Computer Lab 2 Block 1 (732A99 Machine Learning)

Lennart Schilling (lensc874)

07 December 2018

Assignment 2: Analysis of credit scoring

2.1

The data from the Excel file *creditscoring.xls* will be imported and splitted into train (50%), validation (25%) and test data (25%).

```
# importing data
data = read_excel("creditscoring.xls")

# changing class of dependent variable good_bad
data$good_bad = as.factor(data$good_bad)

# dividing data into train (50%), validation (25%) and test set (25%)
n = dim(data)[1]
set.seed(12345)
id = sample(1:n, floor(n*0.5))
train = data[id,]
id1 = setdiff(1:n, id)
set.seed(12345)
id2 = sample(id1, floor(n*0.25))
valid = data[id2,]
id3 = setdiff(id1,id2)
test = data[id3,]
```

2.2

```
fitDecisionTree = function(testData, splitMethod) {
  # fitting decision tree
  decisionTreeFit = tree(formula = good_bad ~ ., data = train, split = splitMethod)
  # getting prediction results
  yFit = predict(object = decisionTreeFit, newdata = testData, type = "class")
  # printing misclassification rate
  paste0("Misclassification rate: ", mean(yFit != as.character(testData$good_bad)))
}
fitDecisionTree(testData = train, splitMethod = "deviance")
```

```
## [1] "Misclassification rate: 0.212"
```

```
fitDecisionTree(testData = test, splitMethod = "deviance")
```

```
## [1] "Misclassification rate: 0.268"
```

```
fitDecisionTree(testData = train, splitMethod = "gini")
```

```
## [1] "Misclassification rate: 0.238"
```

```
fitDecisionTree(testData = test, splitMethod = "gini")
```

```
## [1] "Misclassification rate: 0.372"
```

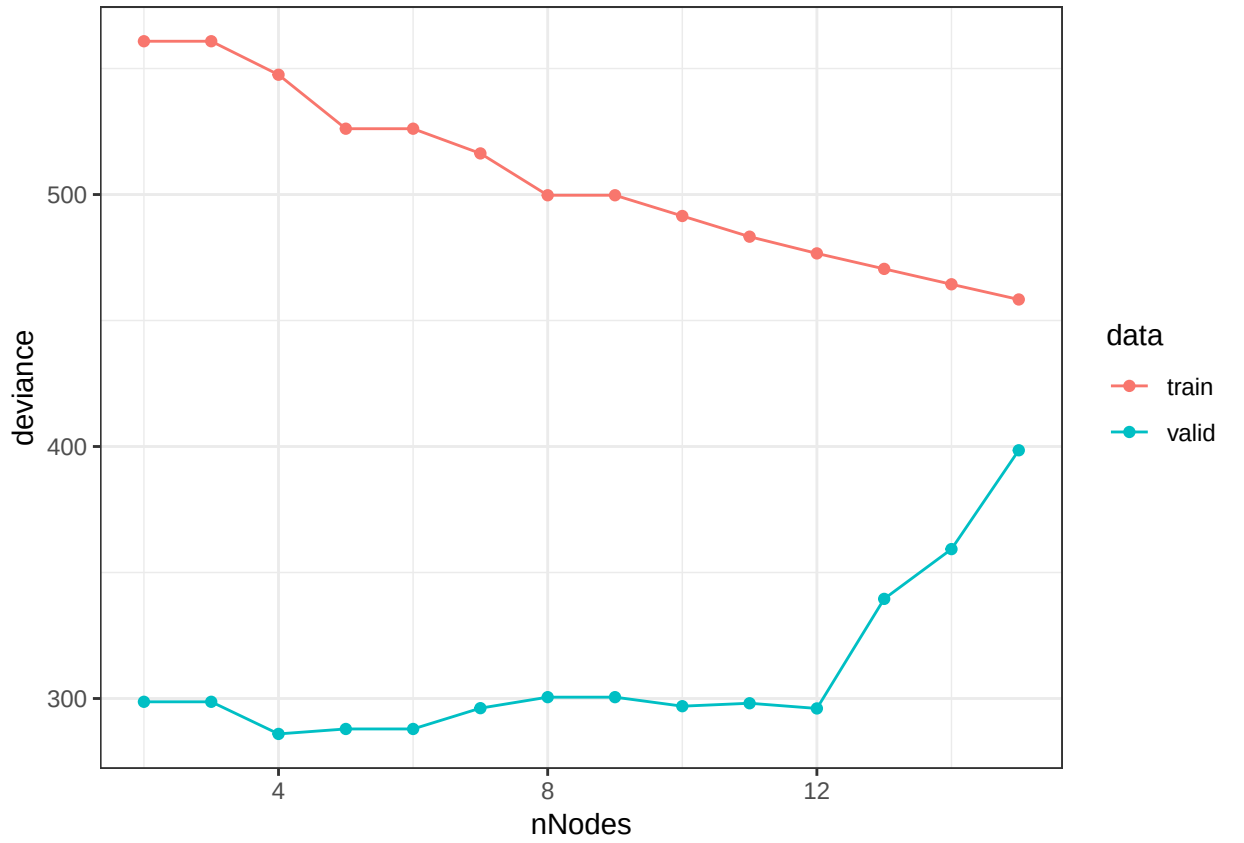
For both split methods it can be seen that the misclassificatin rate for the training data is lower than then the misclassificatin rate for the test data. Using the deviance as a split method results in a much lower misclassification rate compared to the gini split method (26.8 % vs. 37.2%). In this case, using gini leads to a much stronger overfitting. Since deviance provides better results, it will be chosen for the following steps.

The final decision tree using deviance as a splitting method will be saved in the *unprunedTree*-object.

```
# fitting tree with measure providing better results ("deviance")
unprunedTree = tree(formula = good_bad ~ ., data = train, split = "deviance")
```

2.3

```
# identifying number of terminal nodes for unpruned tree
maxNodes = summary(unprunedTree)$size
# calculating deviances for each possible number of leaves
trainScore = rep(0,maxNodes)
validationScore = rep(0,maxNodes)
for(i in 2:maxNodes) {
  prunedTree = prune.tree(unprunedTree, best = i)
  pred = predict(prunedTree, newdata = valid, type = "tree")
  trainScore[i] = deviance(prunedTree)
  validationScore[i] = deviance(pred)
}
# plotting deviances vs. number of leaves
deviances = as.data.frame(cbind(nNodes = c(1:maxNodes),
                                deviance = c(trainScore, validationScore),
                                data = c(rep("train", 15), rep("valid", 15))))
deviances$nNodes = as.numeric(as.character(deviances$nNodes))
deviances$deviance = as.numeric(as.character(deviances$deviance))
ggplot(data = deviances[deviances$nNodes != 1,], aes(x = nNodes, y = deviance, color = data)) +
  geom_point() +
  geom_line() +
  theme_bw()
```



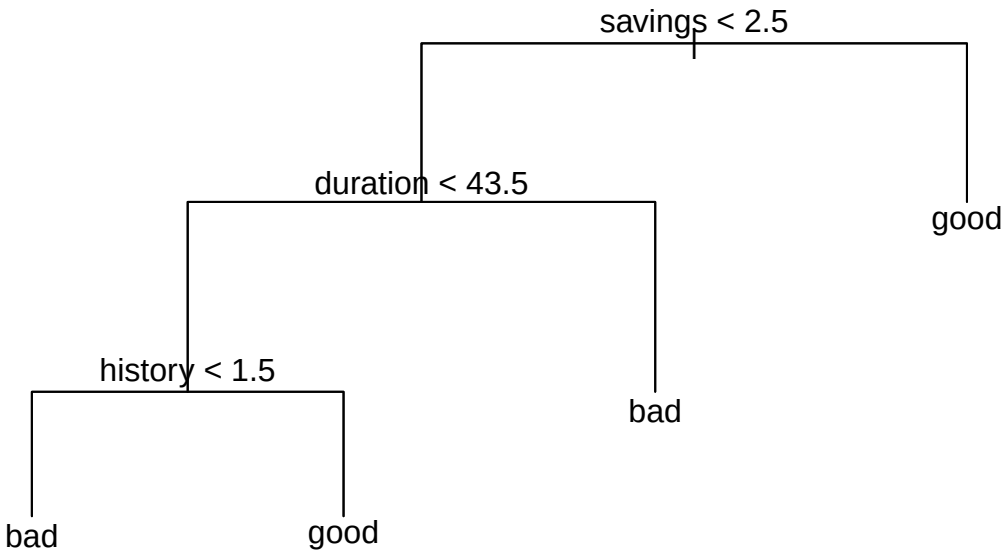

```
# Prooving number of leaves with minimum deviancce
knitr::kable(as.data.frame(deviances[deviances$deviance == min(deviances[deviances$nNodes != 1 & devianc
```

	nNodes	deviance
19	4	285.9425

Observing the deviances for the different number of leaves, in the plot the typical graphs for train and validation set can be found. While for an increasing complexity, the deviance decreases for the train set, first also decreases for the validation set but increases after a certain time. The minimum deviance for the validation set is both graphically and in the computed way shown for the number of four leaves.

The optimal tree with four leaves is plotted:

```
# reporting the optimal tree
optimalTree = prune.tree(unprunedTree, best = 4)
plot(optimalTree)
text(optimalTree, pretty = 0)
```



The depth of this tree equals to 3. The variables used are *savings*, *duration* and *history*. Having savings $>$ 2.5 for example would lead to a classification as *good* which indicates that the customer is likely to pay back the loan. In the other hand, only a duration smaller than 43.5 and a history of more than 1.5 would lead to another *good*-classification.

For the test data, the misclassification rate can be estimated as follows:

```
# estimating misclassification rate for test data
yFit = predict(object = optimalTree, newdata = test, type = "class")
paste0("Misclassification rate: ", mean(yFit != as.character(test$good_bad)))
```

```
## [1] "Misclassification rate: 0.256"
```

2.4

In the next step, we perform classification using Naïve Bayes. The confusion matrices and misclassification rates for the training and for the test data are reported.

```
fitNaiveBayes = function(testData) {
  naiveBayesFit <<- naiveBayes(formula = good_bad ~ ., data = train)
  yFit = predict(naiveBayesFit, newdata = testData, type = "class")
  # printing confusion matrix
  print(table(y = testData$good_bad, yFit))
  # printing misclassificaton rate
  paste0("Misclassification rate: ", mean(yFit != as.character(testData$good_bad)))
}

fitNaiveBayes(testData = train)
```

```
##       yFit
## y      bad good
##   bad   95   52
##   good  98  255
```

```
## [1] "Misclassification rate: 0.3"
```

```
fitNaiveBayes(testData = test)
```

```
##       yFit
## y      bad good
##   bad   46   30
##   good  49  125
```

```
## [1] "Misclassification rate: 0.316"
```

Compared to the optimal tree from *2.3*, the Naive Bayes classifier does not deliver the same accuracy. For both data (train and test data), the misclassification rate exceeds the rate from the decision tree in *2.3*.

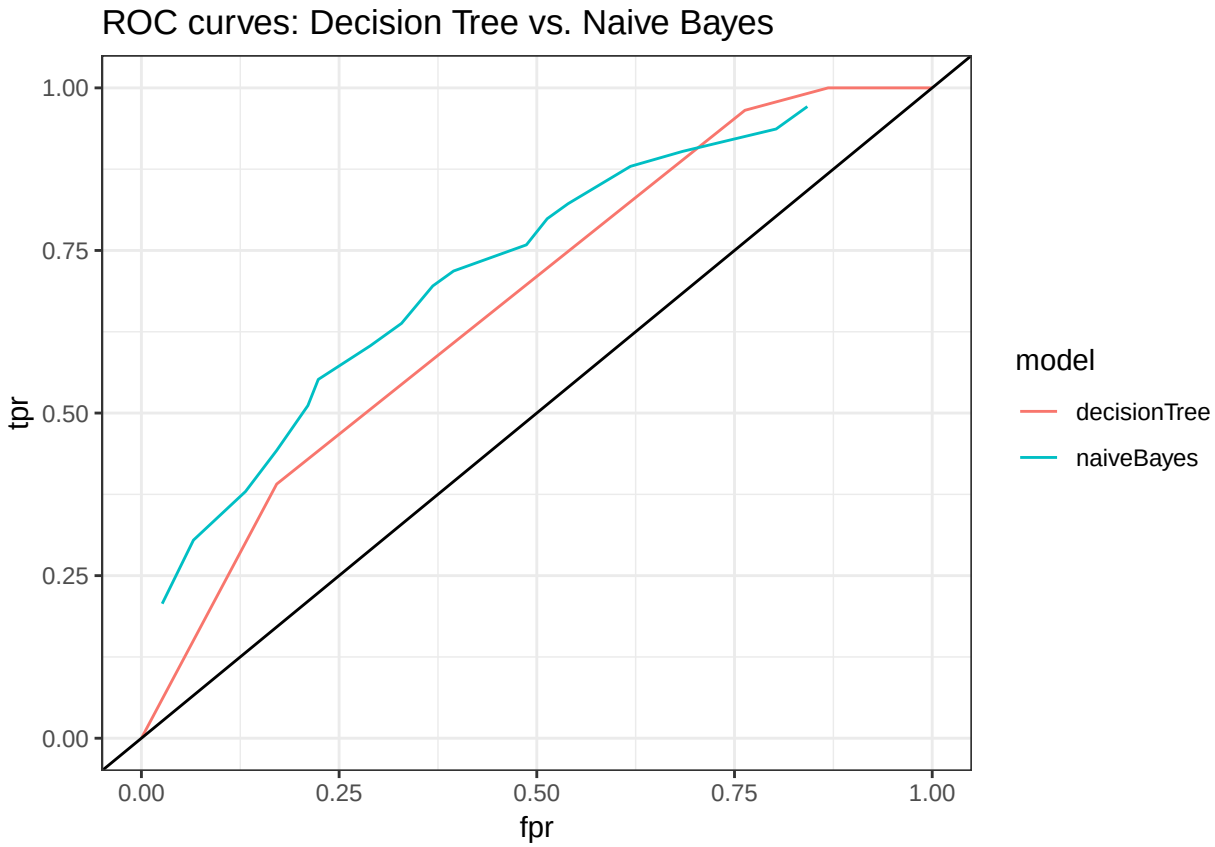
2.5

```
# creating data for ROC curve
getRocData = function(rocData, model) {
  for (pi in seq(from = 0.05, to = 0.95, by = 0.05)) {
    # getting prediction results
    if (model == "decisionTree") {
      yFit = predict(object = optimalTree, newdata = test)
    } else if (model == "naiveBayes") {
      yFit = predict(object = naiveBayesFit,
                     newdata = test,
                     type = "raw")
    }
    # adjusting prediction results
    adjustedYFit = c()
    for (pred in 1:nrow(yFit)) {
      if (yFit[pred, 2] > pi) {
        adjustedYFit = c(adjustedYFit, "good")
      } else {
        adjustedYFit = c(adjustedYFit, "bad")
      }
    }
    # merging actual and predicted values to one data frame
    yVSyFit = as.data.frame(cbind(y = as.character(test$good_bad), yFit = adjustedYFit))
    # calculating true positive rate (TPR)
    tpr = nrow(yVSyFit[yVSyFit$y == "good" &
                         yVSyFit$yFit == "good", ]) / nrow(yVSyFit[yVSyFit$y == "good", ])
    # calculating false positive rate (FPR)
    fpr = nrow(yVSyFit[yVSyFit$y == "bad" &
                         yVSyFit$yFit == "good", ]) / nrow(yVSyFit[yVSyFit$y == "bad", ])
    # adding information to rocData
    rocData = rbind(rocData, cbind(tpr, fpr, model = model, pi))
  }
  # returning rocData
  return(rocData)
}
rocData = getRocData(rocData = data.frame(), model = "decisionTree")
```

```r
rocData = getRocData(rocData = rocData, model = "naiveBayes")

# adjusting classes of rocData-variables
rocData$tpr = as.numeric(as.character(rocData$tpr))
rocData$fpr = as.numeric(as.character(rocData$fpr))

# plotting ROC curve using rocData
ggplot(data = rocData, aes(x = fpr, y = tpr, color = model)) +
  geom_line() +
  geom_abline(intercept = 0, slope = 1) +
  theme_bw() +
  ggtitle("ROC curves: Decision Tree vs. Naive Bayes")
```



Analysing the true positive rate (tpr) and false positive rate (tpr), for most of the classification settings, the Naive Bayes classifier leads to a higher true positive rate compared to the decision tree having the same false positive rate. As closer the curve is oriented towards the top left corner (tpr = 1, fpr = 0), as better the model. That is why related to these statistics, the Naive Bayes classifier performs better.

2.6

In the following, the classification setting was adjusted. Since a classification as *good* and an observation as *bad* would lead to a loss of 10, the classification as *good* should be only done if the probability is at least 10 times as high as the probability for *bad*. The exact implementation can be found after the comment *adjusting prediction results*.

```r
fitNaiveBayesAdjustedLoss = function(testData) {
  naiveBayesFit = naiveBayes(formula = good_bad ~ ., data = train)
```

```
  yFit = predict(naiveBayesFit, newdata = testData, type = "raw")
  # adjusting prediction results
  adjustedYFit = c()
  for (pred in 1:nrow(yFit)) {
    if (yFit[pred, 1] * 10 > yFit[pred, 2]) {
      adjustedYFit = c(adjustedYFit, "bad")
    } else {
      adjustedYFit = c(adjustedYFit, "good")
    }
  }
  # printing confusion matrix
  return(table(y = testData$good_bad, yFit = adjustedYFit))
}
fitNaiveBayesAdjustedLoss(testData = train)
```

```
##       yFit
## y      bad good
##   bad  137   10
##   good 263   90
```

```
fitNaiveBayesAdjustedLoss(testData = test)
```

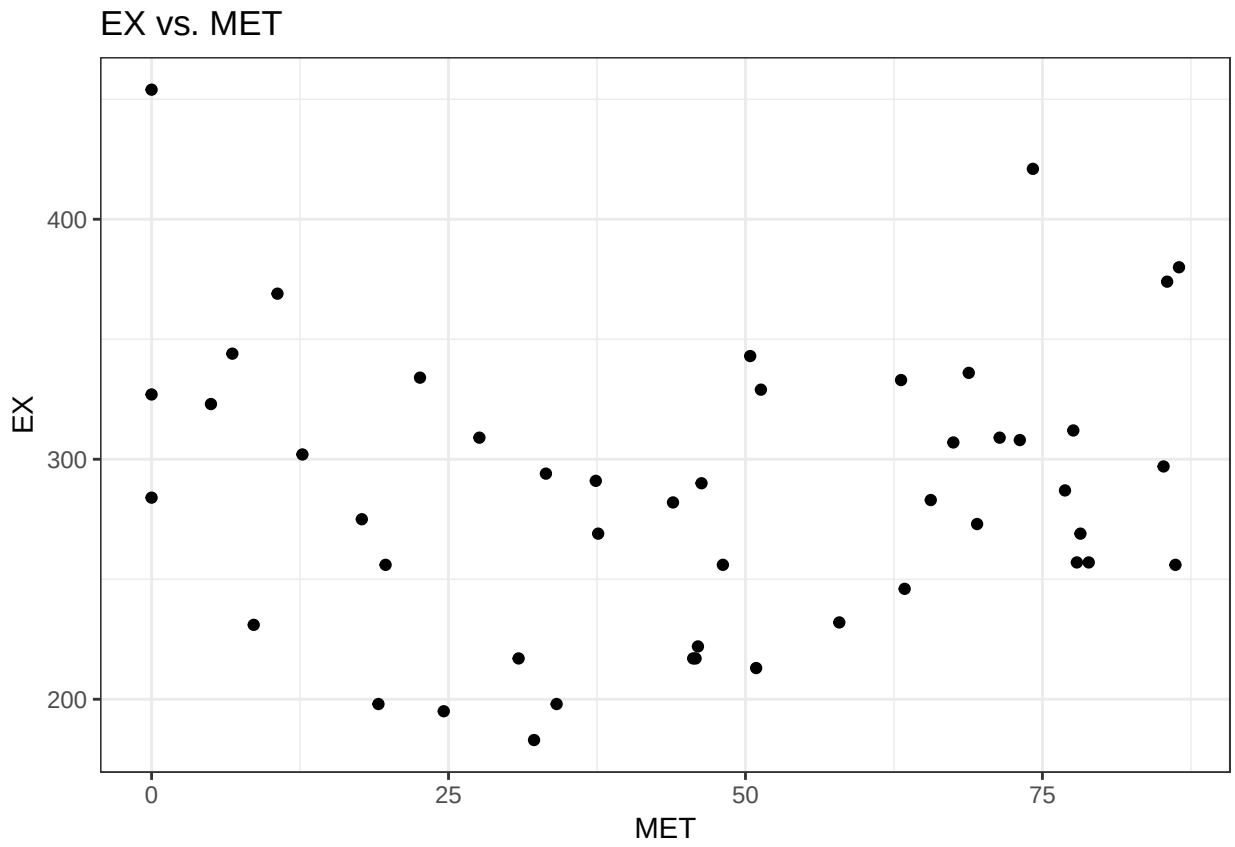
```
##       yFit
## y      bad good
##   bad   71    5
##   good 122   52
```

As a result of the lower chance to be classified as *good*, there are of course less *good*-classifications This automatically leads to the concusion that the number of cases where *bad* is predicted and *good* observed increased dramatrically.

Assignment 3: Uncertainty estimation

3.1

```
# importing data
data =  read.csv2("State.csv", sep = ";")
# reordering data
data = data[order(data$MET),]
# plotting EX vs. MET
ggplot(data = data, mapping = aes(x = MET, y = EX)) +
  geom_point() +
  theme_bw() +
  ggtitle("EX vs. MET")
```

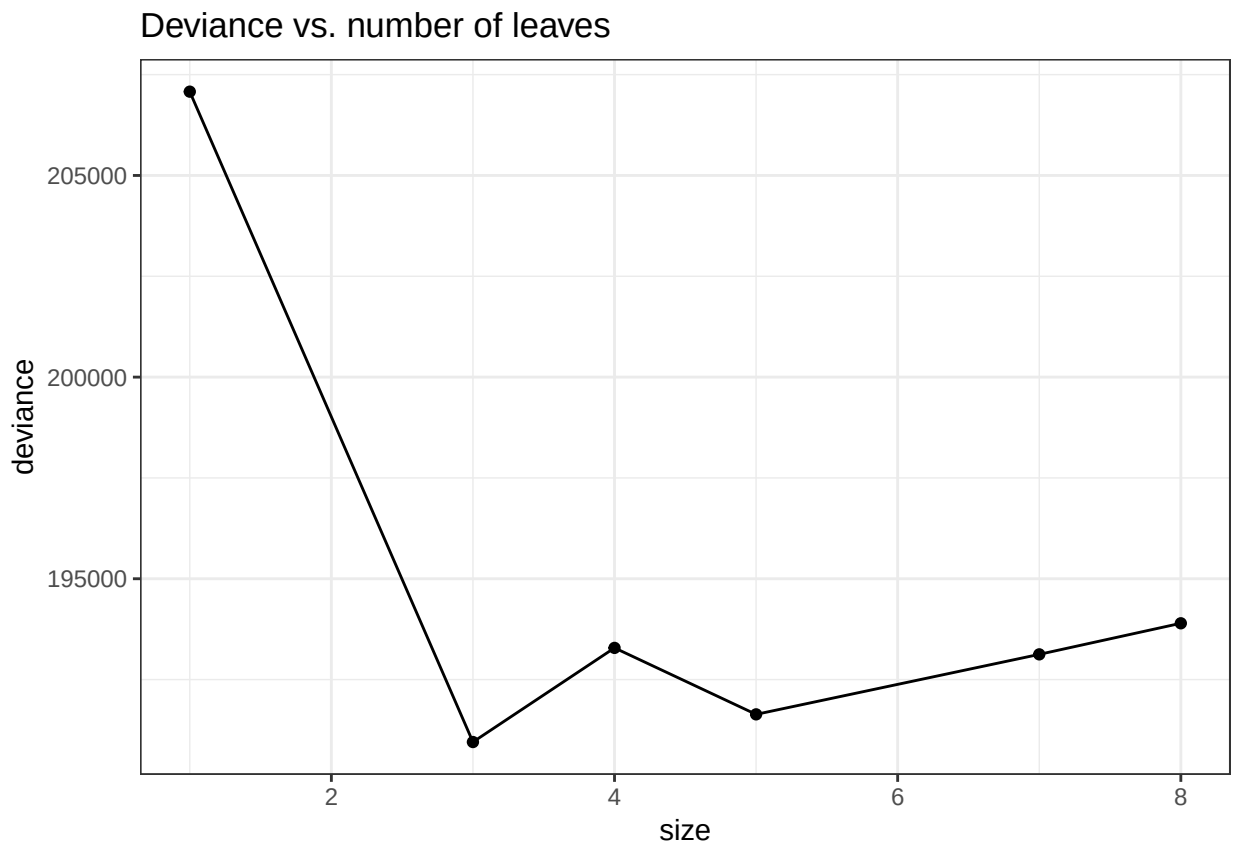


Since there does not seem to be a liner relation between the two variables, a regression tree could be an appropriate model, since it is a non-linear classifier.

3.2

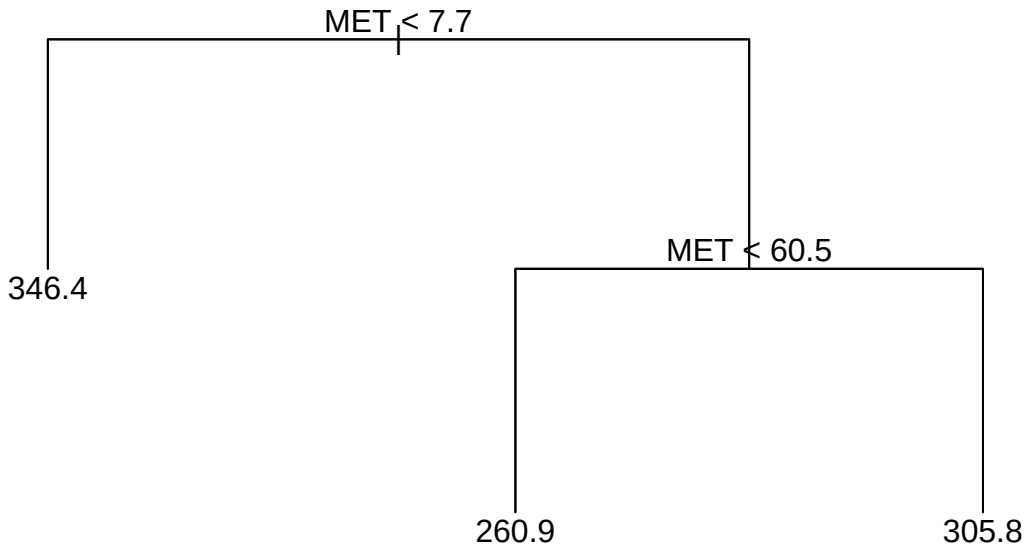
```
# fitting decision tree
unprunedTree = tree(formula = EX ~ MET,
                    data = data,
                    control = tree.control(nobs = nrow(data), minsize = 8))

# identifying optimal number of leaves
set.seed(12345)
cv.res = cv.tree(unprunedTree)
ggplot(data = as.data.frame(cbind(size = cv.res$size, deviance = cv.res$dev)), mapping = aes(x = size, y
  geom_point() +
  geom_line() +
  theme_bw() +
  ggtitle("Deviance vs. number of leaves")
```



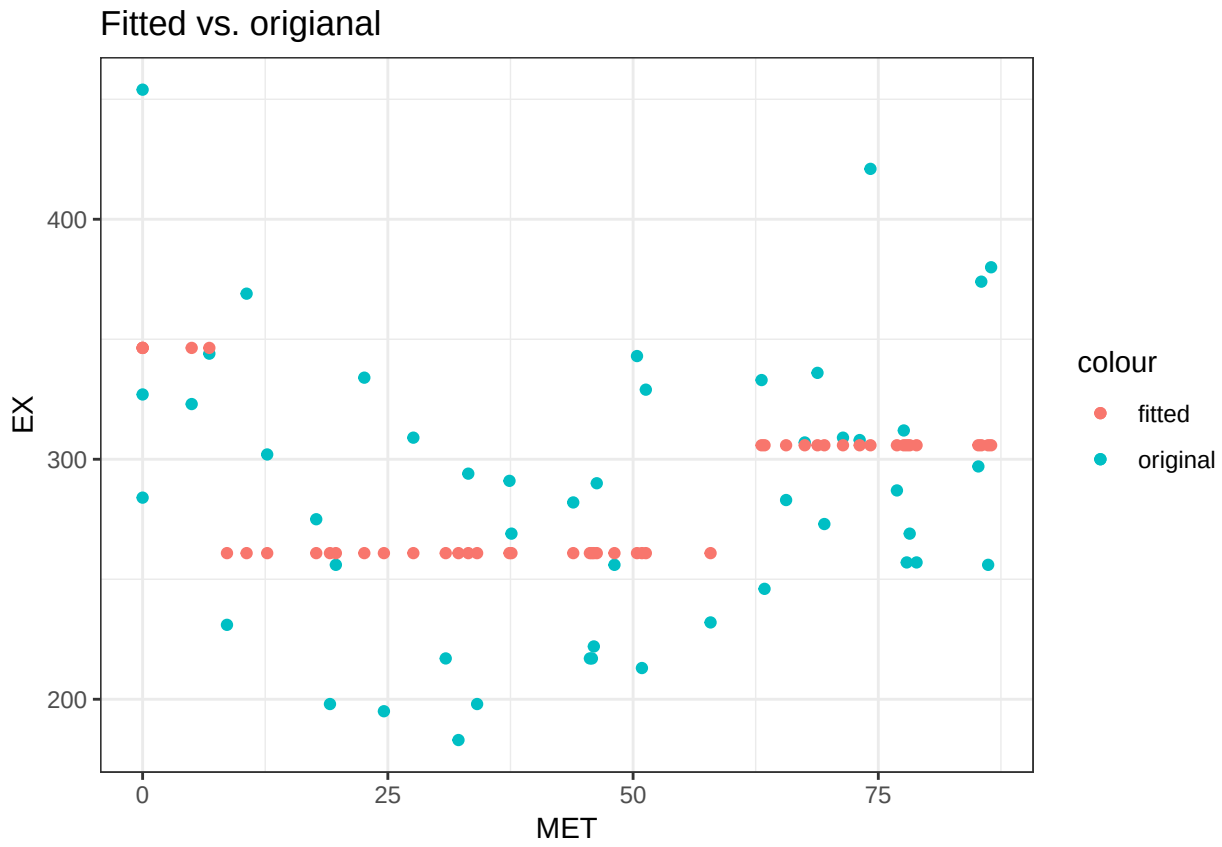
The plot shows that the optimal number of leaves seems to be 3. Using this as a setting for post-pruning, the following pruned optimal tree will be created:

```
# reporting selected tree
optimalTree = prune.tree(unprunedTree, best = 3)
plot(optimalTree)
text(optimalTree, pretty = 0)
```



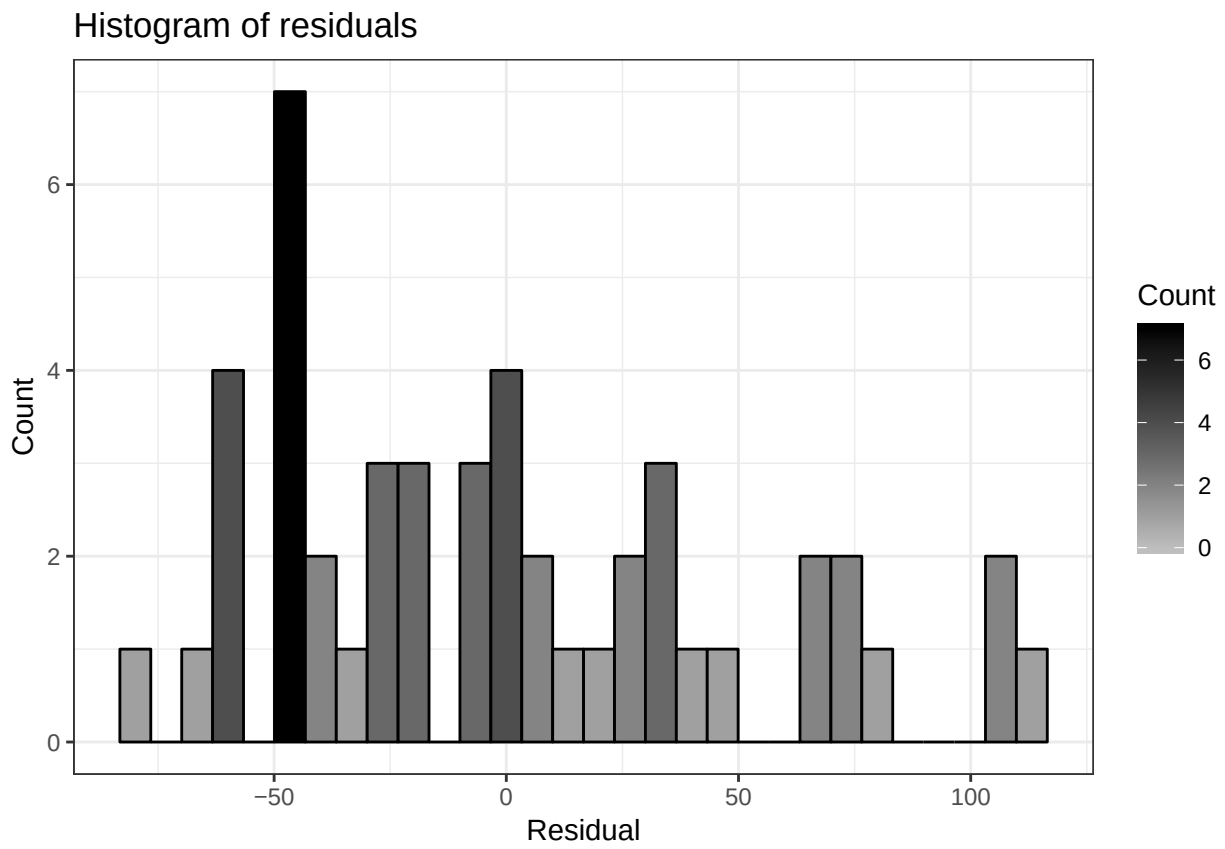
In the next step, the original and fitted data will be compared in a plot:

```
# plotting of original data, fitted data, residuals
yFit = predict(object = optimalTree, newdata = data)
  # Fitted vs. Original
ggplot(data = as.data.frame(cbind(MET = data$MET, EX = data$EX, yFit = yFit))) +
  geom_point(mapping = aes(x = MET, y = EX, colour = "original")) +
  geom_point(mapping = aes(x = MET, y = yFit, color = "fitted")) +
  theme_bw() +
  ggtitle("Fitted vs. origianal")
```



The difference between the original and fitted data leads to the following histogram of the residuals:

```
  # Residuals
ggplot(data = as.data.frame(data$EX - yFit),
       aes(data$EX - yFit)) +
  geom_histogram(col="black",
                 aes(fill=..count..)) +
  scale_fill_gradient("Count", low = "grey", high = "black") +
  theme_bw() +
  ggtitle("Histogram of residuals") +
  labs(x="Residual", y="Count")
```



3.3

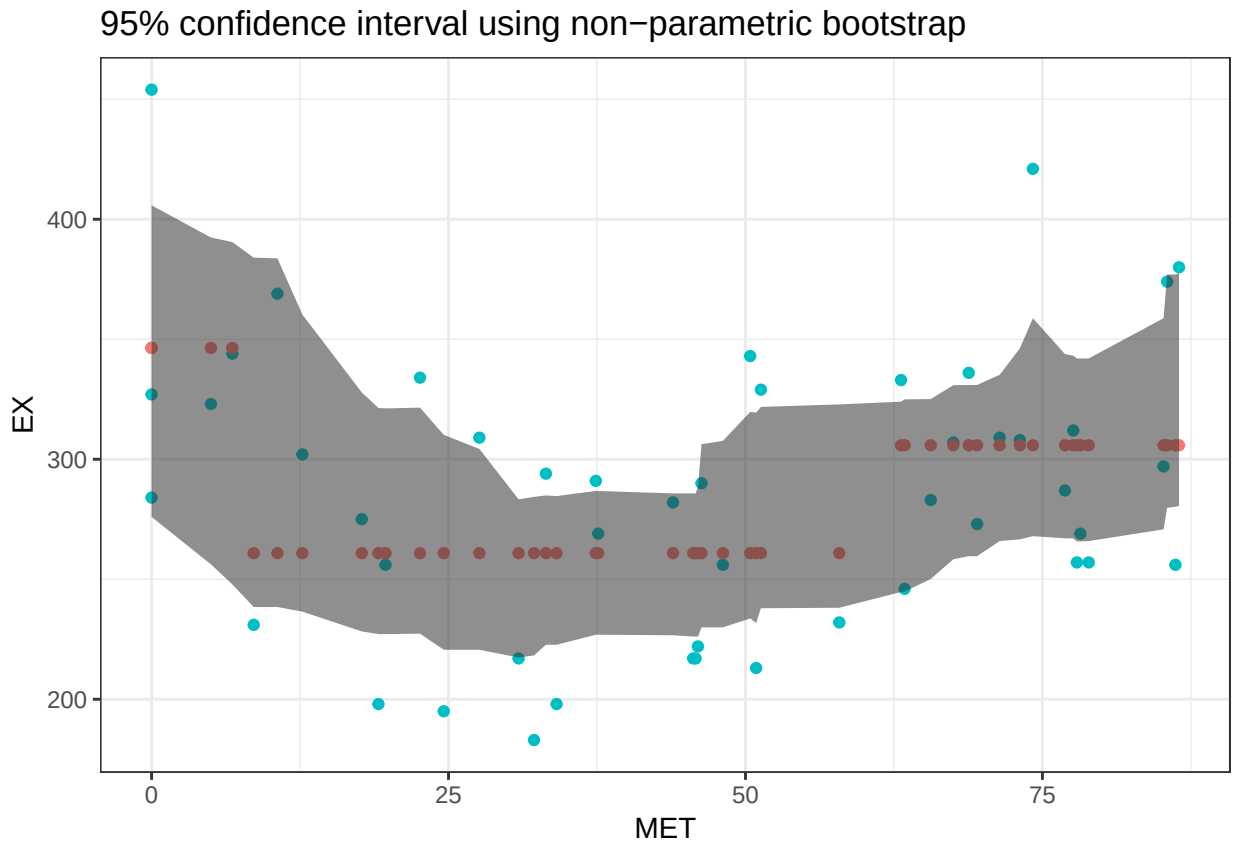
The 95% confidence bands for the regression tree model from step 2 will be computed and plotted by using a non-parametric bootstrap.

```
# defining bootstrap function
nonParametricBootstrap = function(data, ind){
  # extracting bootstrap sample
  bootstrapSample = data[ind,]
  # fitting decision tree
  unprunedTree = tree(formula = EX ~ MET,
                      data = bootstrapSample,
                      control = tree.control(nobs = nrow(bootstrapSample), minsize = 8))
  # pruning decision tree
  prunedTree = suppressWarnings(prune.tree(unprunedTree, best = 3))
  # getting prediction results for original data
  yFit = predict(prunedTree, newdata = data)
  return(yFit)
}
# running bootstrap analysis
nonParametricBootstrapResult = boot(data = data, statistic = nonParametricBootstrap, R = 1000)

# calculating confidence envelopes
nonParametricConfidenceEnvelopes = envelope(nonParametricBootstrapResult)

# plotting original data, predictions from optimalTree and confidence bands
```

```r
plottingConfidenceIntervals = function(confidenceEnvelopes, bootstrap) {
  plotData = as.data.frame(cbind(MET = data$MET,
                                 EX = data$EX,
                                 lower = as.vector(confidenceEnvelopes$point[2,]),
                                 upper = as.vector(confidenceEnvelopes$point[1,])))
  yFit = predict(object = optimalTree, newdata = data)
  ggplot() +
    geom_point(data = data,
               aes(x = data$MET,
                   y = data$EX,
                   color = "original")) +
    geom_point(data = as.data.frame(cbind(MET = data$MET, EX = data$EX, yFit = yFit)),
               aes(x = MET,
                   y = yFit,
                   color = "fitted from step 2")) +
    geom_ribbon(data = plotData,
                aes(x = MET,
                    ymin = lower,
                    ymax = upper,
                    alpha = 1)) +
    theme_bw() +
    labs(title = paste0("95% confidence interval using ", bootstrap, " bootstrap"),
         x = "MET",
         y = "EX") +
    theme(legend.position="none")
}
plottingConfidenceIntervals(nonParametricConfidenceEnvelopes, bootstrap = "non-parametric")
```



While the lower band seems to be quite smooth, the upper band indicates a rather bumpy shape. The results of the regression model in step 2 is integrated into the plot (red dots). Since all predictions lie within the confidence interval, it seems to be reliable.

3.4

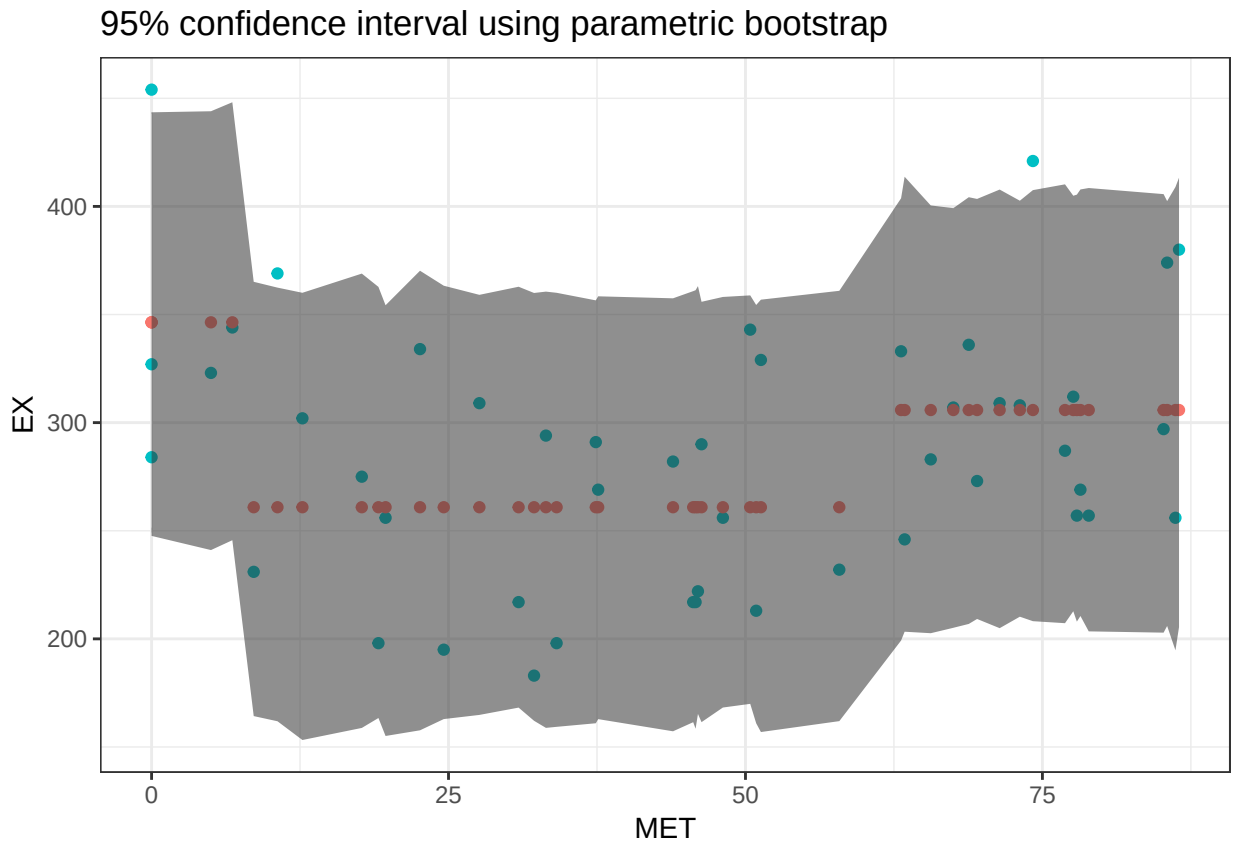
```
parametricBootstrap = function(data) {
  # fitting decision tree
  unprunedTree = tree(formula = EX ~ MET,
                      data = data,
                      control = tree.control(nobs = nrow(data), minsize = 8))
  # pruning decision tree
  prunedTree = prune.tree(unprunedTree, best = 3)
  # getting prediction results
  yFit = predict(prunedTree, newdata = data)
  predictedP = rnorm(n = length(data$EX), mean = yFit, sd = sd(data$EX - yFit))
  return(predictedP)
}

# rng=function(data, mle) {
#   data1=data.frame(Price=data$Price, Area=data$Area)
#   n=length(data$Price)
#   #generate new Price
#   data1$Price=rnorm(n,predict(mle, newdata=data1),sd(mle$residuals))
#   return(data1)
# }
```

```
# running bootstrap analysis
parametricBootstrapResult = boot(data, statistic = parametricBootstrap,
                                 R=1000, mle = prunedTree, sim="parametric")

# calculating confidence envelopes
parametricConfidenceEnvelopes = suppressWarnings(envelope(parametricBootstrapResult))

# plotting original data, predictions from optimalTree and confidence bands
plottingConfidenceIntervals(parametricConfidenceEnvelopes, bootstrap = "parametric")
```

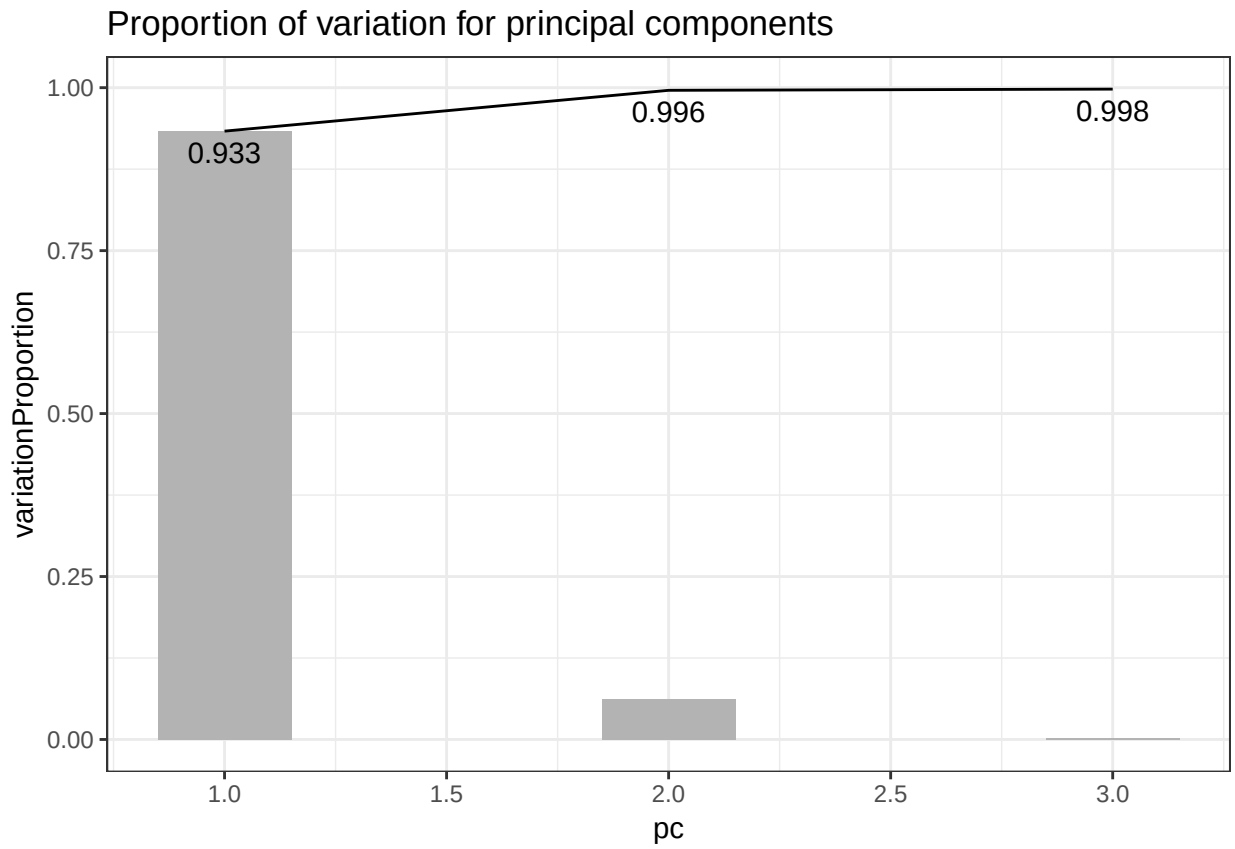


Assignment 4: Principal components

4.1

```
# importing data
data = read.csv2("NIRSpectra.csv", sep = ";")
# preparing data for pca (removing target variable)
pcaData = data[,-127]
# performing pca
pcaResult = prcomp(pcaData)
# calculating eigenvalue for each principal component
eigenvalues = pcaResult$sdev^2
# plotting proportion of variation for principal components
plotData = as.data.frame(cbind(pc = 1:3,
```

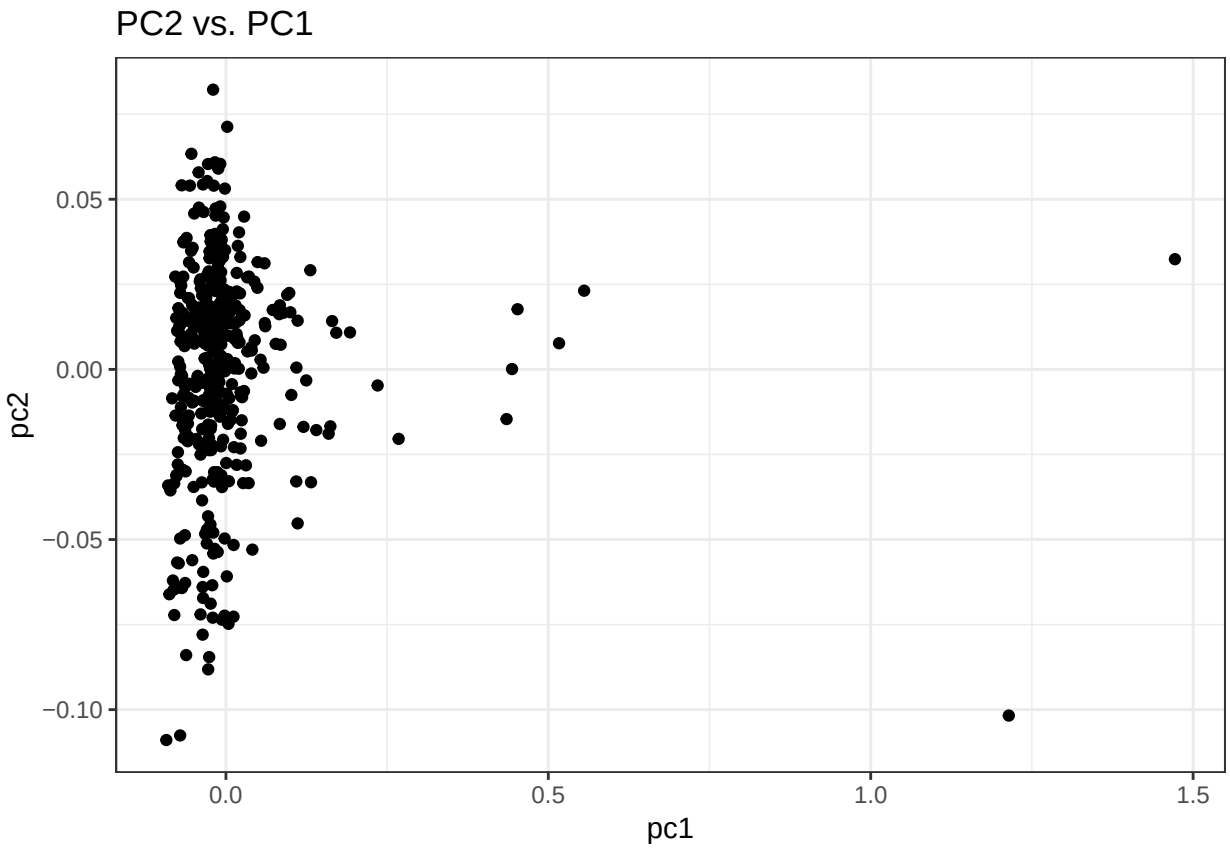
```
                                variationProportion = eigenvalues[1:3]/sum(eigenvalues),
                                cummulative = cumsum(eigenvalues[1:3]/sum(eigenvalues))))
ggplot(data = plotData) +
  geom_col(aes(x = pc, y = variationProportion), width = 0.3, fill = "grey70") +
  geom_line(data = plotData,
            aes(x = pc, y = cummulative)) +
  geom_text(aes(x = pc, y = cummulative, label = round(cummulative, 3)), size = 4,
            position = "identity", vjust = 1.5) +
  theme_bw() +
  ggtitle("Proportion of variation for principal components")
```



The plot shows that only PC1 explains 93.3% variation. To explain at least 99% of the variation, PC1 and PC2 (cummulative percentage: 99.6%) are selected.

A plot of the scores in the coordinates (PC1, PC2) is also provided:

```
# plotting of scores in the coordinates (PC1, PC2)
plotData = as.data.frame(cbind(n = 1:nrow(pcaData),
                               pc1 = pcaResult$x[,1],
                               pc2 = pcaResult$x[,2]))
ggplot(data = plotData, aes(x = pc1, y = pc2)) +
  geom_point() +
  theme_bw() +
  ggtitle("PC2 vs. PC1")
```

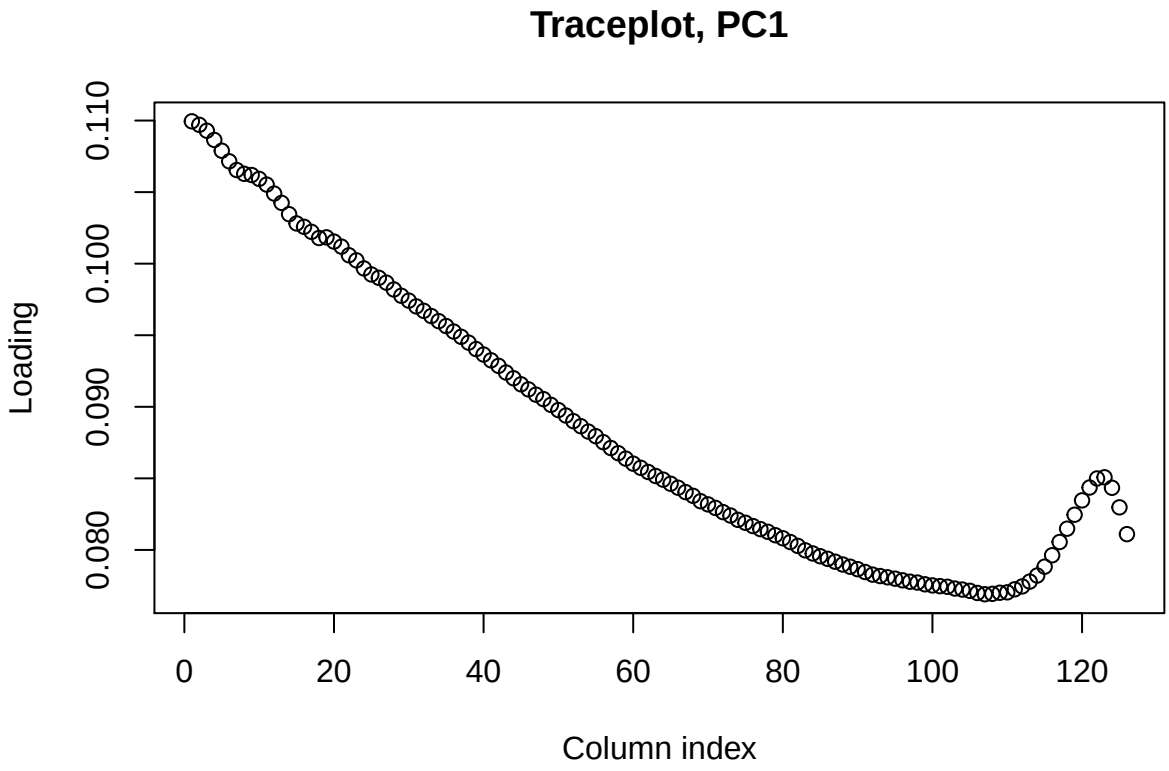


According to this plot, there are a few outlier with a very high value for PC1. These observations seem to be unusual.

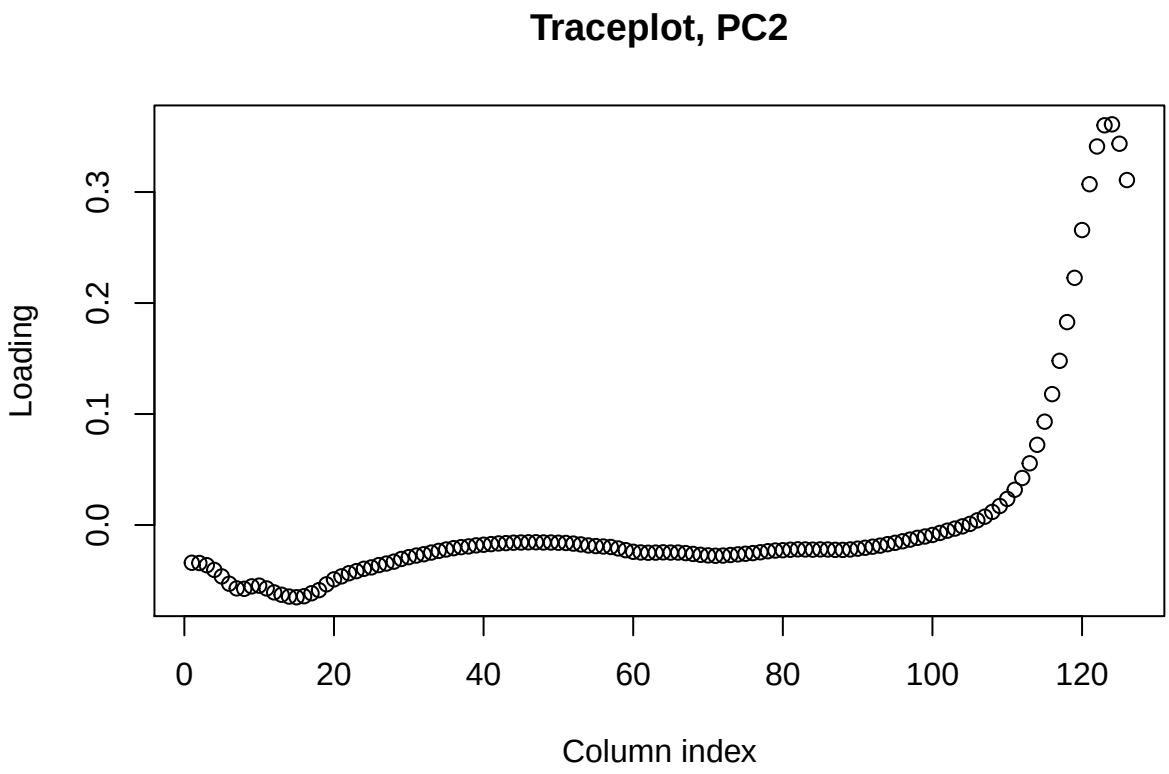
4.2

In the next step, trace plots of the loadings of the components PC1 & PC2 are created:

```
plot(pcaResult$rotation[,1], main="Traceplot, PC1", xlab = "Column index", ylab = "Loading")
```



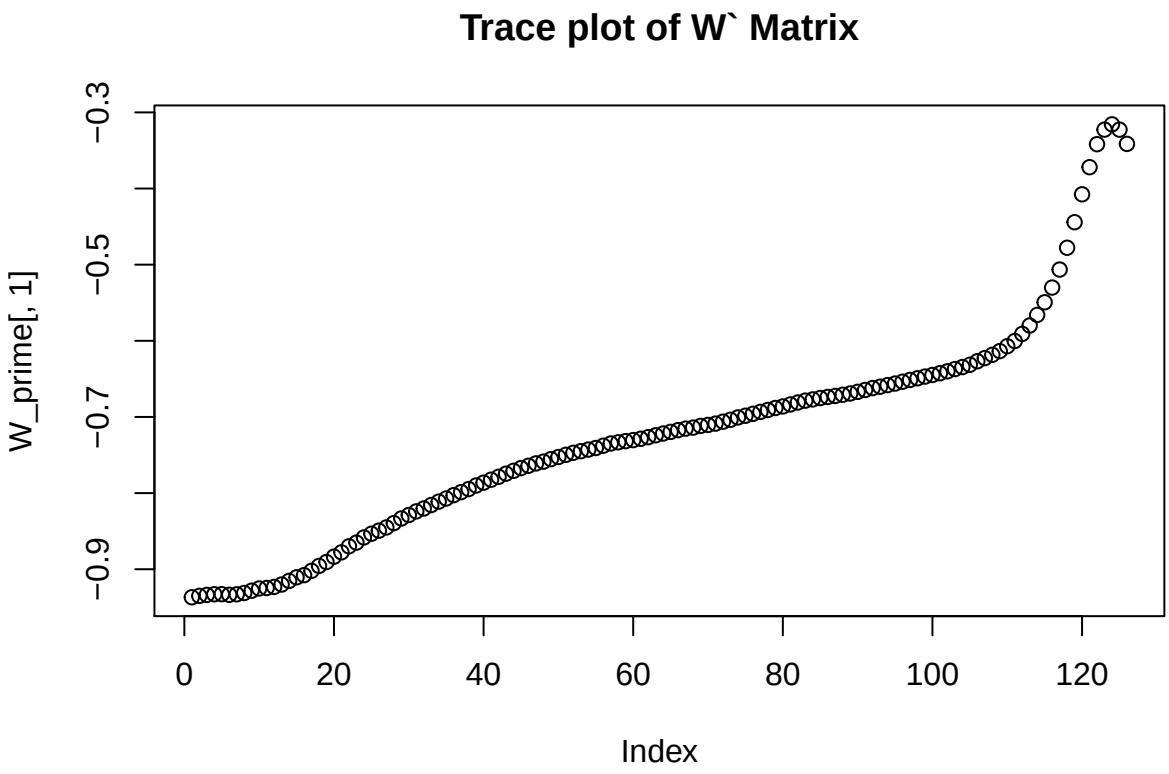

```
plot(pcaResult$rotation[,2], main="Traceplot, PC2", xlab = "Column index", ylab = "Loading")
```



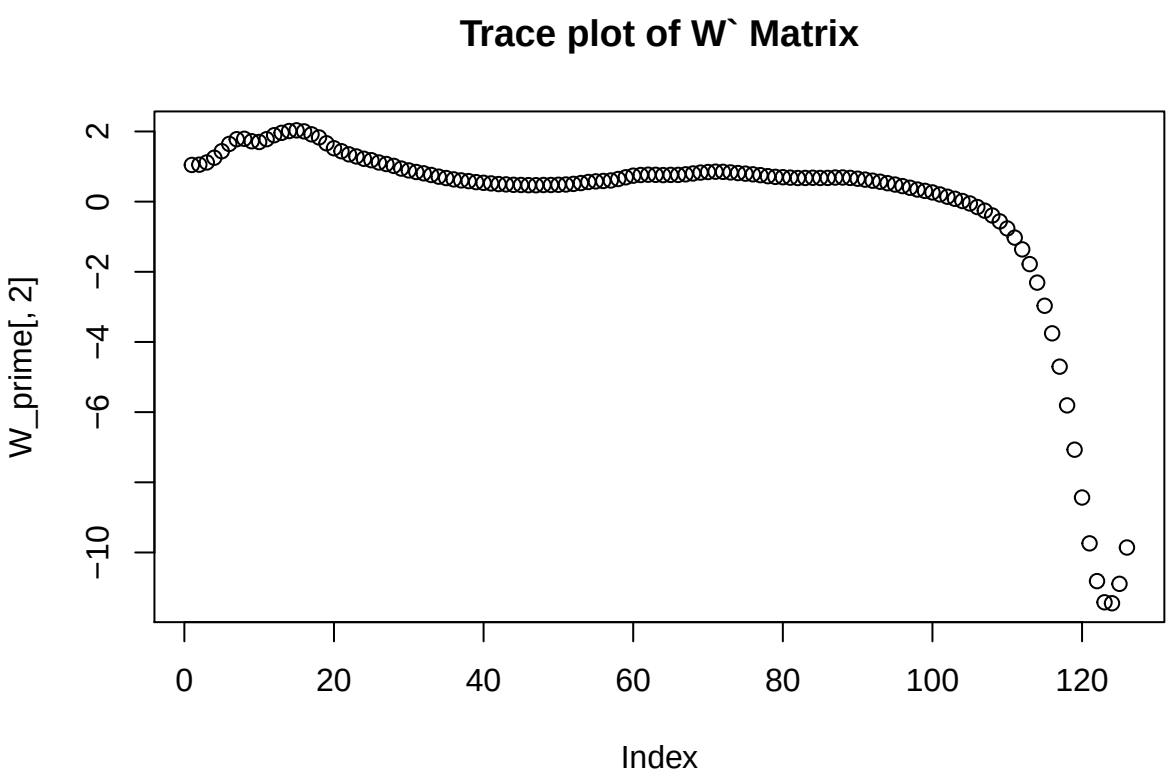
While for PC1, the loadings of the variables do not differ too much (maximum loading ~ 0.11, minimum loading ~ 0.075), for PC the loadings differ extremely. This principal component is explained mainly by the variables with a high index (on the right of the plot). Here, the maximum loading (~ 0.35) is much higher than for most of the other variables.

4.3

```
library(fastICA)
set.seed(12345)
X = as.matrix(pcaData)
icaResult = fastICA(X, 2, alg.typ = "parallel", fun = "logcosh", alpha = 1,
                    method = "R", row.norm = FALSE, maxit = 200, tol = 0.0001,
                    verbose = TRUE)
# tracing plots
W_prime = icaResult$K %*% icaResult$W
plot(W_prime[,1], main = "Trace plot of W` Matrix")
```




```
plot(W_prime[,2], main = "Trace plot of W` Matrix")
```


Trace plot of W` Matrix
W_prime[, 2]
2
0
−2
−4
−6
−10
0
20
40
60
80
100
120
Index