LAB 1

The data file optdigits.csv contains information about normalized bitmaps of handwritten digits from a preprinted form from a total of 43 people. The data were first derived as 32x32 bitmaps which were then divided into nonoverlapping blocks of 4x4 and the number of on pixels are counted in each block. This has generated the resulting image of size 8x8 where each element is an integer in the range 0..16. Accordingly, each row in the data file is a sequence corresponding to 8x8 matrix, and the last element shows the actual digit from 0 to 9

1. Import the data into R and divide it into training, validation and test sets (50%/25%/25%)

```
library("kknn")

# Read in the optdigits.csv file and store the data in a variable named "data"
# Set the "header" argument to "F" to specify that the file does not contain a header row
data = read.csv("optdigits.csv", header = F)

# Get the number of rows in the data and store it in a variable named "n"
n = dim(data)[1]

# Set the seed of the random number generator to 12345
set.seed(12345)

# Randomly select 50% of the rows in the data and store them in a variable named "train"
id=sample(1:n, floor(n*0.5))
train=data[id,]

# Get the remaining rows that were not selected for the training set
id1=setdiff(1:n, id)
set.seed(12345)

# Randomly select 25% of the remaining rows and store them in a variable named "valid"
id2=sample(id1, floor(n*0.25))
valid=data[id2,]

# Get the remaining rows that were not selected for the training or validation sets
id3=setdiff(id1,id2)

# Store the remaining rows in a variable named "test"
test=data[id3,]

# Fit a k-NN classifier to the training data, with k=30 and a rectangular kernel
# Use the classifier to predict the class labels for the training data and store the predictions in "k_train"
k_train <- kknn(as.factor(V65)~., train = train, test = train, k = 30, kernel = "rectangular")

# Use the classifier to predict the class labels for the validation data and store the predictions in "k_valid"
k_valid <- kknn(as.factor(V65)~., train = train, test = valid, k = 30, kernel = "rectangular")

# Use the classifier to predict the class labels for the test data and store the predictions in "k_test"
k_test <- kknn(as.factor(V65)~., train = train, test = test, k = 30, kernel = "rectangular")

# Create a confusion matrix for the training data by comparing the predicted labels to the true labels
train_table = table(k_train$fitted.values, train$V65)

# Create a confusion matrix for the validation data by comparing the predicted labels to the true labels
valid_table = table(k_valid$fitted.values, valid$V65)

# Create a confusion matrix for the test data by comparing the predicted labels to the true labels
test_table = table(k_test$fitted.values, as.factor(test$V65))

# Define a function to calculate the misclassification rate for a given set of predictions and true labels
missclass = function(X, X1) {
  n = length(X)
  return (1 - sum(diag(table(X1, X)))/n)
}

missclass_test = missclass(k_test$fitted.values, as.factor(test$V65))
missclass_train = missclass(train$V65, k_train$fitted.values)
missclass_valid = missclass(k_valid$fitted.values, valid$V65)
```

Confusion matrices

```
test_table
```

```
##
##       0   1   2   3   4   5   6   7   8   9
##   0  77   0   0   0   0   0   0   0   0   0
##   1   0  81   0   0   0   1   0   0   7   1
##   2   0   2  98   0   0   1   0   0   0   1
##   3   0   0   0 107   0   0   0   1   1   1
##   4   1   0   0   0  94   0   0   0   0   0
##   5   0   0   0   2   0  93   0   0   0   0
##   6   0   0   0   0   2   2  90   0   0   0
##   7   0   0   0   0   6   1   0 111   0   1
##   8   0   0   3   1   2   0   0   0  70   0
##   9   0   3   0   1   5   5   0   0   0  85
```

```
train_table
```

```
##
##       0   1   2   3   4   5   6   7   8   9
##   0 202   0   0   0   1   0   0   0   0   1
##   1   0 179   1   0   3   0   2   3  10   3
##   2   0  11 190   0   0   0   0   0   0   0
##   3   0   0   0 185   0   1   0   0   2   5
##   4   0   0   0   0 159   0   0   0   0   2
##   5   0   0   0   1   0 171   0   0   0   0
##   6   0   0   0   0   0   0 190   0   2   0
##   7   0   1   1   1   7   1   0 178   0   3
##   8   0   1   0   0   1   0   0   1 188   3
##   9   0   3   0   1   4   8   0   0   2 183
```

Missclass Values

```
missclass_test
```

```
## [1] 0.05329154
```

```
missclass_train
```

```
## [1] 0.04500262
```

Comment on the quality of predictions for different digits and on the overall prediction quality.

Some numbers have worse predictions than others, for example number 4 was predicted to be a 7 a couple of times. This is most likely due to slopy writing that makes different numbers look more similar to others. The overall prediction quality is good.

#Exercise 3

3. Find any 2 cases of digit “8” in the training data which were easiest to classify and 3 cases that were hardest to classify (i.e. having highest and lowest probabilities of the correct class). Reshape features for each of these cases as matrix 8x8 and visualize the corresponding digits (by using e.g. heatmap() function with parameters Colv=NA and Rowv=NA) and comment on whether these cases seem to be hard or easy to recognize visually

```
# Get the probability of each training sample being classified as the digit "8"
prob_eight <- k_train$prob[, 9]

# Order the probabilities in decreasing order and store the resulting indices in "ordered_high"
# Keep only the top two indices
ordered_high <- order(prob_eight, decreasing = T)
ordered_high <- ordered_high[1:2]

# Order the probabilities in increasing order and store the resulting indices in "ordered_low"
ordered_low <- order(prob_eight, decreasing = F)

# Get the indices of the "ordered_low" vector that correspond to samples with a non-zero probability of being cla
ssified as "8"
ordered_index <- which(prob_eight > 0)

# Create an empty vector to store the indices of the first three samples in "ordered_low" that are actually the d
igit "8"
ordered_low_eights <- c()

# Set an index variable to 1
index = 1

# Use a loop to append the first three indices from "ordered_low" that correspond to samples that are the digit "
8" to "ordered_low_eights"
while(length(ordered_low_eights) < 3) {
  # Get the class label of the current sample
  value <-train[ordered_low[index], 65]

  # If the sample is the digit "8", append its index to "ordered_low_eights"
  if (value == 8) {
    ordered_low_eights <- append(ordered_low_eights, ordered_low[index])
  }

  # Increment the index variable
  index <- index + 1
}

# Print the "ordered_low_eights" vector
print(ordered_low_eights)
```

```
## [1]  520  431 1294
```

```
# Combine the "ordered_low_eights" and "ordered_high" vectors to create the "ordered_matrix" vector
ordered_matrix <- c(ordered_low_eights, ordered_high)

# Use a loop to create a heatmap for each of the samples corresponding to the indices in the "ordered_matrix" vec
tor
for (i in ordered_matrix){
  # Reshape the sample's pixel values into a 8x8 matrix
  my_heatmap <- matrix(as.numeric(train[i,1:64]), nrow=8,ncol=8, byrow = T)

  # Create a heatmap of the matrix and set the main title to the true class label of the sample
  heatmap(my_heatmap, Colv = NA, Rowv = NA, main =train[i, 65])
}
```

The eights that were easy to classify are also very easy to see as eights. The eights that were most difficult to spot are very difficult to see without knowing that they’re eights beforehand.

#Exercise 4

Fit a K-nearest neighbor classifiers to the training data for different values of KK = 1,2, … , 30 and plot the dependence of the training and validation misclassification errors on the value of K (in the same plot). How does the model complexity change when K increases and how does it affect the training and validation errors? Report the optimal KK according to this plot. Finally, estimate the test error for the model having the optimal K, compare it with the training and validation errors and make necessary conclusions about the model quality.

```
# Initialize two empty vectors to store the misclassification rates for the training and validation sets
# for different values of k
train_miss_error <- numeric(30)
val_miss_error <- numeric(30)

# Use a loop to fit a k-NN classifier to the training data for each value of k from 1 to 30
# Use the classifier to predict the class labels for the training and validation sets
# Calculate the misclassification rate for each set and store the results in the "train_miss_error" and "val_miss
_error" vectors
for (i in 1:30){
  k_valid <- kknn(as.factor(V65)~., train = train, test = valid, k = i, kernel = "rectangular")
  k_train <- kknn(as.factor(V65)~., train = train, test = train, k = i, kernel = "rectangular")

  train_miss_error[i] <- missclass(k_train$fitted.values, train$V65)
  val_miss_error[i] <- missclass(k_valid$fitted.values, valid$V65)
}

# Create a plot of the "train_miss_error" and "val_miss_error" vectors
# Use k as the x-axis and the misclassification rate as the y-axis
plot = plot(c(1:30), train_miss_error, ylab = "train_miss_error", xlab = "k", col="pink")
points(c(1:30), val_miss_error, col="green")
```

```
# Fit a k-NN classifier to the training data with k=4
# Use the classifier to predict the class labels for the training, validation, and test sets
k_test <- kknn(as.factor(V65)~., train = train, test = test, k = 4, kernel = "rectangular")
k_train <- kknn(as.factor(V65)~., train = train, test = train, k = 4, kernel = "rectangular")
k_valid <- kknn(as.factor(V65)~., train = train, test = valid, k = 4, kernel = "rectangular")

missclass_test = missclass(k_test$fitted.values, as.factor(test$V65))
missclass_train = missclass(train$V65, k_train$fitted.values)
missclass_valid = missclass(k_valid$fitted.values, valid$V65)
```

How does the model complexity change when K increases and how does it affect the training and validation errors?

The model complexity remains the same as k increases, since it's not learning more parameters or features as k increases. One can argue that the overall complexity is increased since there are more data points added, but this does not mean that the complexity of the model is higher. The best K values are the ones with the lowest missclass error, in this case k=3 and k=4 are the best values.

We can see that train gives the lowest missclass error because it is the date we made the model from. Both test and validation is slightly higher which is to be expected. With the right k value computed with the validation data we get a test error which is just above 2.5% Which results in a pretty high quality model.

#Exercise 5

Fit K-nearest neighbor classifiers to the training data for different values of KK = 1,2, … , 30, compute the error for the validation data as cross-entropy (when computing log of probabilities add a small constant within log, e.g. 1e-15, to avoid numerical problems) and plot the dependence of the validation error on the value of KK. What is the optimal KK value here? Assuming that response has multinomial distribution, why might the cross-entropy be a more suitable choice of the error function than the misclassification error for this problem?

```
cross.entropy <- function(p, phat){
  x <- 0
  for (i in 1:length(p)){
    x <- x + (p[i] * log(phat[i]))
  }
  return(-x)
}

train_miss_error <- as.vector(matrix(0,ncol = 30))
val_miss_error <- as.vector(matrix(0,ncol = 30))

entropy_error <- as.vector(matrix(0,ncol = 30))

for (i in 1:30){
  k_valid <- kknn(as.factor(V65)~., train = train, test = valid, k = i, kernel = "rectangular")
  k_train <- kknn(as.factor(V65)~., train = train, test = train, k = i, kernel = "rectangular")


  for (j in 0:9){
    cross_val <- valid$V65 == j
    cross_train <- train$V65 == j

    bool_val <- (which(cross_val, useNames = T))
    prob_val <- k_valid$prob[cross_val, as.character(j)] + 1e-15

    bool_train <- (which(cross_train, useNames = T))
    prob_train <- k_train$prob[cross_train, as.character(j)] + 1e-15

    val_miss_error[i] <- val_miss_error[i] + sum(-(bool_val*log(prob_val)))
    train_miss_error[i] <- train_miss_error[i] + sum(-(bool_train*log(prob_train)))

    #val_miss_error[i] <- cross.entropy(bool_val, prob_val)
    #train_miss_error[i] <- cross.entropy(bool_train, prob_train)



    entropy_error[i] <- abs(val_miss_error[i]-train_miss_error[i])
  }

}

plot(c(1:30), entropy_error, ylab = "cross_entropy_error", xlab = "k", col="blue")
```

```
which.min(entropy_error)
```

```
## [1] 5
```

What is the optimal KK value here? Assuming that response has multinomial distribution, why might the cross-entropy be a more suitable choice of the error function than the missclassification error for this problem?

This model might be more suitable because of lower complexity levels than missclass error. The optimal K value here is K = 5.

#Assigment 2

The data file parkinson.csv is composed of a range of biomedical voice measurements from 42 people with early-stage Parkinson's disease recruited to a six-month trial of a telemonitoring device for remote symptom progression monitoring. The purpose is to predict Parkinson's disease symptom score (motor UPDRS) from the following voice characteristics: • Jitter(%),Jitter(Abs),Jitter:RAP,Jitter:PPQ5,Jitter:DDP - Several measures of variation in fundamental frequency • Shimmer,Shimmer(dB),Shimmer:APQ3,Shimmer:APQ5,Shimmer:APQ11,Shi mmer:DDA - Several measures of variation in amplitude • NHR,HNR - Two measures of ratio of noise to tonal components in the voice • RPDE - A nonlinear dynamical complexity measure • DFA - Signal fractal scaling exponent • PPE - A nonlinear measure of fundamental frequency variation

Divide it into training and test data (60/40) and scale it appropriately. In the coming steps, assume that motor_UPDRS is normally distributed and is a function of the voice characteristics, and since the data are scaled, no intercept is needed in the modelling.

```
library(caret)
```

```
## Warning: package 'caret' was built under R version 4.1.3
```

```
## Loading required package: ggplot2
```

```
## Warning: package 'ggplot2' was built under R version 4.1.3
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:kknn':
##
##     contr.dummy
```

```
parkin = read.csv("parkinsons.csv", header = T)
parkin_corr <- data.frame(parkin[c(5,7:22)]) #Remove unused voice characteristics
parkin_scaled <- as.data.frame(scale(parkin_corr))

n = dim(parkin_corr)[1]
set.seed(12345)

id=sample(1:n, floor(n*0.6))

train_pre=parkin_corr[id,]
test_pre=parkin_corr[-id,]

data_scaler<-preProcess(train_pre)
train<-predict(data_scaler,train_pre)
test<-predict(data_scaler,test_pre)
```

#Excercise 2

Compute a linear regression model from the training data, estimate training and test MSE and comment on which variables contribute significantly to the model.

```
fit = lm(motor_UPDRS ~ .-1, data = train)
sum = summary(fit)
MSE_train<-mean(sum$residuals^2)
pred_test<-predict(fit,test)
MSE_test<-mean((test$motor_UPDRS-pred_test)^2)
print(MSE_train)
```

```
## [1] 0.8785431
```

```
print(MSE_test)
```

```
## [1] 0.9354477
```

```
print(sum)
```

```
##
## Call:
## lm(formula = motor_UPDRS ~ . - 1, data = train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0255 -0.7363 -0.1087  0.7333  2.1960
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## Jitter...       0.186931   0.149561   1.250 0.211431
## Jitter.Abs.    -0.169609   0.040805  -4.157 3.31e-05 ***
## Jitter.RAP     -5.269544  18.834160  -0.280 0.779658
## Jitter.PPQ5    -0.074568   0.087766  -0.850 0.395592
## Jitter.DDP      5.249558  18.837525   0.279 0.780510
## Shimmer         0.592436   0.205981   2.876 0.004050 **
## Shimmer.dB.    -0.172655   0.139316  -1.239 0.215315
## Shimmer.APQ3   32.070932  77.159242   0.416 0.677694
## Shimmer.APQ5   -0.387507   0.113789  -3.405 0.000668 ***
## Shimmer.APQ11   0.305546   0.061236   4.990 6.34e-07 ***
## Shimmer.DDA   -32.387241  77.158814  -0.420 0.674695
## NHR            -0.185387   0.045567  -4.068 4.84e-05 ***
## HNR            -0.238543   0.036395  -6.554 6.41e-11 ***
## RPDE            0.004068   0.022664   0.179 0.857556
## DFA            -0.280318   0.020136 -13.921  < 2e-16 ***
## PPE             0.226467   0.032881   6.887 6.70e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9394 on 3509 degrees of freedom
## Multiple R-squared:  0.1212, Adjusted R-squared:  0.1172
## F-statistic: 30.25 on 16 and 3509 DF,  p-value: < 2.2e-16
```

The variables p-values that are higher than 0.05 does not contribute at all (insignificant P-values). The values that contribute significantly are the ones with low p-values. The variables that will contribute significantly to the model are the ones with the highest amount of stars in the table, which represents the lowest p-values.

MSE_train = 0.8785431 MSE_test = 0.9354477

#Exercise3

Implement 4 following functions by using basic R commands only (no external packages): a. Loglikelehood function that for a given parameter vector $\theta\theta$ and dispersion $\sigma\sigma$ computes the log-likelihood function log $PP(TT|\theta\theta, \sigma\sigma)$ for the stated model and the training data

b. Ridge function that for given vector $\theta\theta$, scalar $\sigma\sigma$ and scalar $\lambda\lambda$ uses function from 3a and adds up a Ridge penalty $\lambda\lambda\|\theta\theta\|2$ to the minus loglikelihood.

c. RidgeOpt function that depends on scalar $\lambda\lambda$, uses function from 3b and function optim() with method="BFGS" to find the optimal $\theta\theta$ and $\sigma\sigma$ for the given $\lambda\lambda$.

d. Df function that for a given scalar $\lambda\lambda$ computes the degrees of freedom of the Ridge model based on the training data.

```
log_likelihood <- function(train, Y, theta, sigma){

  n <- dim(train)[1]
  train_theta = train%*%theta

  sum1 <- n*log(sigma^2)/2
  sum2 <- n*log(2*pi)/2
  sum3 <- sum((train_theta-Y)^2)
  sum4 <- sum3/(2*sigma^2)

  return (-sum1-sum2-sum4)
}


ridge <- function(train, theta, lambda, Y){
  n<-dim(train)[2]
  sigma <- theta[n+1]
  theta <-as.matrix(theta[1:n])
  log_like <- log_likelihood(theta=theta,Y=Y,sigma=sigma,train=train)
  ridge <- -log_like + lambda*sum(theta^2)
  return(ridge)

}

ridgeOpt <- function(lambda, train, Y){

  train <- as.matrix(train)
  N = dim(train)[2]
  init_theta = integer(N)
  init_sigma = 1

  opt <- optim(par = c(init_theta,init_sigma), fn = ridge, lambda = lambda, train = train, Y = Y, method = "BFGS"
)
  return(opt)
}

dF <- function(X, lambda){
  #From the course formula
  X <- as.matrix(X)
  Xt <- t(X)
  n <- dim(X)[2]
  I <- diag(n)
  P <- X%*%solve((Xt%*%X + (lambda*I)))%*%Xt
  return(sum(diag(P)))
}
```

```
  AIC = function(train,Y,theta, sigma, lambda){
    log_like = log_likelihood(train = train, Y=Y, theta = theta, sigma = sigma)
    N = dim(train)[1] # No of data points
    df = dF(train,lambda)
    aic = (-2*log_like/N) + (2*df/N) #(-2*Log-likelihood/N) + 2*(df/N)
    return(aic)
  }
```

#Exercise 4

By using function RidgeOpt, compute optimal $\theta\theta$ parameters for $\lambda\lambda$ = 1, $\lambda\lambda$ = 100 and $\lambda\lambda$ = 1000. Use the estimated parameters to predict the motor_UPDRS values for training and test data and report the training and test MSE values. Which penalty parameter is most appropriate among the selected ones? Compute and compare the degrees of freedom of these models and make appropriate conclusions.

```
xtrain<-as.matrix(train[2:17])
ytrain<-as.matrix(train[1])
xtest=as.matrix(test[2:17])
ytest<-as.matrix(test[1])


for (lambda in c(1, 100, 1000)){
  opt = ridgeOpt(lambda, xtrain, ytrain)
  theta <- as.matrix(opt$par[1:16])
  sigma<- opt$par[17]
  MSE_train = mean((xtrain%*%theta - ytrain)^2)
  MSE_test = mean((xtest%*%theta - ytest)^2)
  aic = AIC(train=xtrain,Y= ytrain,theta = theta,sigma= sigma, lambda= lambda)
  print(paste("Lambda:",lambda))
  print(paste("TrainMSE:",MSE_train))
  print(paste("TestMSE:",MSE_test))
  print(paste("AIC:", aic)) #Unecessary


}
```

```
## [1] "Lambda: 1"
## [1] "TrainMSE: 0.878627086520567"
## [1] "TestMSE: 0.934996946852113"
## [1] "AIC: 2.71634659521451"
## [1] "Lambda: 100"
## [1] "TrainMSE: 0.884410410201325"
## [1] "TestMSE: 0.932331574068859"
## [1] "AIC: 2.72067414889541"
## [1] "Lambda: 1000"
## [1] "TrainMSE: 0.921121567431251"
## [1] "TestMSE: 0.953948178939008"
## [1] "AIC: 2.75891604085416"
```

Which penalty parameter is most appropriate among the selected ones? Compute and compare the degrees of freedom of these models and make appropriate conclusions.

We have read that to find the best aic value among mutiple aic values is the lowest. In our example it seems like lambda = 1 and lambda = 100 gives similar aic values which is hard to draw conclusions from. Lambda = 1 gives the lowest AIC value. Since the the Lambda = 100 has the lowest test_MSE, this is the best value. The degrees of freedom reflects the complexity of the model, so the larger lambda, the smaller degrees of freedom we would get. We get more complex models, basically penalizing the complexity.

##Assignment 3.

The data file pima-indians-diabetes.csv contains information about the onset of diabetes within 5 years in Pima Indians given medical details. The variables are (in the same order as in the dataset):

Make a scatterplot showing a Plasma glucose concentration on Age where observations are colored by Diabetes levels. Do you think that Diabetes is easy to classify by a standard logistic regression model that uses these two variables as features? Motivate your answer.

```
prime = read.csv("pima-indians-diabetes.csv", header = F)

set.seed(12345)
```

```
coloor <- function(x){
  if (x==1){
    c = "red"
  } else{
    c="green"
  }
  return(c)
}

coloors = sapply(prime$V9, coloor)
plot( prime$V2, prime$V8, xlab = "Plasma", ylab = "Age", main = "doabets", col = coloors)
```

seems to be different, no obvious correlation between the variables. Making a linear regression would not provide a meaningful line.

#Exercise 2

Train a logistic regression model with yy =Diabetes as target $xx1$ =Plasma glucose concentration and $xx2$ =Age as features and make a prediction for all observations by using rr = 0.5 as the classification threshold. Report the probabilistic equation of the estimated model (i.e., how the target depends on the features and the estimated model parameters probabilistically). Compute also the training misclassification error and make a scatter plot of the same kind as in step 1 but showing the predicted values of Diabetes as a color instead. Comment on the quality of the classification by using these results

```
glm.fits = glm(V9~ V2 + V8, prime, family = "binomial" )
prob=predict(glm.fits, type="response")
pred=ifelse(prob>0.5, 'red','green')

table = table(pred, prime$V9)

miss <- missclass(pred, prime$V9)

plot(prime$V2, prime$V8,col=pred, xlab = "Plasma glucose levels", ylab = "Age", main = paste("Missclass error", t
oString(miss), sep=" = ") )
```

```
table
```

```
##
## pred      0   1
##   green 436 138
##   red    64 130
```

```
#summary(table)
#summary(glm.fits)
glm.fits$coefficients
```

```
## (Intercept)          V2          V8
## -5.91244906  0.03564404  0.02477835
```

Probabilistic equation:

$$p(y=1\,|\,x_*) = g(x_*, \hat{\theta}) = \frac{1}{1+e^{-\hat{\theta}^T x_*}} = \frac{1}{1+e^{-(-5.91244906+0.03564404x_1+0.02477835x_2)}}$$

It seems to be an acceptable classification with about 1/4 error but not perfect There seems to be a division when you reach above 150 in plasma-glucose levels. But this threshold seems to lower as you age. An older person is more likely to have diabetes even with lower plasma-glucose levels.

#Exercise 3

Use the model estimated in step 2 to a) report the equation of the decision boundary between the two classes b) add a curve showing this boundary to the scatter plot in step 2. Comment whether the decision boundary seems to catch the data distribution well.

```
r=.5
glm.fits = glm(V9~ V2 + V8, prime, family = "binomial" )
cf = glm.fits$coefficients
prob=predict(glm.fits, type="response")
pred=ifelse(prob>0.5, 'red','green')

plot(prime$V2, prime$V8,col=pred, ylab = "Age", xlab= "Plasma", main = paste("Missclass Error", toString(miss), s
ep=" = "))

slope = coef(glm.fits)[2]/(-coef(glm.fits)[3])
intercept = (coef(glm.fits)[1] + log(-r/(r-1)))/(-coef(glm.fits)[3])

#clip(min(x1),max(x1), min(x2), max(x2))
abline(intercept , slope, lwd = 2, lty = 2)
```

```
he = log(-.2/(.2-1))
he
```

```
## [1] -1.386294
```

```
slope
```

```
##        V2
## -1.438516
```

```
intercept
```

```
## (Intercept)
##    238.6135
```

y = -1.438516X + 238.6135

We can see how the distribution is hard to define and capture in the plot from the first assignment. This attempt is a pretty good compromise, where most none diabetes are captured in the green to the left, and most diabetes cases are captured to the right. But with a missclass error of 26% it still misses alot, which we can also see when compared to the plot from the first assignment.

#Exercise 4

Make same kind of plots as in step 2 but use thresholds rr = 0.2 and rr = 0.8. By using these plots, comment on what happens with the prediction when rr value changes.

```
for(r in c(.2,.5, .8)) {
  pred=ifelse(prob>r, 'red','green')
  table(pred, prime$V9)


  miss <- missclass(pred, prime$V9)

  plot(prime$V2, prime$V8,col=pred, ylab = "Age", xlab= "Plasma", main = paste("Missclass_Error = ", toString(mis
s), "\n r = ", r, sep= ""))

  slope = coef(glm.fits)[2]/(-coef(glm.fits)[3])
  intercept = (coef(glm.fits)[1] - log(-r/(r-1)))/(-coef(glm.fits)[3])

  #clip(min(x1),max(x1), min(x2), max(x2))
  abline(intercept , slope, lwd = 2, lty = 2)
}
```

With r=.2 and r=.8, the missclassification error increased. r=.2 was clearly worse (37% error).

#Exercise 5

Perform a basis function expansion trick by computing new features $zz1$ = $xx1$ 4, $zz2$ = $xx1$ 3$xx2$, $zz3$ = $xx1$ 2$xx2$ 2, $zz4$ = $xx1xx2$ 3, $zz5$ = $xx2$ 4 , adding them to the data set and then computing a logistic regression model with yy as target and $xx1$, $xx2$, $zz1$, … , $zz5$ as features. Create a scatterplot of the same kind as in step 2 for this model and compute the training misclassification rate. What can you say about the quality of this model compared to the previous logistic regression model? How have the basis expansion trick affected the shape of the decision boundary and the prediction accuracy?

```
expanded <- prime

expanded$z1 <- expanded$V2 ** 4
expanded$z2 <- expanded$V2 ** 3 * expanded$V8
expanded$z3 <- expanded$V2 ** 2 * expanded$V8 ** 2
expanded$z4 <- expanded$V2 * expanded$V8 ** 3
expanded$z5 <- expanded$V8 ** 4

glm.fits = glm(V9~ V2 + V8 + z1 + z2 + z3 + z4 + z5, expanded, family = "binomial" )


for(r in c(.2,.5, .8)) {
  prob=predict(glm.fits, type="response")
  pred=ifelse(prob>r, 'red','green')
  table(pred, expanded$V9)

  miss <- missclass(pred, prime$V9)

  plot(expanded$V2, expanded$V8,col=pred, ylab = "Age", xlab= "Plasma", main = paste("Missclass_Error = ", toStri
ng(miss), "\n r = ", r, sep= ""))
}
```

This model has a lower missclass error, so the quality of this model is slightly better than the previous one. The linearity of the model is however gone, since we've added non-linear variables. This causes the plotted colors to look more like a “slice”

Untitled

```
library(glmnet)
```

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

```
## Loaded glmnet 4.1-4
```

```
library(tree)
library(caret)
```

```
## Loading required package: ggplot2
```

```
## Loading required package: lattice
```

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
 #Assume that Fat can be modeled as a linear regression in which absorbance characteristics (Channels) are used a
s features. Report the underlying probabilistic model, fit the linear regression to the training data and estimat
e the training and test errors. Comment on the quality of fit and prediction and therefore on the quality of mode
l.

#Read in data from "tecator.csv" and store in a data frame called "tecator"
tecator = read.csv("tecator.csv", header = T)

# Get the number of rows in the data frame and store in a variable called "n"
n = dim(tecator)[1]

# Set the seed for the random number generator
set.seed(12345)

# Create a new data frame called "df" that contains all the columns from the "tecator" data frame except for the
first column
df = data.frame(tecator[c(2:102)])

# Generate a random sample of row indices from the "tecator" data frame
id=sample(1:n, floor(n*0.5))

# Use the "id" indices to select a subset of rows from the "df" data frame and store them in a new data frame cal
led "train"
train = df[id,]

# Use the "id" indices to select all rows except those in the "id" subset from the "df" data frame and store them
in a new data frame called "test"
test = df[-id,]

# Fit a linear regression model to the "train" data frame, with the "Fat" column as the response variable and all
the other columns as predictor variables
fit = lm(Fat~ ., data = train)

# Use the "fit" model to make predictions on the training data and store the predictions in a vector called "trai
n_preds"
train_preds = predict(fit, train)

# Use the "fit" model to make predictions on the test data and store the predictions in a vector called "test_pre
ds"
test_preds = predict(fit, test)

# Generate a summary of the "fit" model and store it in an object called "sum"
sum = summary(fit)

# Calculate the mean squared error (MSE) of the model's predictions on the training data and store in a variable
called "MSE_train"
MSE_train=mean((train_preds - train$Fat)^2)

# Calculate the MSE of the model's predictions on the test data and store in a variable called "MSE_test"
MSE_test=mean((test_preds - test$Fat)^2)

# Print "Test error" and the MSE of the model's predictions on the test data
print("Test error")
```

```
## [1] "Test error"
```

```
MSE_test
```

```
## [1] 722.4294
```

```
# Print "Train error" and the MSE of the model's predictions on the training data
print("Train error")
```

```
## [1] "Train error"
```

```
MSE_train
```

```
## [1] 0.005709117
```

```
#Fit the LASSO regression model to the training data. Present a plot illustrating how the regression coefficients depend on the log of penalty factor (log λλ) and interpret this plot. What value of the penalty factor can be chosen if we want to select a model with only three features?"

# Get the "Fat" column from the "train" data frame and store in a variable called "y"
y = train$Fat

# Get the first 100 columns from the "train" data frame and store in a variable called "x"
x = train[1:100]

# Fit a lasso regression model to the "x" and "y" data using the "glmnet" function, with a Gaussian error distribution
model_lasso= glmnet(as.matrix(x), as.matrix(y), alpha=1,family="gaussian")

# Plot the model's coefficients as a function of the regularization parameter "lambda"
plot(model_lasso, xvar = "lambda")
```

```
# Use the "model_lasso" model to make predictions on the "x" data and store the predictions in a variable called "ynew"
ynew=predict(model_lasso, newx=as.matrix(x), type="response")

#Repeat step 3 but fit Ridge instead of the LASSO regression and compare the plots from steps 3 and 4. Conclusions?"

# Get the "Fat" column from the "train" data frame and store in a variable called "y"
y = train$Fat

# Get the first 100 columns from the "train" data frame and store in a variable called "x"
x = train[1:100]

# Fit a ridge regression model to the "x" and "y" data using the "glmnet" function, with a Gaussian error distribution
model_lasso= glmnet(as.matrix(x), as.matrix(y), alpha=0,family="gaussian")

# Plot the model's coefficients as a function of the regularization parameter "lambda"
plot(model_lasso, xvar = "lambda")
```

```
# Use the "model_lasso" model to make predictions on the "x" data and store the predictions in a variable called "ynew"
ynew=predict(model_lasso, newx=as.matrix(x), type="response")


#Use cross-validation with default number of folds to compute the optimal LASSO model. Present a plot showing the dependence of the CV score on log λλ and comment how the CV score changes with log λλ. Report the optimal λλ and how many variables were chosen in this model. Does the information displayed in the plot suggests that the optimal λλ value results in a statistically significantly better prediction than log λλ = −4? Finally, create a scatter plot of the original test versus predicted test values for the model corresponding to optimal lambda and comment whether the model predictions are good."

# Fit a lasso regression model to the "x" and "y" data using cross-validation and the "cv.glmnet" function, with a Gaussian error distribution
model_lasso= cv.glmnet(as.matrix(x), as.matrix(y), alpha=1,family="gaussian")

# Get the value of the regularization parameter "lambda" that resulted in the lowest cross-validation error
lambda_min = model_lasso$lambda.min

# Plot the model's coefficients as a function of the regularization parameter "lambda"
plot(model_lasso, xvar = "lambda")
```

```
# Fit a lasso regression model to the "x" and "y" data using the "glmnet" function and the "lambda_min" value, with a Gaussian error distribution
better_model = glmnet(as.matrix(x), as.matrix(y), lambda = lambda_min, alpha = 1, family = "gaussian")

# Use the "better_model" model to make predictions on the "x" data and store the predictions in a variable called "ynew"
ynew=predict(better_model, newx=as.matrix(x), s = lambda_min , type="response")

# Plot the original "y" values against the predicted "ynew" values, with a red color and a labeled x-axis and y-axis
plot(y, ynew, xlab = "Original", ylab = "Predicted",col = "red", main = "Scatter plot")

# Add a line with slope 1 and intercept 0 to the plot
abline(0,1)
```

```
#Import the data to R, remove variable “duration” and divide into training/validation/test as 40/30/30: use data
partitioning code specified in Lecture 2a."



# Read in data from "bank-full.csv" and store in a data frame called "d"
d = read.csv("bank-full.csv", sep = ";", stringsAsFactors = TRUE)

# Create a copy of the "d" data frame called "data"
data = d

# Remove the "duration" column from the "data" data frame
data$duration = c()

# Get the "y" column from the "d" data frame and store in a variable called "output"
output = d['y']

# Get the number of rows in the "data" data frame and store in a variable called "n"
n = dim(data)[1]

# Set the seed for the random number generator
set.seed(12345)

# Generate a random sample of row indices from the "data" data frame
id=sample(1:n, floor(n*0.4))

# Use the "id" indices to select a subset of rows from the "data" data frame and store them in a new data frame c
alled "train"
train=data[id,]

# Get the indices of the rows in the "data" data frame that are not in the "id" subset
id1=setdiff(1:n, id)

# Set the seed for the random number generator
set.seed(12345)

# Generate a random sample of row indices from the "id1" subset
id2=sample(id1, floor(n*0.3))

# Use the "id2" indices to select a subset of rows from the "data" data frame and store them in a new data frame
called "valid"
valid=data[id2,]

# Get the indices of the rows in the "id1" subset that are not in the "id2" subset
id3=setdiff(id1,id2)

# Use the "id3" indices to select a subset of rows from the "data" data frame and store them in a new data frame
called "test"
test=data[id3,]

#Fit decision trees to the training data so that you change the default settings one by one (i.e. not simultaneou
sly): a. Decision Tree with default settings. b. Decision Tree with smallest allowed node size equal to 7000. c.
Decision trees minimum deviance to 0.0005. and report the misclassification rates for the training and validation
data.  Which model is the best one among these three? Report how changing the deviance and node size affected the
size of the trees and explain why."


# Fit a decision tree model to the "train" data using the "tree" function, using all of the columns except the "y
" column as predictors
fit=tree(as.factor(y)~., data=train)

# Plot the decision tree model
plot(fit)

# Add text labels to the plot to show the splits at each node of the tree
text(fit, pretty=0)
```

```
# Fit a decision tree model to the "train" data using the "tree" function, using all of the columns except the "y
" column as predictors, with a minimum size of 7000 for each leaf
fit2=tree(as.factor(y)~., data=train, minsize=7000)

# Plot the decision tree model
plot(fit2)

# Add text labels to the plot to show the splits at each node of the tree
text(fit2, pretty=0)
```

```
# Fit a decision tree model to the "train" data using the "tree" function, using all of the columns except the "y
" column as predictors, with a minimum relative decrease in impurity of 0.0005 required to split a node
fit3=tree(as.factor(y)~., data=train, mindev=0.0005)

# Plot the decision tree model
plot(fit3)

# Add text labels to the plot to show the splits at each node of the tree
text(fit3, pretty=0)
```

```
#Use training and validation sets to choose the optimal tree depth in the model 2c: study the trees up to 50 leav
es. Present a graph of the dependence of deviances for the training and the validation data on the number of leav
es and interpret this graph in terms of bias-variance tradeoff. Report the optimal amount of leaves and which var
iables seem to be most important for decision making in this tree. Interpret the information provided by the tree
structure (not everything but most important findings)."

# Make predictions on the "train" data using the first decision tree model and store the predictions in a variabl
e called "Yfit_t"
Yfit_t=predict(fit, newdata=train, type="class")

# Create a confusion matrix for the predictions on the "train" data using the first decision tree model
t1<-table(train$y,Yfit_t)

# Calculate the misclassification rate for the predictions on the "train" data using the first decision tree mode
l
mis_t1 <- 1-sum(diag(t1))/sum(t1)

# Make predictions on the "train" data using the second decision tree model and store the predictions in a variab
le called "Yfit_t2"
Yfit_t2=predict(fit2, newdata=train, type="class")

# Create a confusion matrix for the predictions on the "train" data using the second decision tree model
t2<-table(train$y,Yfit_t2)

# Calculate the misclassification rate for the predictions on the "train" data using the second decision tree mod
el
mis_t2 <- 1-sum(diag(t2))/sum(t2)

# Make predictions on the "train" data using the third decision tree model and store the predictions in a variabl
e called "Yfit_t3"
Yfit_t3=predict(fit3, newdata=train, type="class")

# Create a confusion matrix for the predictions on the "train" data using the third decision tree model
t3<-table(train$y,Yfit_t3)

# Calculate the misclassification rate for the predictions on the "train" data using the third decision tree mode
l
mis_t3 <- 1-sum(diag(t3))/sum(t3)

# Make predictions on the "valid" data using the first decision tree model and store the predictions in a variabl
e called "Yfit_v"
Yfit_v=predict(fit, newdata=valid, type="class")

# Create a confusion matrix for the predictions on the "valid" data using the first decision tree model
v1<-table(valid$y,Yfit_v)

# Calculate the misclassification rate for the predictions on the "valid" data using the first decision tree mode
l
mis_v1<-1-sum(diag(v1))/sum(v1)

# Make predictions on the "valid" data using the second decision tree model and store the predictions in a variab
le called "Yfit_v2"
Yfit_v2=predict(fit2, newdata=valid, type="class")

# Create a confusion matrix for the predictions on the "valid" data using the second decision tree model
v2<-table(valid$y,Yfit_v2)

# Calculate the misclassification rate for the predictions on the "valid" data using the second decision tree mod
el
mis_v2<-1-sum(diag(v2))/sum(v2)

# Make predictions on the "valid" data using the third decision tree model and store the predictions in a variabl
e called "Yfit_v3"
Yfit_v3=predict(fit3, newdata=valid, type="class")

# Create a confusion matrix for the predictions on the "valid" data using the third decision tree model
v3<-table(valid$y,Yfit_v3)

# Calculate the misclassification rate for the predictions on the "valid" data using the third decision tree mode
l
mis_v3<-1-sum(diag(v3))/sum(v3)

# Print the misclassification rates for the predictions on the "train" and "valid" data using the first decision
tree model
print("1) Training and validation")
```

```
## [1] "1) Training and validation"
```

```
print(mis_t1)
```

```
## [1] 0.1048441
```

```
print(mis_v1)
```

```
## [1] 0.1092679
```

```
# Print the misclassification rates for the predictions on the "train" and "valid" data using the second decision tree model
print("2) Training and validation")
```

```
## [1] "2) Training and validation"
```

```
print(mis_t2)
```

```
## [1] 0.1048441
```

```
print(mis_v2)
```

```
## [1] 0.1092679
```

```
# Print the misclassification rates for the predictions on the "train" and "valid" data using the third decision tree model
print("3) Training and validation")
```

```
## [1] "3) Training and validation"
```

```
print(mis_t3)
```

```
## [1] 0.09400575
```

```
print(mis_v3)
```

```
## [1] 0.1119221
```

```
#Estimate the confusion matrix, accuracy and F1 score for the test data by using the optimal model from step 3. C
omment whether the model has a good predictive power and which of the measures (accuracy or F1-score) should be p
referred here."


# Create two vectors called "trainScore" and "testScore" with length 50 and filled with 0s
trainScore=rep(0,50)
testScore=rep(0,50)

# Loop through the numbers 2 to 50
for(i in 2:50) {
  # Prune the third decision tree model to have "i" leaves
  prunedTree=prune.tree(fit3,best=i)

  # Make predictions on the "valid" data using the pruned decision tree model and store the predictions in a vari
able called "pred"
  pred=predict(prunedTree, newdata=valid, type="tree")

  # Calculate the deviance for the pruned decision tree model on the "train" data and store it in the "trainScore
" vector
  trainScore[i]=deviance(prunedTree)

  # Calculate the deviance for the predictions on the "valid" data using the pruned decision tree model and store
it in the "testScore" vector
  testScore[i]=deviance(pred)
}

# Plot the deviances for the pruned decision tree models on the "train" data (in red) and the deviances for the p
redictions on the "valid" data (in blue)
plot(2:50, trainScore[2:50], type="b", col="red", ylim=c(min(testScore[-1]), max(trainScore[-1])))
points(2:50, testScore[2:50], type="b", col="blue")
```

```
# Find the number of leaves that gives the minimum deviance on the "valid" data
optimal_leaves = which.min(testScore[2:50])

# Prune the third decision tree model to have the optimal number of leaves
finalTree=prune.tree(fit3, best=optimal_leaves)

# Make predictions on the "valid" data using the final pruned decision tree model and store the predictions in a
variable called "finalfit"
finalfit=predict(finalTree, newdata=valid, type="class")

# Create a confusion matrix for the predictions on the "valid" data using the final pruned decision tree model
tab = table(valid$y,finalfit)

# Plot the final pruned decision tree model
plot(finalTree)
```

```
# Print the summary of the third decision tree model
summary(fit3)
```

```
##
## Classification tree:
## tree(formula = as.factor(y) ~ ., data = train, mindev = 5e-04)
## Variables actually used in tree construction:
##  [1] "poutcome"  "month"     "contact"   "marital"   "day"       "campaign"
##  [7] "job"       "pdays"     "age"       "balance"   "housing"   "education"
## [13] "previous"
## Number of terminal nodes:  122
## Residual mean deviance:  0.5213 = 9363 / 17960
## Misclassification error rate: 0.09362 = 1693 / 18084
```

```
# Print the summary of the final pruned decision tree model
summary(finalTree)
```

```
##
## Classification tree:
## snip.tree(tree = fit3, nodes = c(581L, 17L, 577L, 79L, 37L, 77L,
## 576L, 153L, 580L, 6L, 1157L, 16L, 5L, 1156L, 156L, 152L, 579L,
## 7L))
## Variables actually used in tree construction:
## [1] "poutcome" "month"    "contact"  "pdays"    "age"      "day"      "balance"
## [8] "housing"
## Number of terminal nodes:  21
## Residual mean deviance:  0.5706 = 10310 / 18060
## Misclassification error rate: 0.1041 = 1882 / 18084
```

```
#text(fit3, pretty=0)

#Estimate the confusion matrix, accuracy and F1 score for the test data by using the optimal model from step 3. C
omment whether the model has a good predictive power and which of the measures (accuracy or F1-score) should be p
referred here."

# Prune the decision tree model fit3 using the optimal number of leaves
opt_tree <- prune.tree(fit3, best=optimal_leaves)

# Make predictions on the test set using the pruned tree
ffitTest <- predict(opt_tree, newdata=test, type="class")

# Calculate the confusion matrix
c_m <- table(test$y, ffitTest)

# Print the confusion matrix
c_m
```

```
##      ffitTest
##          no   yes
##   no  11812   167
##   yes  1294   291
```

```
# Calculate the accuracy
acc <- sum(c_m[1], c_m[4])/sum(c_m[1:4])

# Calculate the precision
prec <- c_m[4] / sum(c_m[4], c_m[2])

# Calculate the recall
recall <- c_m[4] / sum(c_m[4], c_m[3])

# Calculate the F1 score
f1_score <- (2*(recall*prec)) / (recall + prec)

#Perform a decision tree classification of the test data with the following loss matrix, 732A99/732A68/ TDDE01 Ma
chine Learning Division of Statistics and Machine Learning Department of Computer and Information Science and rep
ort the confusion matrix for the test data. Compare the results with the results from step 4 and discuss how the
rates has changed and why."

# Make predictions on the test set using the pruned tree model
predtree5 <- predict(opt_tree, newdata=test, type="vector")

# Extract the probability of the positive class from the predictions
probY <- predtree5[,2]

# Extract the probability of the negative class from the predictions
probN <- predtree5[,1]

# Create a new vector of predictions based on the ratio of the probabilities
pred5 <- ifelse((probY/probN)>1/5, "yes", "no")

# Calculate the confusion matrix for the new predictions
c_m <- table(test$y, pred5)

# Print the confusion matrix
c_m
```

```
##      pred5
##          no   yes
##   no  11030   949
##   yes   771   814
```

```r
# Calculate the accuracy
acc <- sum(c_m[1], c_m[4])/sum(c_m[1:4])

# Calculate the precision
prec <- c_m[4] / sum(c_m[4], c_m[2])

# Calculate the recall
recall <- c_m[4] / sum(c_m[4], c_m[3])

# Calculate the F1 score
f1_score <- (2*(recall*prec)) / (recall + prec)


#Use the optimal tree and a logistic regression model to classify the test data by using the following principle:
where 𝜋𝜋 = 0.05,0.1,0.15, … 0.9,0.95. Compute the TPR and FPR values for the two models and plot the correspondi
ng ROC curves. Conclusion?  Why precisionrecall curve could be a better option here?"

# Train a decision tree model on the training data
optimalTree <- tree(as.factor(y)~., data=train, mindev=0.0005)

# Prune the tree using the optimal number of leaves
optimalTree <- prune.tree(optimalTree, best=21)

# Create a sequence of probability thresholds
pi <- seq(0.05, 0.95, 0.05)

# Train a logistic regression model on the training data
logic_model <- glm(as.factor(y)~.,data = train ,family="binomial")

# Make predictions on the test set using the logistic regression model
pred6_probY = predict(logic_model, newdata = test, type = "response")

# Calculate the probability of the negative class
pred6_probN = 1 -pred6_probY

# Make predictions on the test set using the pruned decision tree model
tree_pred = predict(optimalTree, newdata = test, type = "vector")

# Initialize vectors for false positive rate and true positive rate for both models
fpr_1 <- c(1:length(pi))
tpr_1 <- c(1:length(pi))
fpr_2 <- c(1:length(pi))
tpr_2 <- c(1:length(pi))

# Initialize the predictions for the decision tree model
pred6 <- tree_pred[, 1]

# Loop through each probability threshold
for (i in 1:length(pi)){

  # Decision tree model
  tpr_1[i] = 0
  fpr_1[i] = 0

  # Create predictions for the decision tree model based on the current threshold
  pred6 <- ifelse(tree_pred[,2]>pi[i], "yes", "no")

  # Calculate the confusion matrix for the decision tree model
  pred6_matrix <- table(test$y, pred6)

  # If the confusion matrix has more than one column, calculate TPR and FPR
  if(ncol(pred6_matrix) > 1){
    tpr_1[i] <- pred6_matrix[2,2] / (pred6_matrix[2,1]+pred6_matrix[2,2])
    fpr_1[i] <- pred6_matrix[1,2] / (pred6_matrix[1,1]+pred6_matrix[1,2])
  }

  # Logistic regression model
  tpr_2[i] = 0
  fpr_2[i] = 0

  # Create predictions for the logistic regression model based on the current threshold
  pred6_logic <- ifelse(pred6_probY > pi[i], "yes", "no")

  # Calculate the confusion matrix for the logistic regression model
  pred6_logic_matrix <- table(test$y,pred6_logic)

  # Calculate TPR and FPR for the logistic regression model
  tpr_2[i] <- pred6_logic_matrix[2,2] /(pred6_logic_matrix[2,1] +pred6_logic_matrix[2,2])
```

```
  fpr_2[i] <- (pred6_logic_matrix[1,2] / (pred6_logic_matrix[1,1]
                                          +pred6_logic_matrix[1,2]))
}

# Plot the TPR and FPR for the decision tree model
plot(fpr_1,tpr_1, type='l',xlim = c(0,1), ylim = c(0,1),
     xlab='FPR', ylab='TPR', col='red')

# Plot the TPR and FPR for the logistic regression model
lines(fpr_2,tpr_2, type='l',xlim = c(0,1),
      xlab='FPR', ylab='TPR', col='blue')

# Add a reference line to the plot
abline(0,1, lty = 5)

# Add a legend to the plot
legend(x = "bottomright" , col = c("red", "blue", "black"),
       legend = c("Tree","Logistic", "Reference"), lwd = 2,title = "Lines",
       lty = c(1,1,5))
```

```
#Scale all variables except of ViolentCrimesPerPop and implement PCA by using function eigen(). Report how many c
omponents are needed to obtain at least 95% of variance in the data. What is the proportion of variation explaine
d by each of the first two principal components? "

  rm(list = ls(all = TRUE))
  graphics.off()
  shell("cls")
  # Read in the communities data from a CSV file
  data = read.csv(file = "communities.csv", header = TRUE)

  # Create a logical vector indicating which column is "ViolentCrimesPerPop"
  index <- names(data) %in% "ViolentCrimesPerPop"

  # Scale the data, excluding "ViolentCrimesPerPop"
  data.scaled <- scale(x = data[, !index], center = TRUE, scale = TRUE)

  # Calculate the eigenvalues and eigenvectors of the covariance matrix of the original data
  e = eigen(cov(data[, -1]))

  # Calculate the eigenvalues and eigenvectors of the covariance matrix of the scaled data
  e.scaled = eigen(cov(data.scaled))

  # Calculate the cumulative sum of the scaled eigenvalues
  cum_var = cumsum(e.scaled$values/sum(e.scaled$values))

  # Calculate the number of eigenvalues needed to explain 95% of the variance
  sum(cum_var<0.95)
```

```
## [1] 34
```

```
  # Calculate the proportion of the variance explained by the first two eigenvalues
  e.scaled$values[1:2]/sum(e.scaled$values)
```

```
## [1] 0.2501699 0.1693597
```

```
#Repeat PCA analysis by using princomp() function and make the trace plot of the first principle component. Do ma
ny features have a notable contribution to this component? Report which 5 features contribute mostly (by the abso
lute value) to the first principle component. Comment whether these features have anything in common and whether
they may have a logical relationship to the crime level. Also provide a plot of the PC scores in the coordinates
(PC1, PC2) in which the color of the points is given by ViolentCrimesPerPop. Analyse this plot (hint: use ggplot2
package )."

  # Read in the communities data from a CSV file
  data = read.csv(file = "communities.csv", header = TRUE)

  # Create a logical vector indicating which column is "ViolentCrimesPerPop"
  index <- names(data) %in% "ViolentCrimesPerPop"

  # Scale the data, excluding "ViolentCrimesPerPop"
  data.scaled <- scale(x = data[, !index], center = TRUE, scale = TRUE)

  # Perform principal component analysis on the scaled data
  pr=princomp(data.scaled)

  # Calculate the eigenvalues
  lambda=pr$sdev^2

  # Calculate the proportion of variance explained by each eigenvalue
  var = sprintf("%2.3f",lambda/sum(lambda)*100)

  # Extract the loadings for the first principal component
  ev1 = pr$loadings[,1]

  # Find the top five absolute loadings for the first principal component
  ev1[order(abs(ev1),decreasing = TRUE)[1:5]]
```

```
##      medFamInc      medIncome    PctKids2Par     pctWInvInc PctPopUnderPov
##     -0.1833080     -0.1819830     -0.1755423     -0.1748683      0.1737978
```

```
  # Load the ggfortify library for plot visualization
  library(ggfortify)

  # Create a scatterplot matrix of the first two principal components, colored by "ViolentCrimesPerPop"
  autoplot(pr, data = data, colour = "ViolentCrimesPerPop")

#Split the original data into training and test (50/50) and scale both features and response appropriately, and e
stimate a linear regression model from training data in which ViolentCrimesPerPop is target and all other data co
lumns are features. Compute training and test errors for these data and comment on the quality of model"

  # Read in the communities data from a CSV file
  df = read.csv("communities.csv")

  # Scale the data
  df = scale(df, TRUE, TRUE)

  # Set a seed for reproducibility
  set.seed(12345)

  # Split the data into a training set and a test set
  n <- dim(df)[1]
  id <- sample(1:n,floor(n*0.5))
  df_train <- data.frame(df[id,])
  df_test <- data.frame(df[-id,])

  # Fit a linear regression model to the training data
  lr = lm(ViolentCrimesPerPop ~ .,df_train)

  # Make predictions on the training and test sets
  train.pred = predict(lr, df_train)
  test.pred = predict(lr, df_test)

  # Calculate the mean squared error (MSE) for the training and test sets
  train_MSE = mean((train.pred - df_train$ViolentCrimesPerPop) ^ 2)
  test_MSE = mean((test.pred - df_test$ViolentCrimesPerPop) ^ 2)

  # Print the MSE for the training and test sets
  print("Train error")
```

```
## [1] "Train error"
```

```
  train_MSE
```

```
## [1] 0.2591772
```

```
  print("Test error")
```

```
## [1] "Test error"
```

```
  test_MSE
```

```
## [1] 0.4000579
```

```
#Implement a function that depends on parameter vector θθ and represents the cost function for linear regression
without intercept on the training data set. Afterwards, use BFGS method (optim() function without gradient specif
ied) to optimize this cost with starting point θθ0 = 0 and compute training and test errors for every iteration n
umber. Present a plot showing dependence of both errors on the iteration number and comment which iteration numbe
r is optimal according to the early stopping criterion. Compute the training and test error in the optimal model,
compare them with results in step 3 and make conclusions. a. Hint 1: don’t store parameters from each iteration (
otherwise it will take a lot of memory), instead compute and store test errors directly. b. Hint 2: discard some
amount of initial iterations, like 500, in your plot to make the dependences visible."

  # Initialize empty vectors for storing training and test errors
  train_error <<- numeric(0)
  test_error <<- numeric(0)

  # Set the seed for reproducibility
  set.seed(12345)

  # Define the cost function
  cost <- function(theta, train, acc_train, test, acc_test){
    # Calculate predictions on the training and test data using theta
    pred_train = train %*% theta
    pred_test = test %*% theta

    # Calculate MSE of predictions on training and test data
    mse_train = mean((acc_train-pred_train)^2)
    mse_test = mean((acc_test - pred_test)^2)

    # Append MSEs to the train_error and test_error vectors
    train_error <<- append(train_error,mse_train)
    test_error <<- append(test_error,mse_test)

    # Return MSE on training data as cost
    return(mse_train)
  }

  # Create matrices of predictors and responses for the training and test data
  trainy = as.matrix(df_train[,1:(dim(df_train)[2]-1)])
  acc_train = as.matrix(df_train['ViolentCrimesPerPop'])
  testy = as.matrix(df_test[,1:(dim(df_test)[2]-1)])
  acc_test = as.matrix(df_test['ViolentCrimesPerPop'])

  # Initialize theta as a matrix of all 0s with the same number of columns as trainy
  theta = numeric(dim(trainy)[2])
  theta = as.matrix(theta)

  # Use the optim function to find the optimal value of theta
  opt = optim(par=theta, fn=cost, train = trainy, acc_train = acc_train,
              test=testy, acc_test=acc_test, method = "BFGS")

  # Extract the optimal value of theta from the optim function output
  opt_theta = opt$par

  # Calculate the MSE of the predictions made using opt_theta on the training and test data
  train_opt_error = opt$value
  test_opt_error = mean((acc_test - (testy %*% opt_theta))^2)

  # Print the MSEs
  print("calculated optimal train")
```

```
## [1] "calculated optimal train"
```

```
  train_opt_error
```

```
## [1] 0.2592247
```

```
  print("Lm train error")
```

```
## [1] "Lm train error"
```

```
  train_MSE
```

```
## [1] 0.2591772
```

```
  print("calculated optimal test")
```

```
## [1] "calculated optimal test"
```

```
  test_opt_error
```

```
## [1] 0.3997238
```

```
  print("Lm test error")
```

```
## [1] "Lm test error"
```

```
  test_MSE
```

```
## [1] 0.4000579
```

```
  # Create a logical vector indicating which elements of train_error and test_error should be plotted
  excluded = c(TRUE,rep(FALSE,500))

  # Extract the relevant elements of train_error and test_error
  rest_train = train_error[excluded]
  rest_test = test_error[excluded]

  # Find the index of the minimum test error
  test_min_ind = which(test_error==min(test_error))

  print("Early stopping index and MSE")
```

```
## [1] "Early stopping index and MSE"
```

```
  test_min_ind
```

```
## [1] 2183
```

```
  min(test_error)
```

```
## [1] 0.3769468
```

```
  plot(rest_train, xlim=c(0,length(rest_train)), ylim=c(0,1.5), col = "blue")
  points(rest_test, col="red")

  # Add horizontal lines at the final MSE values for the training and test data
  lines(c(0,1000), rep(train_MSE, 2), col="blue")
  lines(c(0,1000), rep(test_MSE, 2), col="red")

  # Print the early stopping index and minimum test error
  print("Early stopping index and MSE")
```

```
## [1] "Early stopping index and MSE"
```

```
  test_min_ind
```

```
## [1] 2183
```

```
  min(test_error)
```

```
## [1] 0.3769468
```

lab3

2022-12-08

##Task 1

Implement a kernel method to predict the hourly temperatures for a date and place in Sweden. The forecast should consist of the predicted temperatures from 4 am to 24 pm in an interval of 2 hours.

#Setup

```
set.seed(1234567890)
library(geosphere)
stations <- read.csv("stations.csv")
temps <- read.csv("temps50k.csv")

date <- as.Date("2013-11-04") # The date to predict (up to the students)
filtered_temps <- temps[temps$date < as.Date(date),] #Filter the tempratures to discard irrelevant dates.No time traverller.

st <- merge(stations,filtered_temps,by="station_number")
# These three values are up to the students
h_distance <- 40000 #Unsure how much we want to consider stations further away.
h_date <- 9
h_time <- 3
a <- 58.4274 # The point to predict (up to the students) #latitud
b <- 14.826 #longitud
pos_vec <- cbind(b,a)

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", "24:00:00")
times_numbers <-c(4,6,8,10,12,14,16,18,20,22,24) #for the plot
temp <- vector(length=length(times)) # length for times: used in forloop.
```

● The first to account for the physical distance from a station to the point of interest. For this purpose, use the function distHaversine from the R package geosphere. ● The second to account for the distance between the day a temperature measurement was made and the day of interest. ● The third to account for the distance between the hour of the day a temperature measurement was made and the hour of interest.

```r
#Calculations for relative distances in location, date (day) and time (hours).
# Students' code here


#physical distance with distHaversine
st_loc<-cbind(st$longitude,st$latitude)
dist_hav<-distHaversine(p1=pos_vec,p2=st_loc) #Compare our pos with all the other pos
m<-cbind(dist_hav)

#Gaussian Kernel
#(x_* - x_i)/h from lectures
#diff represents (x_* - x_i)
gaussian_kernel<-function(diff, h_val) {

  u <- diff/h_val
  return(exp(-u*u))
}

#Relative day distance
relative_day_dist <- function(d1,d2){
  diff<- as.Date(d1) - as.Date(d2) #Difference between our date and the date we're comparing (in days).
  return (as.numeric(diff))
}

#relative hour distance, compare our time with another given time. Returns amount of hours.
relative_hour_dist <- function(time1, time2) {
  time_obj1 <-strptime(time1,format="%H:%M:%S") #Create time object so that we can extract hour
  time_obj2 <-strptime(time2,format="%H:%M:%S")
  h1<-as.integer(format(time_obj1,"%H")) #take the hour value as an integer
  h2<-as.integer(format(time_obj2,"%H"))
  # Convert hours to minutes
  minute1 <- h1 * 60
  minute2 <- h2 * 60

  # Compute the absolute difference in minutes
  minute_diff <- abs(minute1 - minute2)

  # Compute the relative distance in hours
  hour_diff <- minute_diff / 60

  return(hour_diff)
}
```

Use a kernel that is the sum of three Gaussian kernels:

Choose an appropriate smoothing coefficient or width for each of the three kernels above. No cross-validation should be used. Instead, choose manually a width that gives large kernel values to closer points and small values to distant points. Show this with a plot of the kernel value as a function of distance.

#Calculations with sum of kernels.

```
#Calculations
predictions = rep(0,11)
k_distance <- gaussian_kernel(dist_hav,h_distance) #kernel distance
k_days <- gaussian_kernel(relative_day_dist(date,filtered_temps$date),h_date) #kernel days



#Loop over the different times in a day.
for (i in 1:length(temp)) {
  rel_h<-relative_hour_dist(times[i],filtered_temps$time)

  #Relative time difference for each time specified in the assignment
  k_time <- gaussian_kernel(relative_hour_dist(times[i],filtered_temps$time),h_time)

  #Summation of kernels
  k_sum <- cbind(k_distance + k_days + k_time)

  # Normalize the values in the k_sum matrix by dividing each element by the sum of the elements in each row
  k_sum <- (k_sum/sum(k_sum))

  #get weighted temperatures
  weighted_temps <- k_sum * filtered_temps$air_temperature

  predictions[i] <- sum(weighted_temps)
}


#Used to look at h values
plot(rel_h,k_time,main = "h value for time")
```

```
plot(dist_hav,k_distance,xlim=c(0,100000),main = "h value for distance")
```

```
plot(relative_day_dist(date,filtered_temps$date),k_days,xlim=c(0,100),main = "h value for days")
```

The different plots for h-values has been given the following h values: h_distance <- 40000 h_date <- 9 h_time <- 3 This has given us plots that look reasonably good, since closer points are considered more, while points further away will be less and less considered.

```
#sum of kernel plot
plot(times_numbers,predictions,type="o",xlab = "Time of day (hours)",ylab = "Predicted temp",main = "Prediction o
f temperature using summation")
```

These values look very reasonable. Looking at the curve going from colder at the earlier and later hours, while having higher temperatures at the middle of the day. November of 2013 was a very cold month in general, so having a prediction of +6 C is not unreasonable.

Finally, repeat the exercise above by combining the three kernels into one by multiplying them, instead of summing them up. Compare the results obtained in both cases and elaborate on why they may differ.

```
predictions2 = rep(0,11)
k_distance <- gaussian_kernel(dist_hav,h_distance) #kernel distance
k_days <- gaussian_kernel(relative_day_dist(date,filtered_temps$date),h_date) #kernel days



for (i in 1:length(temp)) {
  rel_h<-relative_hour_dist(times[i],filtered_temps$time)

  #Relative time difference for each time specified in the assignment
  k_time <- gaussian_kernel(relative_hour_dist(times[i],filtered_temps$time),h_time)

  #Multiply kernels
  k_sum <- cbind(k_distance * k_days * k_time)

  # Normalize the values in the k_sum matrix by dividing each element by the sum of the elements in each row
  k_sum <- (k_sum/sum(k_sum))

  #get weighted temperatures
  weighted_temps <- k_sum * filtered_temps$air_temperature

  predictions2[i] <- sum(weighted_temps)


}

plot(times_numbers,predictions2,type="o",xlab = "Time of day (hours)",ylab = "Predicted temp",main = "Prediction
of temperature using multiplication")
```

This graph looks a bit more unclear. It's difficult to interpret if there's something wrong with how the data has been divided when multiplying, since the curve is not distributed like a bell curve.

The reason for this graph being less "well distributed" is because the multiplied kernels will give more weight to the higher elements in the kernels, while suppressing the lower ones. This means that values that are not very close to the points we're looking at will not be considered nearly as much (even if they're already not being considered much).

The summed kernel graph gives equal weights to all values which will give our predictions more "real" data to use, since values a bit further away from our initial ones will still be considered. In this case these values are very relevant, since weather predictions (more specifically, temperature predictions) require a lot of data, which should be distributed over time (and not only looking at the exact same day/time/location as what we're trying to predict).

##Task 2

#Error estimates:

```
# Load the kernlab library
library(kernlab)

# Set seed for reproducibility
set.seed(1234567890)

# Load the spam data set
data(spam)

# Shuffle the rows of the data set
foo <- sample(nrow(spam))
spam <- spam[foo,]

# Scale the features (except for the target column)
spam[,-58] <- scale(spam[,-58])

# Split the data into training, validation, and test sets
tr <- spam[1:3000, ]
va <- spam[3001:3800, ]
trva <- spam[1:3800, ]
te <- spam[3801:4601, ]

# Set the step size for the loop
by <- 0.3

# Initialize a vector to store the validation errors
err_va <- NULL

# Loop over different values of the regularization parameter C
for(i in seq(by,5,by)){
  # Train a SVM classifier with the RBF kernel and the current value of C
  filter <- ksvm(type~., data=tr, kernel="rbfdot", kpar=list(sigma=0.05), C=i, scaled=FALSE)

  # Make predictions on the validation set
  mailtype <- predict(filter, va[,-58])

  # Calculate the error on the validation set
  t <- table(mailtype, va[,58])
  err_va <- c(err_va, (t[1,2]+t[2,1])/sum(t))
}

# Find the value of C that achieved the lowest validation error
best_C <- which.min(err_va) * by

# Retrain the classifier with the optimal value of C
filter0 <- ksvm(type~., data=tr, kernel="rbfdot", kpar=list(sigma=0.05), C=best_C, scaled=FALSE)

# Evaluate the error on the validation set
mailtype <- predict(filter0, va[,-58])
t <- table(mailtype, va[,58])
err0 <- (t[1,2]+t[2,1])/sum(t)
print("Error 0")
```

```
## [1] "Error 0"
```

```
err0
```

```
## [1] 0.0675
```

```
# Retrain the classifier on the training set and evaluate the error on the test set
filter1 <- ksvm(type~., data=tr, kernel="rbfdot", kpar=list(sigma=0.05), C=best_C, scaled=FALSE)
mailtype <- predict(filter1, te[,-58])
t <- table(mailtype, te[,58])
err1 <- (t[1,2]+t[2,1])/sum(t)
print("Error 1")
```

```
## [1] "Error 1"
```

```
err1
```

```
## [1] 0.08489388
```

```
# Retrain the classifier on the combined training and validation sets and evaluate the error on the test set
filter2 <- ksvm(type~., data=trva, kernel="rbfdot", kpar=list(sigma=0.05), C=best_C, scaled=FALSE)
mailtype <- predict(filter2, te[,-58])
t <- table(mailtype, te[,58])
err2 <- (t[1,2]+t[2,1])/sum(t)
print("Error 2")
```

```
## [1] "Error 2"
```

```
err2
```

```
## [1] 0.082397
```

```
# Retrain the classifier on the entire data set and evaluate the error on the test set
filter3 <- ksvm(type~., data=spam, kernel="rbfdot", kpar=list(sigma=0.05), C=best_C, scaled=FALSE)
mailtype <- predict(filter3, te[,-58])
t <- table(mailtype, te[,58])
err3 <- (t[1,2]+t[2,1])/sum(t)

print("Error 3")
```

```
## [1] "Error 3"
```

```
err3
```

```
## [1] 0.02122347
```

Questions

1. Which filter do we return to the user ? filter0, filter1, filter2 or filter3? Why?

The filter we return is filter1. Because the SVM model is trained on trianing data (tr) and validated on validation data (va). Then we should use the optimal err_va to make prediction on the test data. Using ksvm on training data and not any other. In general you do your training on your training set, evalutate its performance on the validation set, and then use the best model to predict on the test dataset.

New: The filter we should return is the filter trained on the biggest dataset which is filter3.

2. What is the estimate of the generalization error of the filter returned to the user? err0, err1, err2 or err3? Why?

it is err1 = 0.08489388. Filter0 gives low error estimate due to the fact that it is filtered on validation, the same way we choose the optimal err_va. Filter3 gives the lowest error estimate because of the the model is trained on the same data it is predicted on. As for filter1 and filter2, they give similar error estimates. Filter2 error rate is slightly lower due to the fact that it is trained on a larger data set (training and validation), where filter1 is only trained on training data.

New: filter 2 gives an error of 0.082397, this is the nearest estimate we can get of the filter (filter3) returned to the user. Filter3 is expected to be a bit better because of more training data. If we knew that the user wanted to be able to get an error estimate, then it could be worth to instead return filter2.

3. Implementation of SVM predictions. Once a SVM has been fitted to the training data, a new point is essentially classified according to the sign of a linear combination of the kernel function values between the support vectors and the new point. You are asked to implement this linear combination for filter3. You should make use of the functions alphaindex, coef and b that return the indexes of the support vectors, the linear coefficients for the support vectors, and the negative intercept of the linear combination. See the help file of the kernlab package for more information. You can check if your results are correct by comparing them with the output of the function predict where you set type = “decision”. Do so for the first 10 points in the spam dataset. Feel free to use the template provided in the Lab3Block1 2021 SVMs St.R file.

```
# Get the indices of the support vectors in the SVM classifier
sv <- alphaindex(filter3)[[1]]

# Get the coefficients of the support vectors in the SVM classifier
co <- coef(filter3)[[1]]

# Get the bias term of the SVM classifier and negate it
inte <- -b(filter3)

# Create an RBF kernel with a standard deviation of 0.05
rbfkernel <- rbfdot(sigma = 0.05)

# Initialize an empty vector to store the predicted values
k <- c()

# Loop over the first 10 rows of the spam dataset
for(i in 1:10) {

  # Initialize a variable to store the predicted value for the current row
  k2 <- 0

  # Loop over each support vector
  for(j in 1:length(sv)) {

    # Apply the RBF kernel to the jth support vector and the ith row of the spam dataset. We unlist the matrixes
to perform the calculations.
    f <- rbfkernel(unlist(spam[sv[j], -58]), unlist(spam[i, -58]))

    # Update the predicted value for the current row by adding the product of the jth coefficient and the value o
f the kernel. F is a 1x1 matrix.
    k2 <- k2 + co[j] * f[1]
  }

  # Append the predicted value for the current row to the k vector
  k <- c(k, k2 + inte)
}

# Print the k vector
k
```

```
##  [1] -1.998999  1.560584  1.000278 -1.756815 -2.669577  1.291312 -1.068444
##  [8] -1.312493  1.000184 -2.208639
```

```
# Use the predict function to make predictions on the first 10 rows of the spam dataset using the trained SVM cla
ssifier
prediction = predict(filter3, spam[1:10, -58], type = "decision")

plot(k, col = "red")
lines(prediction)
```

The red dotted points is the predicted values that generated from the linear combination for filter3. The lines is the predictions from the predict function.

Task 3

1:

```
library(neuralnet)
set.seed(1234567890)

# Generate 500 random values between 0 and 10
Var <- runif(500, 0, 10)
mydata <- data.frame(Var, Sin=sin(Var))
tr <- mydata[1:25,] # Training
te <- mydata[26:500,] # Test
winit = runif(31, -1,1) #weights: one for each of the 10 hidden nodes, plus one for the bias term for each of the
10 hidden nodes,
                        #plus one for the output node, plus one for the bias term for the output node

# Fit the neural network using the training data
# The neural network has 10 hidden nodes and uses the initial weights stored in winit
nn <- neuralnet(Sin ~ Var, data = tr,
                hidden = 10, startweights = winit)

# Plot of the training data (black), test data (blue), and predictions (red)
plot(tr, cex=2)
points(te, col = "blue", cex=1)
points(te[,1],predict(nn,te), col="red", cex=1)
```

Question: Train a neural network to learn the trigonometric sine function. To do so, sample 500 points uniformly at random in the interval [0,10]. Apply the sine function to each point. The resulting value pairs are the data points available to you. Use 25 of the 500 points for training and the rest for test. Use one hidden layer with 10 hidden units. You do not need to apply early stopping. Plot the training and test data, and the predictions of the learned NN on the test data. You should get good results. Comment your results.

Answer: The predictions seems pretty accurate. We can se that there is less training data around the points where it kind of diverges from the testdata. this is reasonoble and it then finds its way back.

2:

```
library(neuralnet)

# Set seed for reproducibility
set.seed(1234567890)

# Generate data
Var <- runif(500, 0, 10)
mydata <- data.frame(Var, Sin=sin(Var))
tr <- mydata[1:25,] # Training
te <- mydata[26:500,] # Test
winit = runif(31, -1,1)

# Define custom activation functions
h1 <- function(x) x
h2 <- function(x) ifelse(x>0,x,0)
h3 <- function(x) log(1 + exp(x))

# Train neural network with custom activation functions (h1)
nn1 <- neuralnet(Sin ~ Var, data = tr,
                hidden = 10, act.fct = h1, startweights = winit)

# Train neural network with custom activation functions (h2)
nn2 <- neuralnet(Sin ~ Var, data = tr,
                hidden = 10, act.fct = h2, startweights = winit)

# Train neural network with custom activation functions (h3)
nn3 <- neuralnet(Sin ~ Var, data = tr,
                hidden = 10, act.fct = h3, startweights = winit)

# Plot results
plot(tr, cex=3, ylim=c(-1.5, 1.5))
points(te, col = "blue", cex=2)
points(te[,1],predict(nn1,te), col="red", cex=1)
points(te[,1],predict(nn2,te), col="green", cex=1)
points(te[,1],predict(nn3,te), col="pink", cex=1)
legend(1, -0.5, legend=c("train", "test","linear", "ReLu", "Softplus"),
       col=c("black","blue","red", "green", "pink"), lty=1:2, cex=0.8)
```

Question: In question (1), you used the default logistic (a.k.a. sigmoid) activation function, i.e. act.fct = “logistic”. Repeat question (1) with the following custom activation functions: h1(x) = x, h2(x) = max{0, x} and h3(x) = ln(1 + exp x) (a.k.a. linear, ReLU and softplus). See the help file of the neuralnet package to learn how to use custom activation functions. Plot and comment your results.

Answer: The third custom activation softplus seems to fit the best. while the first, linear is very bad and the second, ReLu manages predict some values.

3:

```
library(neuralnet)
set.seed(1234567890)
Var <- runif(500, 0, 50)

# Create a data frame with two columns: Var and Sin
# Sin contains the sine of the values in Var
mydata <- data.frame(Var, Sin=sin(Var))


# Plot of the training data (black), test data (blue), and predictions (red)

# Use the nn neural network to make predictions on mydata
prediction = predict(nn,mydata)

plot(mydata, col = "blue", cex=1, ylim=c(-15, 2), xlim=c(0, 55))
points(mydata[,1],prediction, col="red", cex=1, )
```

```
# Find the index of the minimum value in the prediction vector
smallestIndex = which.min(prediction)

# Retrieve the minimum value of the prediction vector using the index
smallestvalue = prediction[smallestIndex]

weights = nn$weights

weights
```

```
## [[1]]
## [[1]][[1]]
##            [,1]       [,2]       [,3]       [,4]      [,5]       [,6]      [,7]
## [1,] -11.870831 -0.9153178 -3.0316259  1.5313105  6.555437 -11.768525  1.397325
## [2,]   4.042494 -0.5368677  0.2603355 -0.5487656 -2.233839   1.787862 -1.259834
##            [,8]       [,9]       [,10]
## [1,]  0.2032402  0.2948804 -0.21326487
## [2,] -2.1974036 -0.5886005 -0.03177278
##
## [[1]][[2]]
##              [,1]
##  [1,]  -0.1088032
##  [2,]   0.7716196
##  [3,]  -1.0436754
##  [4,] -16.0529801
##  [5,]  -1.2718986
##  [6,]   3.2805756
##  [7,]   4.3076898
##  [8,]   0.1220703
##  [9,]  -1.2234185
## [10,]  -2.7618205
## [11,]   2.3696931
```

```
smallestvalue
```

```
## [1] -10.74471
```

Question: Sample 500 points uniformly at random in the interval [0,50], and apply the sine function to each point. Use the NN learned in question (1) to predict the sine function value for these new 500 points. You should get mixed results. Plot and comment your results.

Answer: The predictions seem to still be accurate to the data up to Var 10, and after that it converges to -10.74471 up to Var=50.

4:

Question: In question (3), the predictions seem to converge to some value. Explain why this happens. To answer this question, you may need to get access to the weights of the NN learned. You can do it by running nn or nn$weights where nn is the NN learned.

Answer: The predictions seem to still be accurate to the data up to Var 10, and after that it converges to -10.74471 at Var=50. The nn is only trained of var values between 0-10 but attempts to predict 0-50. Same for the hidden units. Not sure why it converges to exactly -10.74.

5:

```
# Sample 500 points uniformly at random in the interval [0,10]
Var <- runif(500, 0, 10)

# Apply the sine function to each point
mydata <- data.frame(Var, Sin=sin(Var))
otherWayData <- data.frame(Sin=sin(Var), Var)

# Generate 31 random numbers between 0 and 10
winit = runif(31, 0,10)


# Use all the points in mydata as training points for a neural network
# The target variable is set to be Var instead of Sin
nn <- neuralnet(Var ~ Sin, data = mydata, hidden = 10, threshold = 0.1, startweights = winit )

# Plot the predictions of the neural network
plot(te, col = "blue", cex=1,ylim=c(-10, 10), xlim=c(0, 10))
points(tr[,1],predict(nn,tr), col="red", cex=1)
```

```
plot(otherWayData,col = "blue", cex=1) #kanske borde plotta enligt gamla training data istället?
points(otherWayData[,1],predict(nn,otherWayData), col="red", cex=1)
```

Question: Sample 500 points uniformly at random in the interval [0,10], and apply the sine function to each point. Use all these points as training points for learning a NN that tries to predict x from sin(x), i.e. unlike before when the goal was to predict sin(x) from x. Use the learned NN to predict the training data. You should get bad results. Plot and comment your results. Help: Some people get a convergence error in this question. It can be solved by stopping the training before reaching convergence by setting threshold = 0.1.

Answer: Looks very bad, it is reasonable because we are trying to map from multiple values and not a clear function. One sin value corresponds to multiple var values which makes it impossible to predict. It kinda finds an avarage.