Lab 2 TDDE01

Daniel Kouznetsov (danko376), Liv Kåreborn (livka967), Erik Johansson (erijo073)

2022-12-04

Contributions

Assignment 1: Erik

Assignment 2: Daniel

Assignment 3: Liv

Assignment 1 - Explicit regularization

1.1

The underlying probabilistic model for the linear regression model is: $p(\beta) = \mathcal{N}(\beta|\mu_0, \Sigma_0)$

```
# Loads tecator but only the Channels 1-100 and Fat as that is the only data needed for this task.
data1 <- read.csv("tecator.csv")[,2:102]

# Divides 50% of data randomly into training data.
n=dim(data1)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train1 <- data1[id,]

# Puts the remaining data into test data.
id1=setdiff(1:n, id)
test1 <- data1[id1,]

# Creates a linear regression model for the training data using the channels as features and Fat as a t
model_linear <- lm(train1$Fat ~., data = train1)
#summary(model_linear)

# Makes predictions on the training data and test data using the model.
train1_prediction <- predict(model_linear,train1)
test1_prediction <- predict(model_linear,test1)

# Calculates the difference of predicted values against true values.
train1_diff <- train1_prediction - train1$Fat
test1_diff <- test1_prediction - test1$Fat

# Computes the Mean Square Errors of the predicted data.
train1_MSE <- sum(train1_diff^2)/dim(train1)[1]
test1_MSE <- sum(test1_diff^2)/dim(test1)[1]
```

```
# Prints the MSE data.
cat("Training data MSE: ", train1_MSE, "\n")
```

```
## Training data MSE:  0.005709117
```

```
cat("Test data MSE: ", test1_MSE, "\n")
```

```
## Test data MSE:  722.4294
```

The model summary shows that the model predictions will be heavily influenced by any change in the input data and is therefor a bad fit. The model predicts well on the training data but performs poorly on the test data. This shows that the model is not very good.

2.2

The cost function that should be optimized here is:

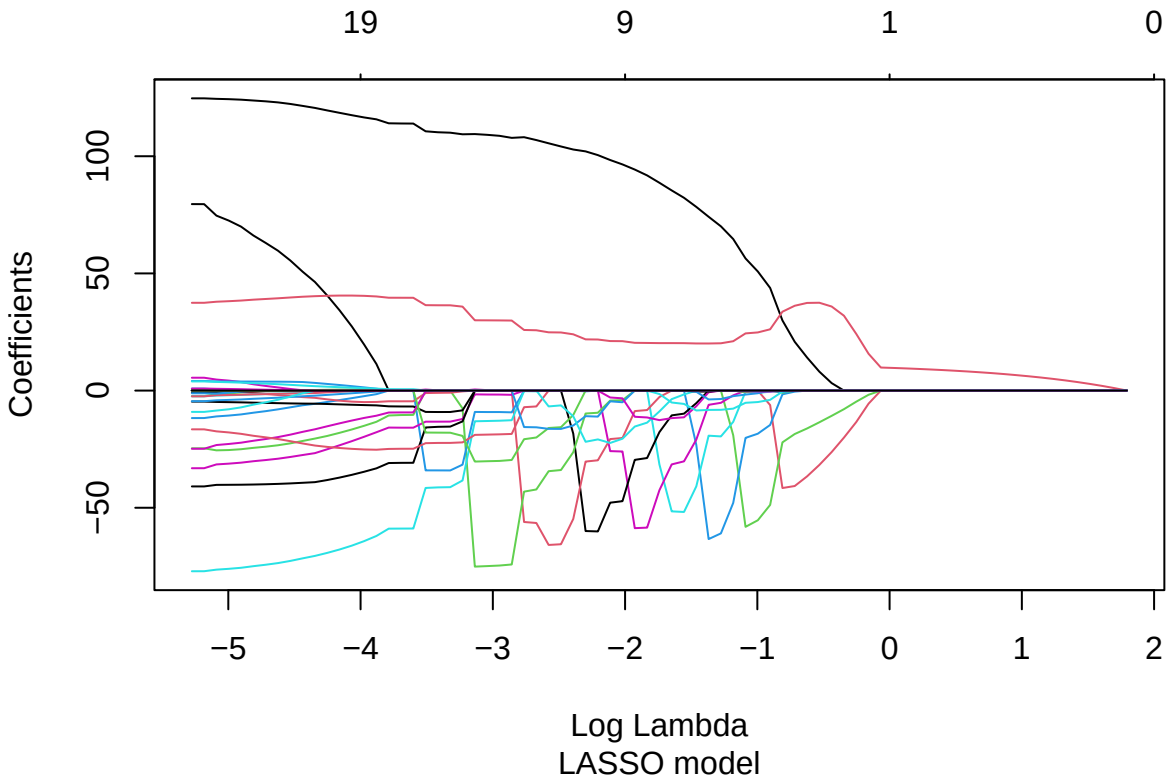
$$\hat{\theta}^{lasso} = argmin \left\{ \frac{1}{n} \sum_{i=1}^{n} (y_i - \theta_0 - \theta_1 x_1 j - ... \theta_p x_{pj})^2 + \lambda \sum_{j=1}^{p} |\theta_i| \right\}$$

```
library(glmnet)
```

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

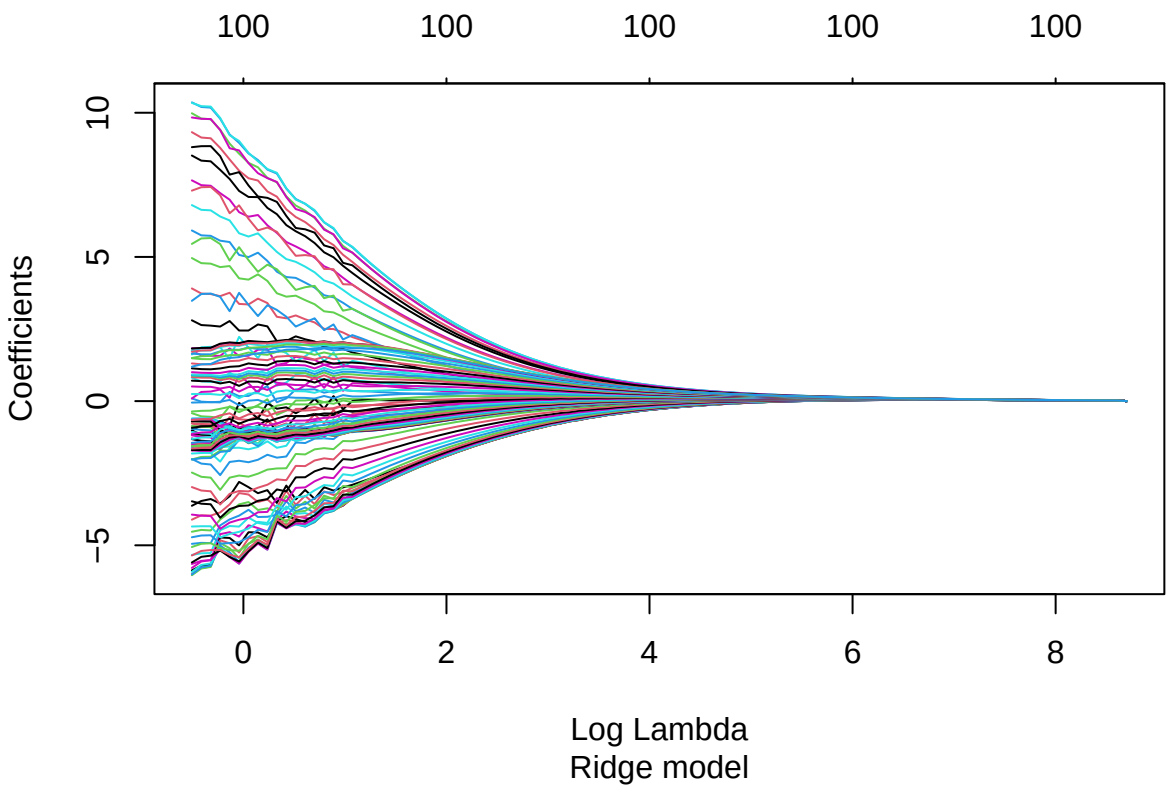
```
## Loaded glmnet 4.1-6
```

```
# X is made a data matrix so that it works later with cross validation.
x <- data.matrix(train1[,1:100])
y <- train1$Fat

# Creates a LASSO model using the first 100 columns as x and the Fat values as y, with a penalty factor
model_LASSO <- glmnet(x, y, alpha = 1, family = 'gaussian')

#summary(model_LASSO)
plot(model_LASSO, xvar = "lambda", sub = "LASSO model")
```



The plot shows that as the penalty factor increases the amount of feature coefficients that go to zero increases. To select a model with only three features a penalty factor of $e^{-0.2}$ can be chosen.

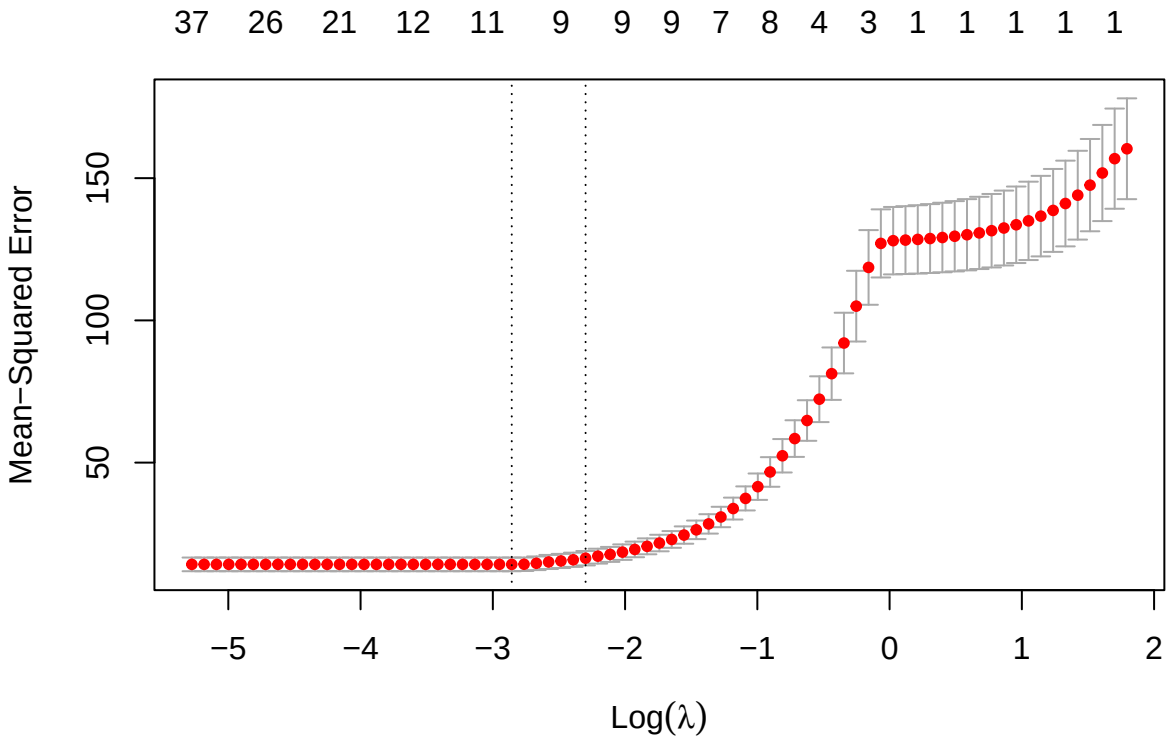
```
# Creates a Ridge model using glmnet by having alpha = 0.
model_ridge <- glmnet(x, y, alpha = 0, family = 'gaussian')
plot(model_ridge, xvar = "lambda", sub = "Ridge model")
```



Comparing the LASSO model and Ridge model plots it shows that the ridge model have lower coefficients and goes who also go towards zero smoother than the LASSO model. However the ridge models coefficients does not go to zero exactly, only approaches zero.

The conclusion is that the LASSO model will make a prediction using only some of the features if the lambda is high enough while a ridge model would still calculate all the features even if their coefficient is close to zero.

```
# Creates a cross validated LASSO model.
model_cvLasso <- cv.glmnet(x, y, alpha = 1, family = 'gaussian')

# Plots the models MSE with varying lambda
plot(model_cvLasso)
```



As the above plot shows as lambda increases the MSE of the model will also increase significantly after $\log \lambda \approx -2.2$. The MSE stops increasing as much around $\log \lambda \approx 0$ and then it continues to increase more and more.

```
# Takes the optimal lambda from the model and prints it.
optimal_lambda <- model_cvLasso$lambda.min
cat("Optimal lambda: ", optimal_lambda)
```

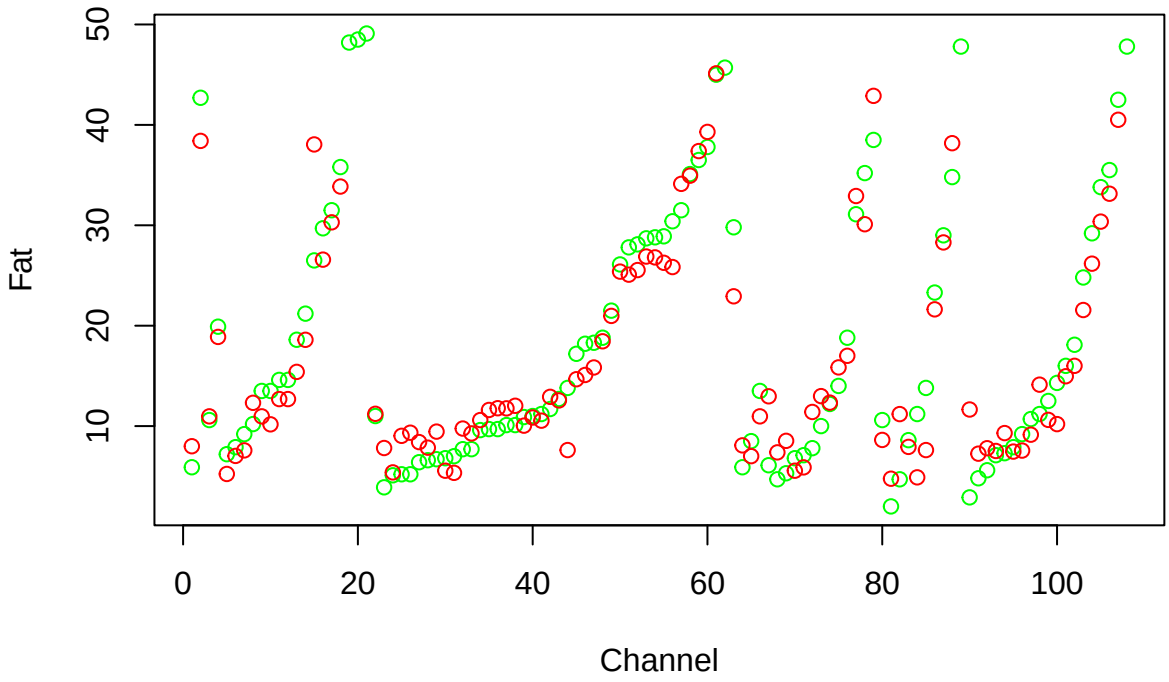
```
## Optimal lambda:  0.05744535
```

```
# Trains a LASSO model using the optimal lambda.
model_Lasso_optimal <- glmnet(x, y, alpha = 1, family = 'gaussian', lambda = optimal_lambda)
#coef(model_Lasso_optimal)
```

The LASSO model using the optimal lambda only uses 7 of the 100 variables.

This optimal λ gives $\log 0.05744535 \approx -2.9$. Comparing the optimal $\log \lambda$ to the plot it shows that it is the greatest penalty value before the MSE starts to increase. According to the plot there does not seem to be a significant difference between the optimal lambda and $\log \lambda = -4$

```
x_test <- data.matrix(test1[,1:100])
y_test <- test1$Fat

# Predicts the values using the model using the test data.
predictLasso <- predict(model_Lasso_optimal, s = optimal_lambda, newx = x_test)

# Plots the true data and the predicted data
plot(y_test, col = "green", xlab = "Channel", ylab = "Fat")
points(predictLasso, col = "red")
```




```
# Calculates the MSE of the prediction
prediction_diff <- predictLasso - y_test
prediction_MSE <- sum(prediction_diff^2)/dim(test1)[1]

# Prints the MSE
cat("Prediction MSE: ", prediction_MSE)
```

```
## Prediction MSE:  13.2998
```

The plot above shows the predicted values with the optimal lambda as red points and the true values as green points.

The model predictions are close to the true values and the MSE is ≈ 13.3 which, given the scale of values from 0 to 50 can be a significant error. The LASSO model with optimal lambda gives a good model that predicts the true values fairly well.

Assignment 2 - Decision trees and logistic regression for bank marketing

2.1

Dividing data into training, validation and test sets (40/30/30).

```
library(dplyr)

# Read data
data = read.csv("bank-full.csv", stringsAsFactors=TRUE, sep=";")

# Delete duration column from data
data = data[, !(names(data) %in% c("duration"))]

n = dim(data)[1]
set.seed(12345)
id = sample(1:n, floor(n*0.4))
train = data[id,]

id1 = setdiff(1:n, id)
set.seed(12345)
id2 = sample(id1, floor(n*0.3))
valid = data[id2,]

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

2.2

When talking about nodes it is every single decision you can pick in the tree. Internal nodes are nodes that is not a leaf, i.e, it is a decision to be made somewhere in the middle of the tree, a terminal node is the last node in a decision sequence. When increasing/decreasing the node size the size of the tree increases/decreases because of this, the number of decisions gets changed.

Deviance in this case is equal to entropy from the literature.

```
library(tree)
library(knitr)

# All trees
default = tree(as.factor(y)~., data=data)
node = tree(as.factor(y)~., data=train, control = tree.control(nrow(data), minsize=7000))
deviance = tree(as.factor(y)~., data=train, control = tree.control(nrow(data), mindev=0.0005))

misclassifications = matrix(nrow=3, ncol=2)
dimnames(misclassifications) = list(c("Default", "Node", "Deviance"), c("Training", "Validation"))

# Calculate misclassification errors for default tree
default.pred.train = predict(default, train, type="class")
default.pred.valid = predict(default, valid, type="class")
misclassifications["Default", "Training"] = mean(default.pred.train != train$y)
misclassifications["Default", "Validation"] = mean(default.pred.valid != valid$y)

# Calculate misclassification errors for node tree
```

```
node.pred.train = predict(node, train, type="class")
node.pred.valid = predict(node, valid, type="class")
misclassifications["Node", "Training"] = mean(node.pred.train != train$y)
misclassifications["Node", "Validation"] = mean(node.pred.valid != valid$y)

# Calculate misclassification errors for deviance tree
deviance.pred.train = predict(deviance, train, type="class")
deviance.pred.valid = predict(deviance, valid, type="class")
misclassifications["Deviance", "Training"] = mean(deviance.pred.train != train$y)
misclassifications["Deviance", "Validation"] = mean(deviance.pred.valid != valid$y)

kable(misclassifications, caption="Misclassification errors.")
```

Table 1: Misclassification errors.

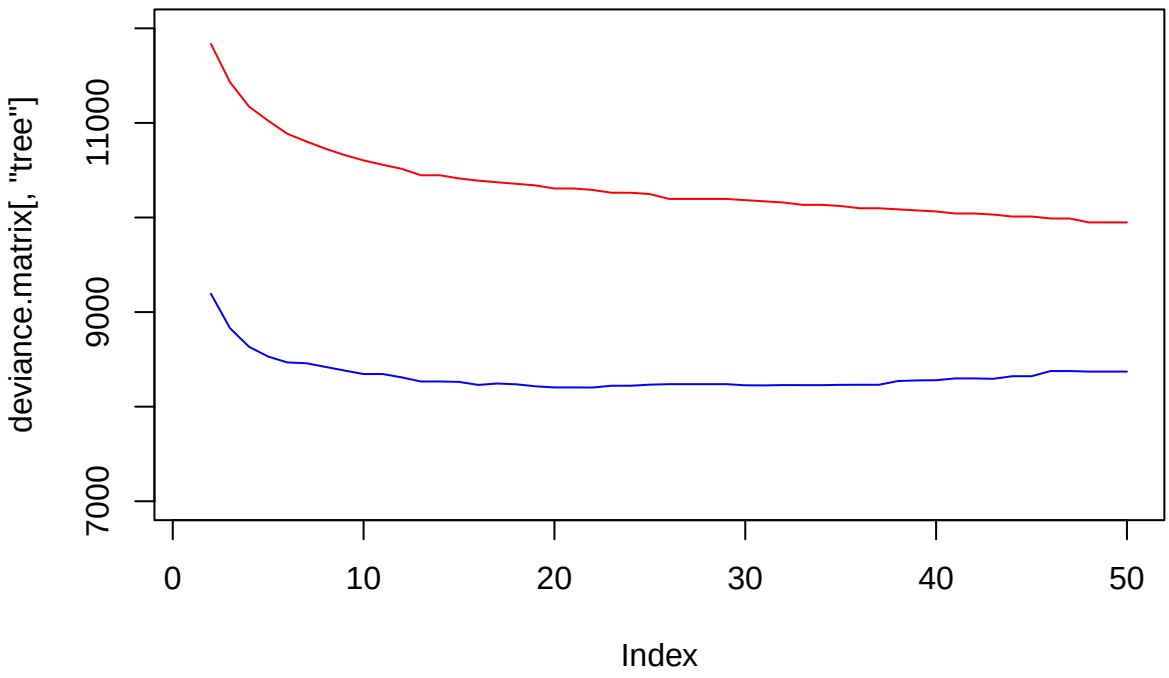
	Training	Validation
Default	0.1048441	0.1092679
Node	0.1048441	0.1092679
Deviance	0.0940058	0.1119221

The tree where we changed the minimum deviance are marginally better than the other two trees for our training data but worse for our validation data which indicates that the model is an overfit.

Increasing the minsize increases the number of data points our node uses to come to a conclusion. Using more data for every decision made the tree smaller but the precision stayed the same.

2.3

```
deviance.matrix = matrix(data = NA, nrow=50, ncol = 2, dimnames = list(c(), c("tree", "fit")))


# number of nodes has to be > 1
for (i in 2:50){
  finalTree = prune.tree(deviance, best=i)
  fit.valid = predict(finalTree, newdata = valid, type = "tree")

  deviance.matrix[i, "tree"] = deviance(finalTree)
  deviance.matrix[i, "fit"] = deviance(fit.valid)
}

plot(deviance.matrix[, "tree"], type="l", col="red", ylim=c(7000, 12000)); lines(deviance.matrix[,"fit"]
```




```
# Lowest deviance
which.min(deviance.matrix[, "fit"])
```

```
## [1] 22
```

```
finalTree = prune.tree(deviance, best=which.min(deviance.matrix[, "fit"]))

best.tree = prune.tree(deviance, best=22)
```

Optimal amount of leaves are 22. Variables used are “poutcome”, “month”, “contact”, “pdays”, age”, “day”, “balance”, “housing” and “job”.

2.4

```
fit = predict(best.tree, newdata=test, type="class")
cm = table(test$y, fit)
accuracy = sum(diag(cm)) / sum(cm)

tp = cm[4]
fp = cm[3]
tpr = tp / (cm[2]+cm[4]) # true positive rates = recall = sensitivity
p = tp / (tp + fp) # precision
f1 = (2 * p * tpr) / (p + tpr) # f1 score

cm
```

```
##      fit
```

```
##          no   yes
##   no  11872   107
##   yes  1371   214
```

```
accuracy
```

```
## [1] 0.8910351
```

```
f1
```

```
## [1] 0.224554
```

An F1-score is better to use when you have imbalanced classes. In this case we do have an imbalanced class since we have almost ten times more "no" in our data, therefore you would rather use an F1-score instead of just calculating the accuracy from the confusion matrix.

When looking at accuracy it's alright with an accuracy of nearly 90%. Our F1-score on the other hand is not so good, the maximum score you can get is 1 and we have 0.22.

2.5

```
tree = tree(as.factor(y)~., data=test)
fit = predict(tree, newdata=test, type="vector")

pred = ifelse(fit[,1]/fit[,2]>5, "no", "yes")

cm = table(test$y, pred)
cm
```

```
##      pred
##          no   yes
##   no  11307   672
##   yes   930   655
```

```
accuracy = sum(diag(cm))/sum(cm)
accuracy
```

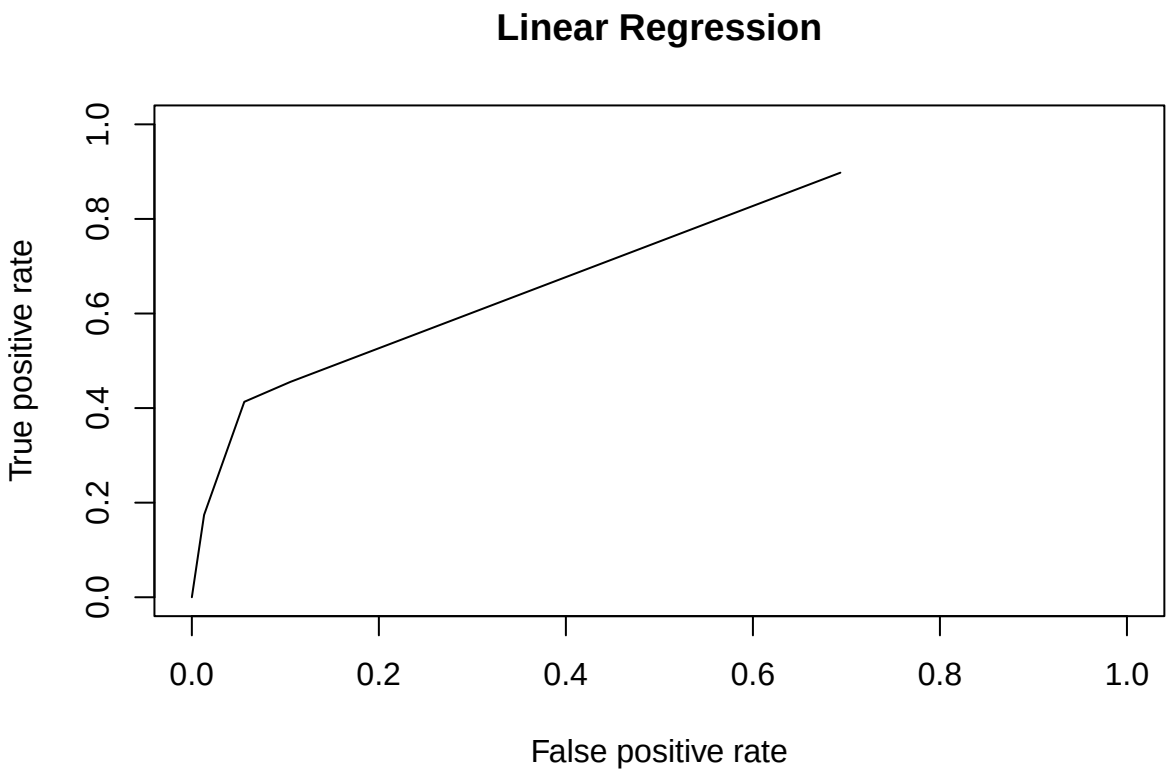
```
## [1] 0.8818932
```

We have nearly the same accuracy when using this method.

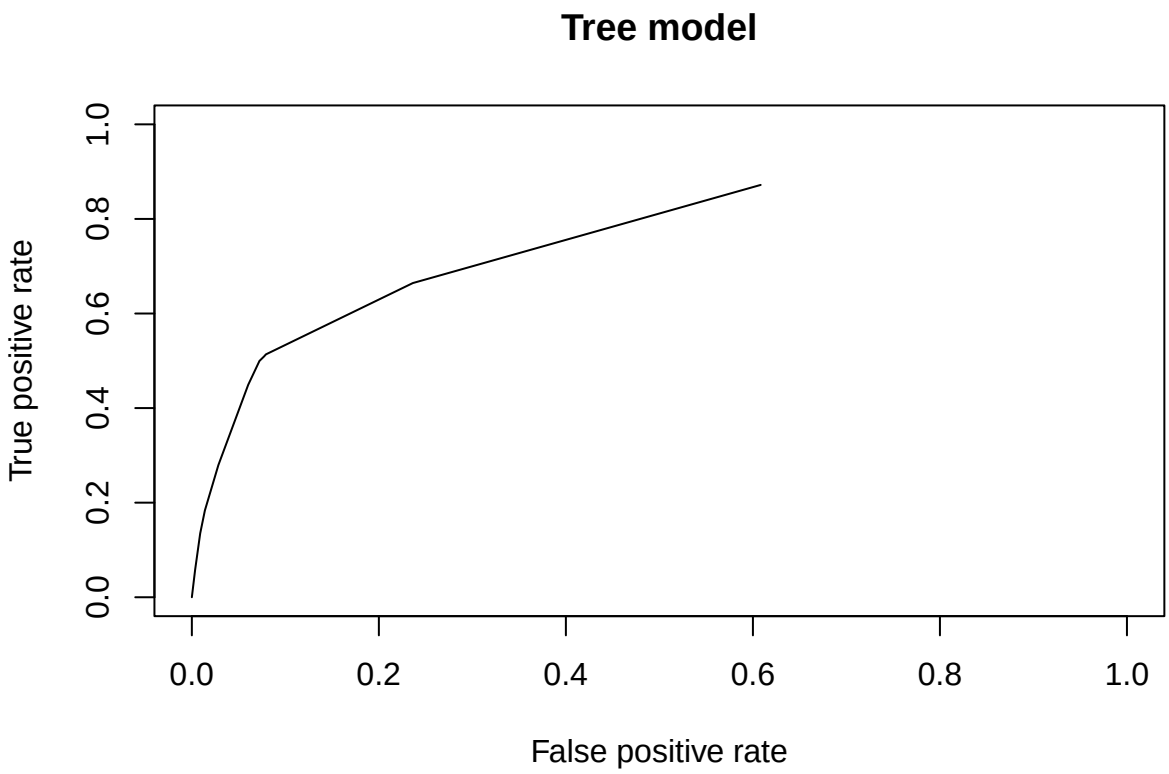
2.6

```
fit.tree = predict(best.tree, newdata=test, type="class")
cm.tree = table(test$y, fit.tree)

tp.tree = cm.tree[4]
fp.tree = cm.tree[3]
tpr.tree = tp.tree / (cm.tree[2]+cm.tree[4]) # true positive rates = recall = sensitivity
fpr.tree = fp.tree / (cm.tree[1]+cm.tree[3]) # false positive rates

interval = seq(from = 0.05, to = 0.95, 0.05)
reg = matrix(data=NA, nrow=length(interval), ncol=2, dimnames = list(c(), c("tpr", "fpr")))
tree.val = matrix(data=NA, nrow=length(interval), ncol=2, dimnames = list(c(), c("tpr", "fpr")))

tree.logreg = tree(as.factor(y)~., data=test)

for (i in 1:length(interval)){
```

```
  # Logistic regression model
  fit = predict(tree.logreg, newdata=test, type="vector")

  pred = ifelse(fit[,2] > interval[i], "yes", "no")
  cm = table(test$y, pred)

  # If we dont have any positives, our cm is not going to be 2x2
  if (is.na(cm[3])){
    tp = 0; fp = 0
  }
  else {
    tp = cm[4]
    fp = cm[3]
  }
  tpr = tp / (cm[2]+tp)
  fpr = fp / (cm[1]+fp)

  reg[i, "tpr"] = tpr
  reg[i, "fpr"] = fpr

  # Best tree model
  fit = predict(best.tree, newdata=test, type="vector")

  pred = ifelse(fit[,2] > interval[i], "yes", "no")
  cm = table(test$y, pred)

  # If we dont have any positives, our cm is not going to be 2x2
  if (is.na(cm[3])){
    tp = 0; fp = 0
  }
  else {
    tp = cm[4]
    fp = cm[3]
  }
  tpr = tp / (cm[2]+tp)
  fpr = fp / (cm[1]+fp)

  tree.val[i, "tpr"] = tpr
  tree.val[i, "fpr"] = fpr
}

reg[is.na(reg)] = 0
tree.val[is.na(tree.val)] = 0

plot(x=reg[,"fpr"], y=reg[,"tpr"], xlab="False positive rate",
     ylab="True positive rate", ylim=c(0, 1), xlim=c(0,1), type="l", main="Linear Regression")
```




```
plot(x=tree.val[,"fpr"], y=tree.val[,"tpr"], xlab="False positive rate",
     ylab="True positive rate", ylim=c(0, 1), xlim=c(0,1), type="l", main="Tree model")
```



We have a slightly better result for using our tree when looking at the AUC. We get the conclusion that our two models aren't working well for all values in our threshold. When we use 0.6~ as the threshold for classifying as a yes we don't get any yes's, neither false or true, therefore we get some values for our FPR and TPR at 0. When using a low threshold we get a lot of yes's which increase our FPR and TPR.

A precision-recall curve might be a better option for evaluating in this case as it is often used when we have class imbalance. A precision-recall curve uses our positives (both true and false) but also our false negatives which would give us some more information about the quality of our model in this case.

Assignment 3 - Principal components and implicit regularization

Task 1

Calculating proportion of variation and study how many components are needed to obtain least 95% of variance in the data.

```
# read data and create data frame
df = read.csv("communities.csv")
# scale all data except df$ViolentCrimesPerPop
scaler = preProcess(df[,-ncol(df)])
df.scaled = predict(scaler,df)
# extract scales eigenvalues and eigenvectors
e.scaled = eigen(cov(df.scaled))
# calculate proportion of variances
pv = e.scaled$values/ncol(df)
# calculate cumulative sum for the proportion of variances
cpv = cumsum(e.scaled$values/ncol(df))
```

```
## [1] "Proportion of variance for PC1: 0.247906179315861"
```

```
## [1] "Proportion of variance for PC2: 0.167722915876905"
```

```
## [1] "Cumulative sum of PC37: 0.948728316304909"
```

```
## [1] "Cumulative sum of PC38: 0.951159857993662"
```

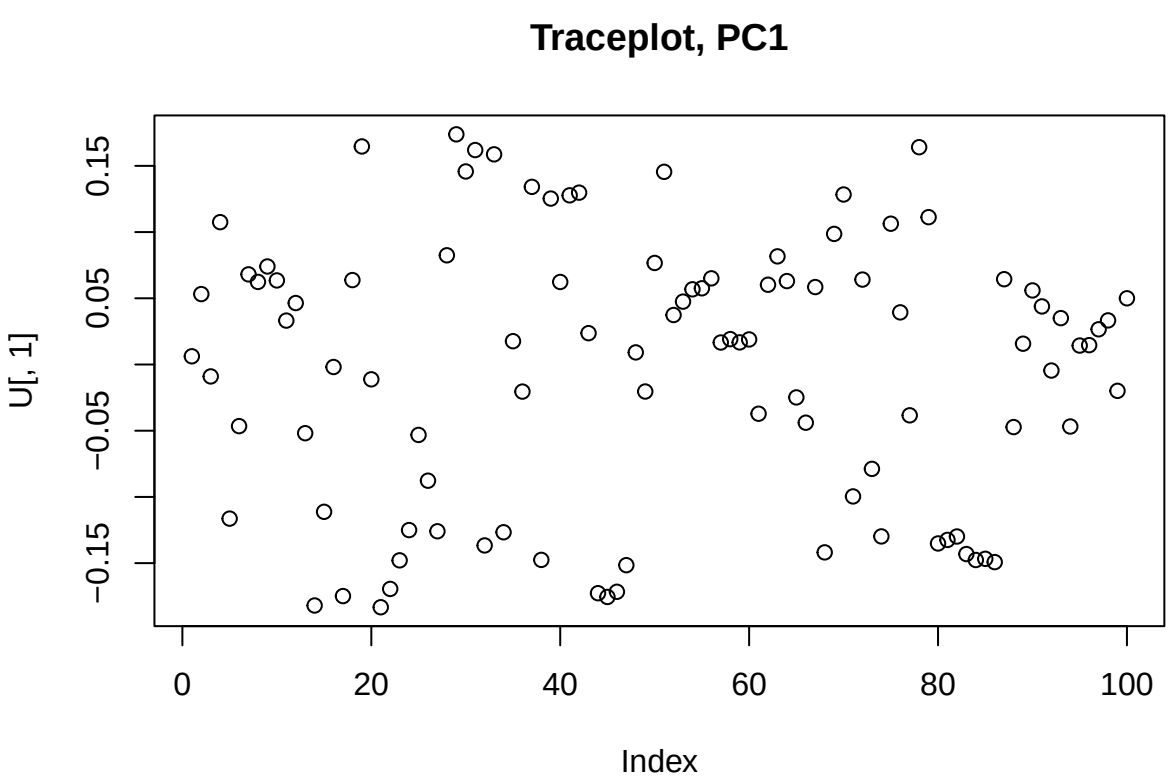
The proportion of variation for the first two components explain the variance for the two components with maximum variation. In this case they are 0.247906179315861 for PC1 and 0.167722915876905 for PC2.

By calculating the cumulative sum of the proportion of variation using function cumsum() one can observe that 38 components are needed to obtain at least 95% of variance in the data.

Task 2

Doing PCA by using princomp() function.

```
# calculate the principal component by using function princomp() and removing lable component
pc = princomp(df.scaled[,-ncol(df)])
# extract the matrix of variable loading's (columns are eigenvectors)
U= pc$loadings
# sorting values in eigenvector corresponding to PC1 in descending order
pc1.features.sorted = sort(abs(U[,1]), decreasing = TRUE)
# extract the coordinates of the individuals (observations) on the principal components.
coord = pc$scores
```



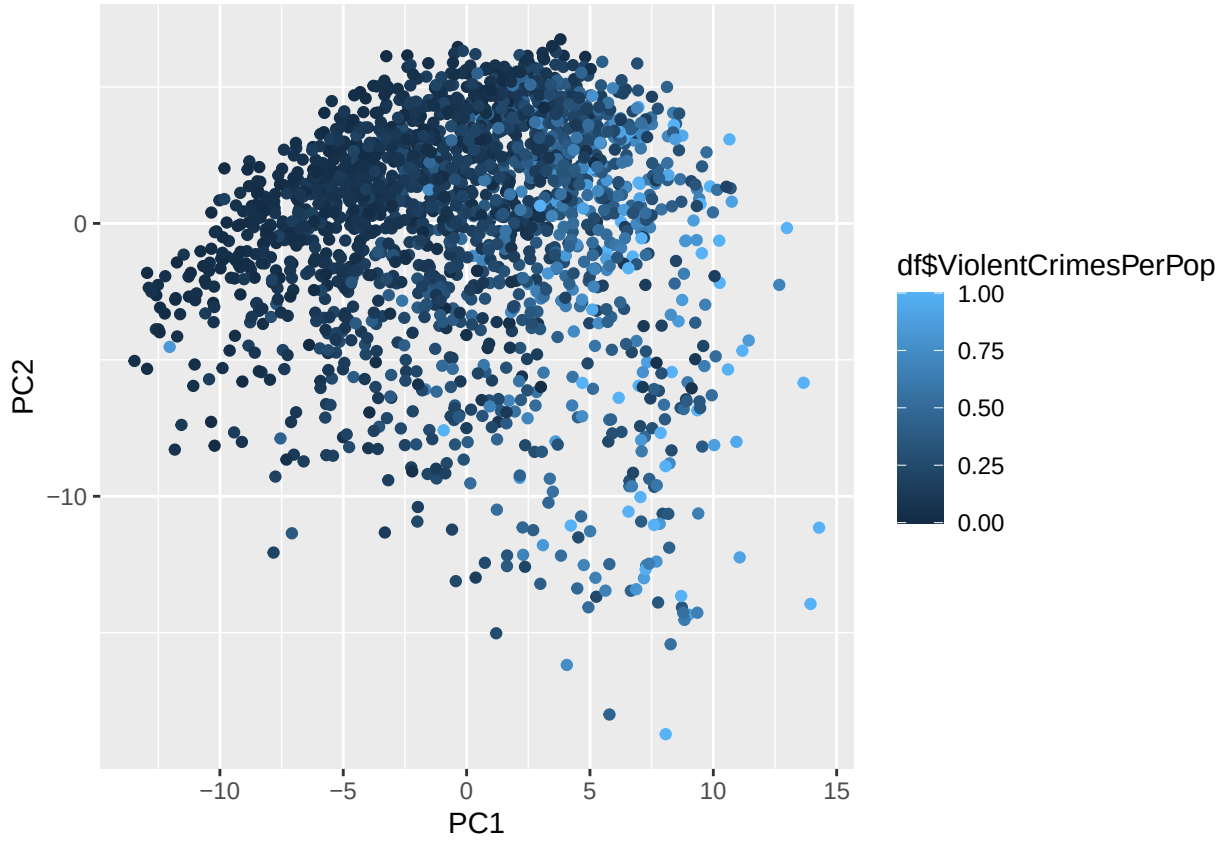
Since the vectors containing the features are multiplied by the convector to all principle component one can determine the effect of the contribution from the features to PC1 by studying the values of PC1's eigenvector. By observing the Traceplot of PC1 one can observe that there are a lot of features contributing to PC1 since there are a lot of points with similar y_value greater than zero.

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

By sorting the eigenvector in descending order and extracting there corresponding feature one can observe that the five features contributing the most to PC1, the first five ones in the sorted vector, are: medFamInc, medIncome, PctKids2Par, pctWInvInc and PctPopUnderPov.

- medFamInc: median family income (differs from household income for non-family households) (numeric - decimal)
- medIncome: median household income (numeric - decimal)
- PctKids2Par: percentage of kids in family housing with two parents (numeric - decimal)
- pctWInvInc: percentage of households with investment / rent income in 1989 (numeric - decimal)
- PctPopUnderPov: percentage of people under the poverty level (numeric - decimal)

Since all of the above have, in some way, a connection to a income or economy it seams reasonable that they would effect the crime level.



When analyzing the score plot above one can observe that many of the variables convey similar information since they are clustered close together. One can also observe that the variables that are clustered together convey less information about the target variable since they are located close to the origin.

There are however some points that are more scattered and further away from the origin indicating that they convey less similar information and have more impact on the target variable.

This is also verified by observing that the color of the clustered points and the more scattered points.

Task 3

Estimating a linear regression model from the training data where ViolentCrimesPerPop is the target and all other data columns are features.

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

# scale data
scaler=preProcess(train)
trainS=predict(scaler,train)
testS=predict(scaler,test)

fit = lm(trainS$ViolentCrimesPerPop~.,data=trainS)
```

```r
# get train mse
y_hat_train = predict(fit, trainS)
train_MSE = mean((trainS$ViolentCrimesPerPop - y_hat_train)^2)

# get test mse
y_hat_test = predict(fit, testS)
test_MSE = mean((testS$ViolentCrimesPerPop - y_hat_test)^2)
```

When computing the training and test MSE for the linear regression model the following is obtained:

- MSE for training data: 0.2752071
- MSE for test data: 0.4248011

Task 4

Implementing a function that depends on parameter vector θ and represents the cost function for a linear regression without intercept. Then optimize θ by optim() function.

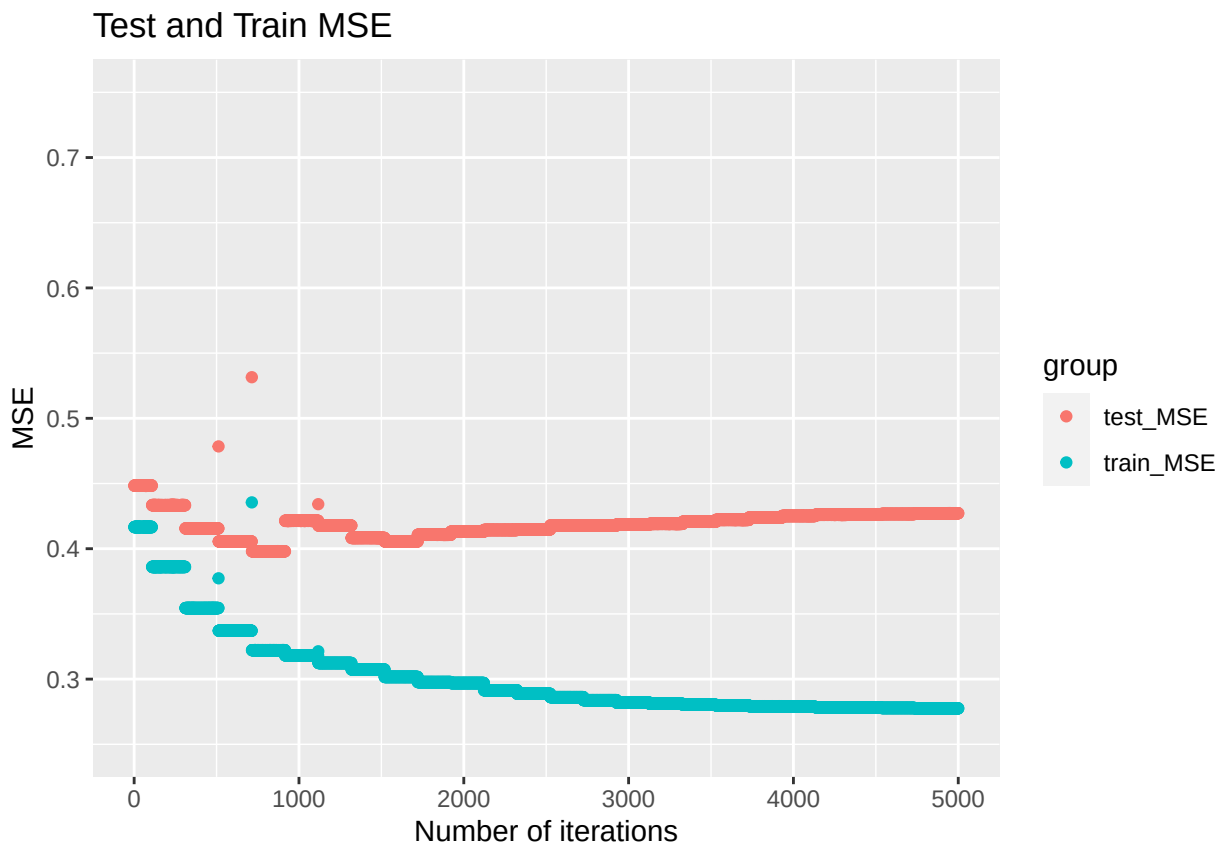
```r
calculateCost = function(theta) {
  # compute matrices for later calculations
  Y = as.matrix(trainS$ViolentCrimesPerPop)
  X_train = as.matrix(trainS[, c(1:ncol(trainS)-1)])
  X_test = as.matrix(testS[, c(1:ncol(trainS)-1)])
  theta = as.matrix(theta)
  dim(theta)
  N = nrow(X_train)
  #Calculate cost
  cost <- sum(((X_train%*%theta)- Y)^2)/(2*N)

  # get train mse
  y_hat_train = X_train %*% theta
  train_MSE = mean((trainS$ViolentCrimesPerPop - y_hat_train)^2)
  # get test mse
  y_hat_test = X_test %*% theta
  test_MSE = mean((testS$ViolentCrimesPerPop - y_hat_test)^2)
  MSE = c(train_MSE, test_MSE)
  .GlobalEnv$df.error[nrow(.GlobalEnv$df.error) + 1,] = MSE

  return(cost)
}
```

Calculate cost by using optim()

```r
z = rep(0, 100)
result = optim(z, fn=calculateCost, method="BFGS")
# discard 500 of the initial iterations
df.error.trim = tail(df.error, -500)
```

```
## [1] "Optimal number of iterations according to the early stopping criterion: 873"
```

```
##     TrainError TestError
## 873  0.3545307 0.4155759
```



When observing the MSE for training and test data for the optimal model according to the early stopping criterion, one can observe:

- MSE for training data: 0.3545307
- MSE for test data: 0.4155759.

From Task 3 the MSE obtain were:

- MSE for training data: 0.2752071
- MSE for test data: 0.4248011.

By comparing these errors it is clear that the liner regression model from Task 3 performed better for the training data then the one used in Task 4. However this does not say much regarding the performance of the model. To compare the performance of the model one can study the MSE for the test data. The MSE from the test data were very similar in Task 3 and Task 4, which means that the performance of the models are very similar. However the method used in task 3 is slightly better.