetes"), col=c("red","green"),
pch=c("o","o"))

confusionMatrix_basis_f_exp = table(plasma_age_diabetes[,3], predictions)
misclassification_basis_f_exp = ( 1-
sum(diag(confusionMatrix_basis_f_exp))/nrow(plasma_age_diabetes))


theta_expanded = model_glm_expanded[["coefficients"]]
```