TDDE01: Machine Learning, LAB2

Max Lund

November 28, 2017

1 Assignment 2

The dataset for the first assignment contains information from a private enterprise about their customers. The goal is to predict how well a customer will manage their loans based on a number of predictive features such as marital status, job, age etc. We first split the data into subsets of 50/25/25 of train/validation/test.

```r
data = read.csv("creditscoring.csv", header = TRUE,
                sep = ",", dec = ".", fill = TRUE)
# use half data for training
n = length(data[, 1])
set.seed(12345)
id = sample(1:n, floor(n * 0.5))
train = data[id, ]
remainder = data[-id, ]

# split the remaining data into validation/test
n = length(remainder[, 1])
id = sample(1:n, floor(n * 0.5))
validation = remainder[id, ]
test = remainder[-id, ]
```

1.1 Fitting tree models

Next we fit two models using either the *gini* or the *deviance* measures of impurity. The misclassification rates are reported and the model with the lowest rate is selected.

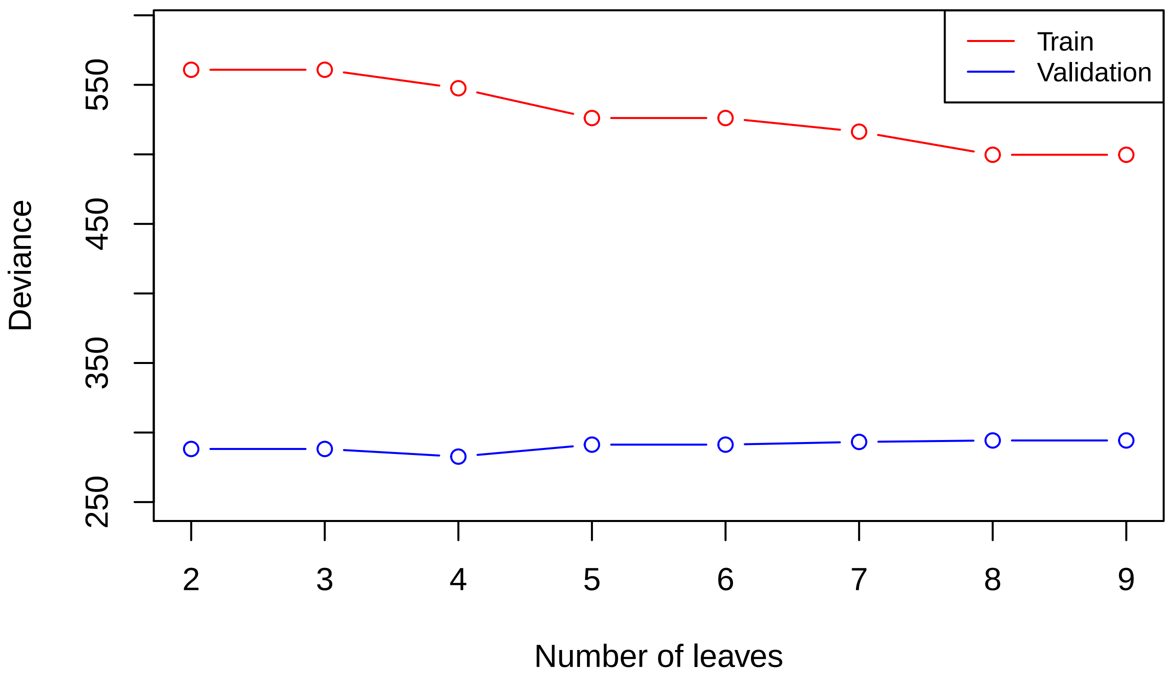
```r
# fit one model using deviance (cross-entropy)
model.deviance = tree(formula = good_bad ~ .,
                      data = train,
                      split = "deviance")

# and another using gini-index
model.gini = tree(formula = good_bad ~ .,
                  data = train,
                  split = "gini")

get_misclass_rate=function(model, data)
{
  cm = get_confusion_matrix(model, data)
  misclass_rate = (cm[1,2] + cm[2,1]) / sum(cm)
  return (misclass_rate)
}

get_confusion_matrix=function(model, data)
{
  prediction = predict(model, newdata = data, type = "class")
  cm = table(prediction, data$good_bad)
  return (cm)
}

cat("misclass rate using 'deviance' for train data: ",
    get_misclass_rate(model.deviance, train),
    "\nmisclass rate using 'deviance' for test data: ",
    get_misclass_rate(model.deviance, test),
    "\nmisclass rate using 'gini' for train data: ",
    get_misclass_rate(model.gini, train),
    "\nmisclass rate using 'gini' for test data: ",
    get_misclass_rate(model.gini, test))
```

```
## misclass rate using 'deviance' for train data:  0.212
## misclass rate using 'deviance' for test data:  0.284
## misclass rate using 'gini' for train data:  0.23
## misclass rate using 'gini' for test data:  0.34
```

Since the deviance method had lower misclassification rates, this model is chosen for the subsequent steps.

Next we use the training and validation sets to choose the optimal tree depth. This is done by measuring the deviance for when pruning the tree to have a set number of terminal nodes (leaves) in the range 2-9.

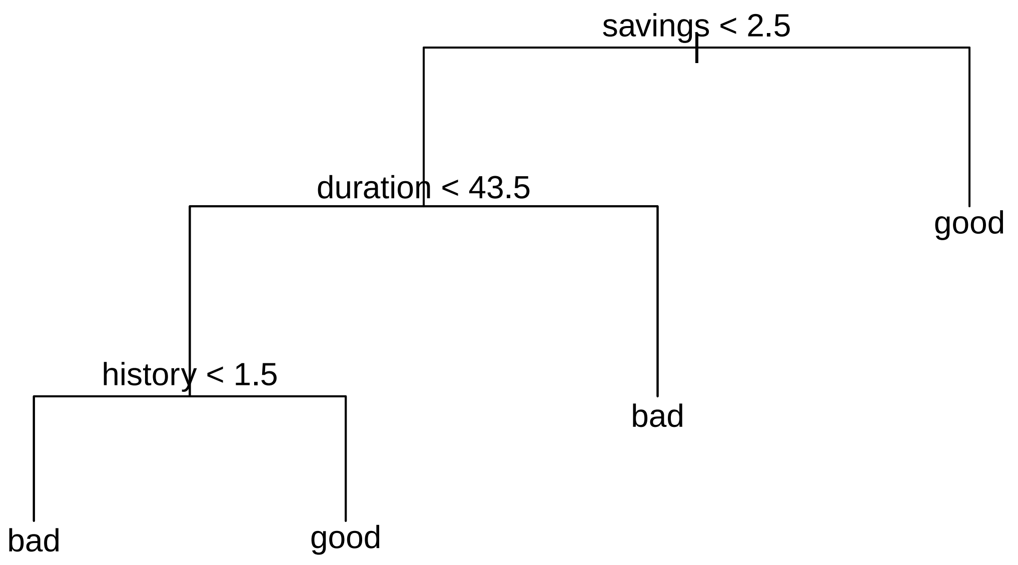
```
train_score = numeric(9)
validation_score = numeric(9)

# try number of terminal nodes in range 2-9 to find
# optimal depth, i.e tree with lowest deviance
for (i in 2:9)
{
  pruned_tree = prune.tree(model.deviance, best = i)
  prediction = predict(pruned_tree, newdata = validation, type = "tree")
  train_score[i] = deviance(pruned_tree)
  validation_score[i] = deviance(prediction)
}

plot(2:9, train_score[2:9],
     type="b", col="red", ylim=c(250, 590),
     xlab="Number of leaves", ylab="Deviance")
points(2:9, validation_score[2:9],
       type="b", col="blue")
legend("topright", legend=c("Train", "Validation"),
       col=c("red", "blue"), lty=1, cex=0.8)
```



```
# since i = 2..9, optimal i is equal to index + 1
optimal.leaves = which.min(validation_score[2:9]) + 1
# prune tree to optimal number of leaves
optimal.tree = prune.tree(model.deviance, best = optimal.leaves)
# get variables selected of optimal tree
summary(optimal.tree)
```

```
##
## Classification tree:
## snip.tree(tree = model.deviance, nodes = c(5L, 3L, 9L))
## Variables actually used in tree construction:
## [1] "savings"  "duration" "history"
## Number of terminal nodes:  4
## Residual mean deviance:  1.117 = 547.5 / 490
## Misclassification error rate: 0.251 = 124 / 494
```

```
# Variables actually used in tree construction:
# "savings"  "duration" "history"

# print the optimal tree
# depth = 3 (if root depth = 0)
plot(optimal.tree)
text(optimal.tree)
```



```r
# get misclassification rate for optimal tree on test data
optimal.misclass = get_misclass_rate(optimal.tree, test)
cat("misclass rate of optimal tree:", optimal.misclass)
```

```
## misclass rate of optimal tree: 0.26
```

The optimal tree has 4 leaves, with a depth of 3. The variables selected by the tree was Savings, Duration, and History, with the tree splitting for different values of these variables to make predictions.

1.2 Fitting a Naive Bayes classifier

Next we use the training data to perform classification using a *Naive Bayes classifier*, measuring its misclassification rates and confusion matrices on the test and traninig data.

```r
# fit a naive bayes model to train data
model.bayes = naiveBayes(formula = good_bad ~ .,
                         data = train)

# misclass-rate is higher for train than test..
cat("misclass rate for naive bayes model on train data:",
    get_misclass_rate(model.bayes, train),
    "\nmisclass rate for naive bayes model on test data:",
    get_misclass_rate(model.bayes, test))
```

```
## misclass rate for naive bayes model on train data: 0.3
## misclass rate for naive bayes model on test data: 0.32
```

```r
cat("CM for naiveBayes (train data):")
```

```
## CM for naiveBayes (train data):
```

```r
print(get_confusion_matrix(model.bayes, train))
```

```
##
## prediction bad good
##       bad   95   98
##       good  52  255
```

```
cat("CM for naiveBayes (test data):")
```

```
## CM for naiveBayes (test data):
```

```
print(get_confusion_matrix(model.bayes, test))
```

```
##
## prediction bad good
##       bad   47   49
##       good  31  123
```

From the results we can see that the Naive Bayes classifier has a higher misclassification rate than our optimal tree model.

For the final task we use a modified loss matrix for our naive bayes classifier where the penalty for misclassifying a future loan management as good when the true classification was actually bad is 10 times higher than making the inverse misclassification. To calculate the new misclassification rate and confusion matrix using this loss matrix, we get the raw probabilities for each class from our classifier, and apply the new classification policy to our probabilities.

```
# get raw probabilities for both classes from naiveBayes classifier
raw.train = predict(model.bayes, newdata = train, type = "raw")
raw.test = predict(model.bayes, newdata = test, type = "raw")

# predicting 'good' must be 10x more probable than bad
# to make the prediction 'good' with new loss matrix
preds.train = (raw.train[, 2] / raw.train[, 1]) > 10
preds.test = (raw.test[, 2] / raw.test[, 1]) > 10

# convert booleans to good/bad labels
preds.train[which(preds.train == TRUE)] = "good"
preds.train[which(preds.train == FALSE)] = "bad"
preds.test[which(preds.test == TRUE)] = "good"
preds.test[which(preds.test == FALSE)] = "bad"

# CM for train and test
cm.train = table(preds.train, train$good_bad)
cm.test = table(preds.test, test$good_bad)

cat("CM for naiveBayes using new loss matrix (train data):")
```

```
## CM for naiveBayes using new loss matrix (train data):
```

```
print(cm.train)
```

```
##
## preds.train bad good
##        bad  137  263
##        good  10   90
```

```
cat("CM for naiveBayes using new loss matrix (test data):")
```

```
## CM for naiveBayes using new loss matrix (test data):
```

```
print(cm.test)
```

```
##
## preds.test bad good
##       bad   70  131
##       good   8   41
```

```
# misclass-rate for train and test
misclass.train = (cm.train[1,2] + cm.train[2,1]) / sum(cm.train)
misclass.test = (cm.test[1,2] + cm.test[2,1]) / sum(cm.test)

cat("misclass rate naiveBayes + new loss matrix (train data):",
    misclass.train,
    "\nmisclass rate naiveBayes + new loss matrix (test data):",
    misclass.test)
```

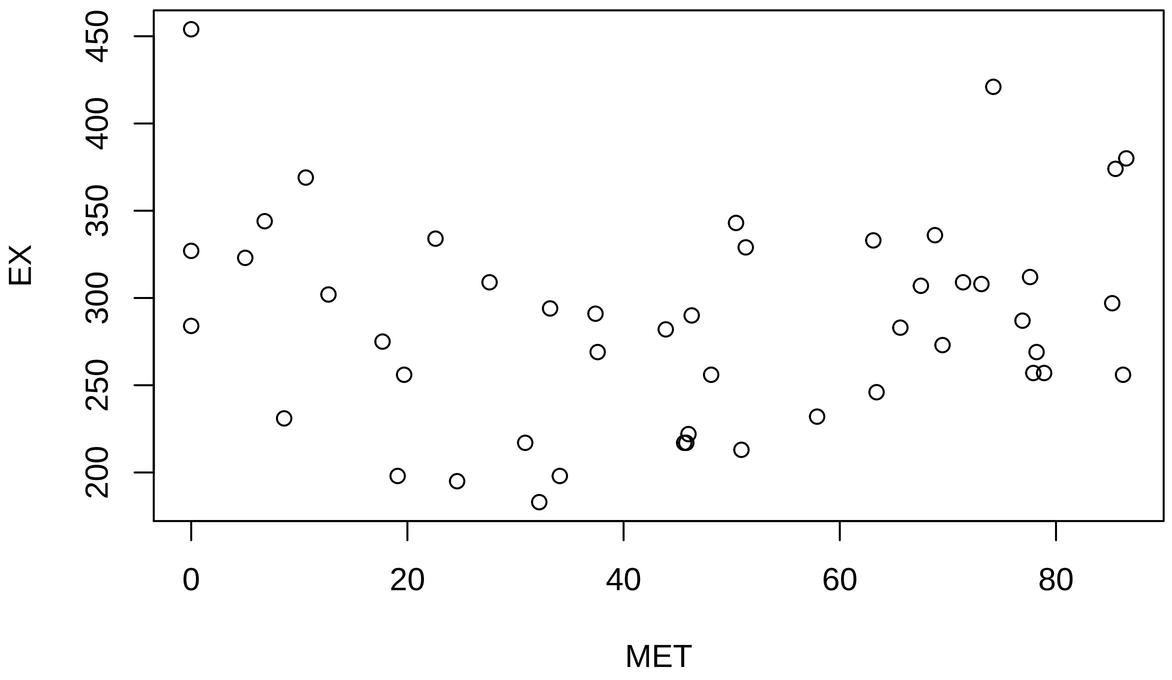
```
## misclass rate naiveBayes + new loss matrix (train data): 0.546
## misclass rate naiveBayes + new loss matrix (test data): 0.556
```

As we can see our misclassification rates have increased from the previous model, but our rates have lowered for classifying a customers future loan management as good when the true classification is bad, which was the point of the modified loss matrix.

2 Assignment 3

The next assignment examines how the per capita state and local public expenditures *(EX)* depend on the percentage of population living in standard metropolitan areas *(MET)*. We order our data with respect to MET, and plot EX vs. MET.

```r
set.seed(12345)
data = read.csv("State.csv", header = TRUE,
                sep = ";", dec = ",", fill = TRUE)
# sort data by variable MET (desc)
data.ordered = data[order(data$MET), ]
# plot EX vs MET
plot(data.ordered$MET, data.ordered$EX, xlab="MET", ylab="EX")
```



2.1 Fitting a regression tree

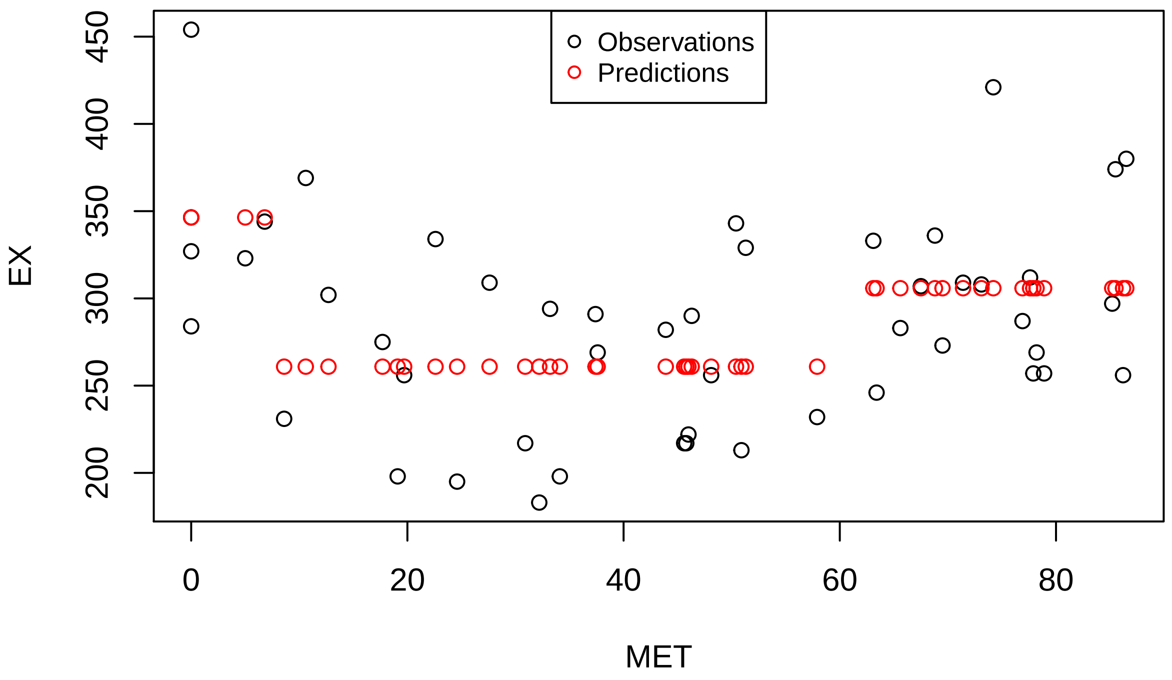
There doesn't seem to be any clear pattern in the relationship between the EX and MET variables, so using linear or polynomial regression won't work. We try using a regression tree model using EX as our target and MET as our predictor variable. Cross-validation is used, and the minimum number of observation in any leaf is set to 8.

```r
# fit a tree model, #observations in leaf >= 8
model = tree(formula = EX ~ MET,
             data = data.ordered,
             control = tree.control(nrow(data.ordered), minsize = 8))
# perform cross-validation on model
model.cv = cv.tree(model)
# select optimal number of leaves
best.size = model.cv$size[which(model.cv$dev==min(model.cv$dev))]
# prune tree using selected num of leaves from CV
model.optimal = prune.tree(model, best = best.size)
# report the selected tree
```

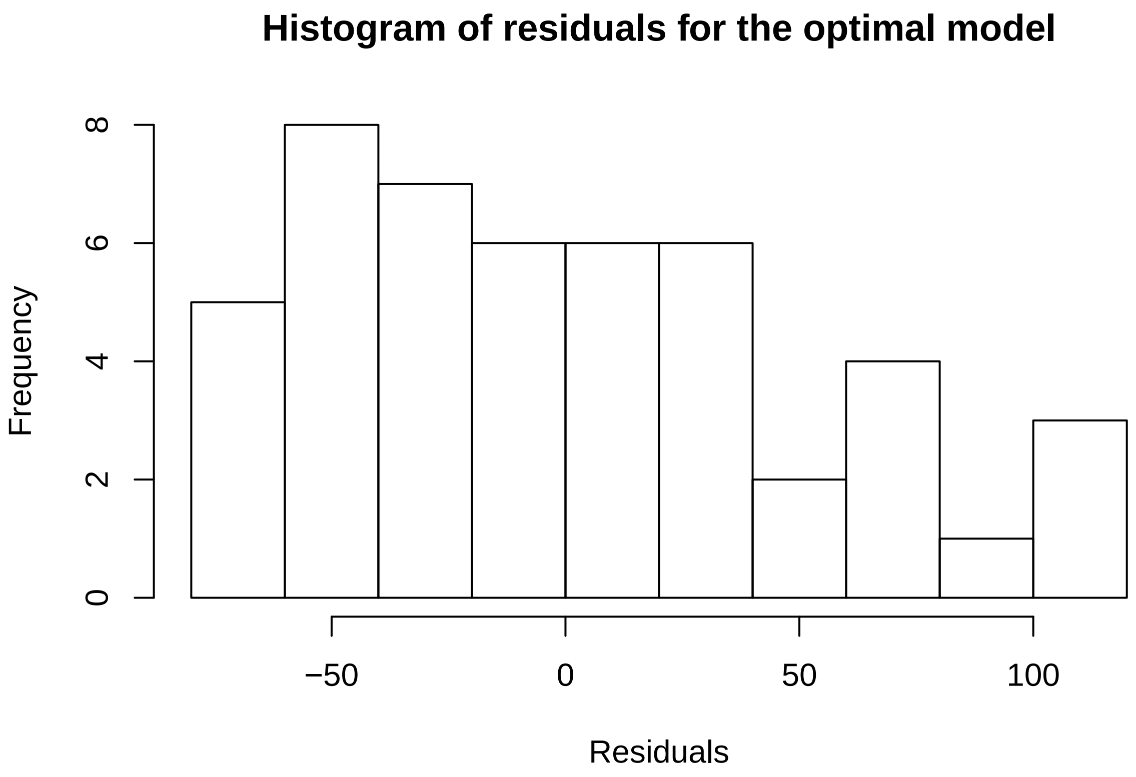
```r
summary(model.optimal)
```

```
##
## Regression tree:
## snip.tree(tree = model, nodes = c(7L, 6L))
## Number of terminal nodes:  3
## Residual mean deviance:  2698 = 121400 / 45
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -77.88  -43.88   -4.88    0.00   30.13  115.20
```

```r
#plot original and fitted data
fitted_data = predict(model.optimal, newdata = data.ordered)
plot(data.ordered$MET, data.ordered$EX, col="black",
     xlab="MET", ylab="EX")
points(data.ordered$MET, fitted_data, col="red")
legend("top", legend=c("Observations", "Predictions"),
       col=c("black", "red"), pch=1, cex=0.8)
```



```r
# histogram of residuals
hist(resid(model.optimal), xlab="Residuals", main="Histogram of residuals for the optimal model")
```

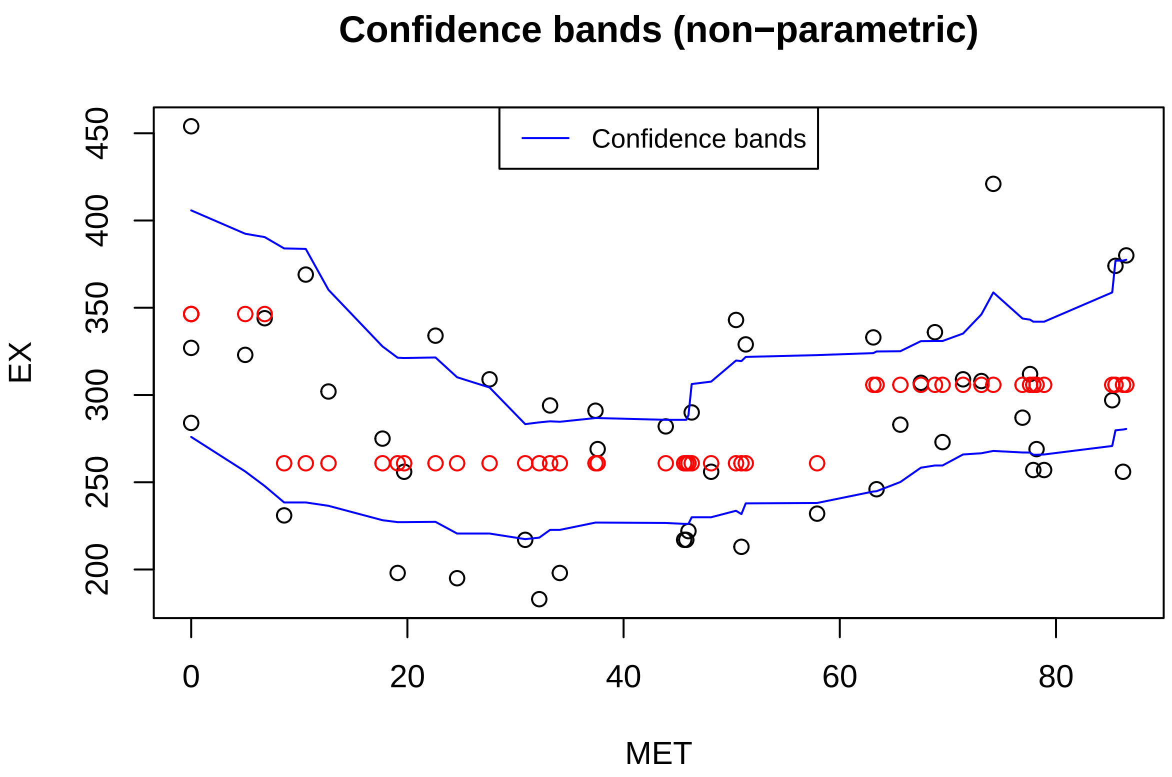


As we can see the predicted values only have three disctinct values for the feature *EX*. This is because the optimal model selected only has three leaf nodes, which makes it difficult to fit the model to the scattered observations. Looking at the histogram we can also see that the distribution of the residuals is not a Gaussian distribution, and the most frequent value for the residuals is around -50, which would also suggest that the model is not a good fit for the data.

2.2 Estimating uncertainty

In order to measure the uncertainty in the predictions of our model, we compute the 95% confidence bands using *bootstrapping*. Since we don't know the underlying distribution of the observations, we use *non-parametric bootstrapping*, which resamples the data with replacement.

```
# function for 'statistics' in the bootstrap function
nonparametric=function(data, index)
{
  sample = data[index, ]
  model = tree(formula = EX ~ MET,
               data = sample,
               control = tree.control(nrow(sample), minsize = 8))
  model.pruned = prune.tree(model, best = best.size)
  prediction = predict(model.pruned, newdata = data.ordered)
  return (prediction)
}
boot.nonparam = boot(data = data.ordered,
                     statistic = nonparametric,
                     R = 1000)
boot.nonparam.cb = envelope(boot.nonparam, level = 0.95)
# plot MET vs EX, predictions and
# confidence bands for model's predictions
plot(data.ordered$MET, data.ordered$EX, col = "black",
     xlab="MET", ylab="EX", main="Confidence bands (non-parametric)")
points(data.ordered$MET, fitted_data, col = "red")
lines(data.ordered$MET, boot.nonparam.cb$point[1, ], col="blue")
lines(data.ordered$MET, boot.nonparam.cb$point[2, ], col="blue")
legend("top", legend=c("Confidence bands"),
       col=c("blue"), lty=1, cex=0.8)
```



We can see that the confidence band is bumpy along each prediction. This is because we have discrete predictions, so the confidence bands will also have discrete points. Since more than 5% of the observations are outside the confidence bands, the model doesn't seem to be a good fit.

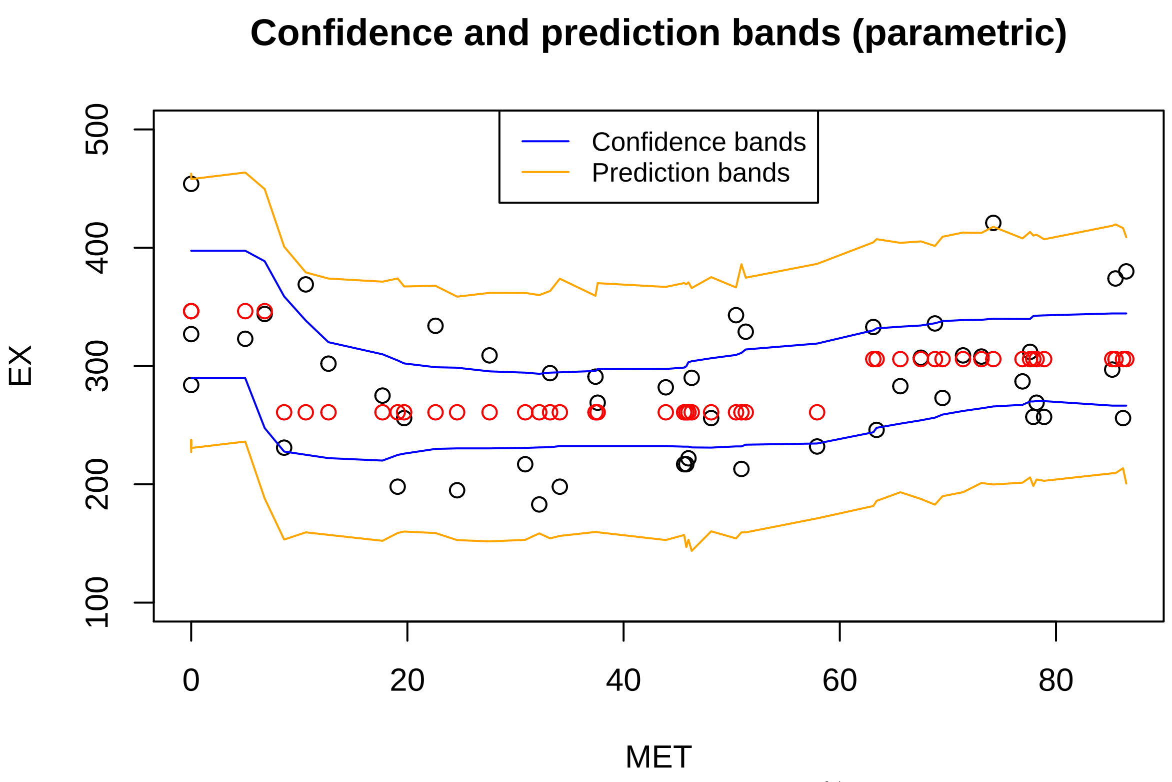
Next we assume that the observations fall under the distribution:

$$Y \sim N(\mu_i, \theta^2) \tag{1}$$

This allows us to use *parametric bootstrapping*, in which we draw new observations from the assumed underlying distribution instead of resampling from the same data. We also calculate and plot the prediction bands, which is the 95% confidence interval for the assumed distribution.

```
parametric=function(data)
{
  model = tree(formula = EX ~ MET, data = data,
               control = tree.control(nrow(data), minsize = 8))
  model.pruned = prune.tree(model, best = best.size)
  prediction = predict(model.pruned, newdata = data.ordered)
  return (prediction)
}

rng=function(data, mle)
{
```

```r
  new_data = data.frame(EX = data$EX, MET = data$MET)
  n = length(data$EX)
  new_data$EX = rnorm(n, predict(mle, newdata = data), sd(resid(mle)))
  return(new_data)
}

prediction=function(data)
{
  model = tree(formula = EX ~ MET,
               data = data,
               control = tree.control(nrow(data), minsize = 8))
  model.pruned = prune.tree(model, best = best.size)
  preds = predict(model.pruned, newdata = data.ordered)
  n = length(data.ordered$EX)
  preds_ = rnorm(n, preds, sd(resid(model.optimal)))
  return (preds_)
}

boot.param = boot(data = data.ordered,
                  statistic = parametric,
                  R = 1000,
                  mle = model.optimal,
                  ran.gen = rng,
                  sim = "parametric")

boot.param.preds = boot(data = data.ordered,
                        statistic = prediction,
                        R = 1000,
                        mle = model.optimal,
                        ran.gen = rng,
                        sim = "parametric")

boot.param.cb = envelope(boot.param, level = 0.95)
boot.param.pb = envelope(boot.param.preds, level = 0.95)

# plot cb:s and pb:s for parametric bootstrap
plot(data.ordered$MET, data.ordered$EX, col="black", ylim=c(100, 500),
     xlab="MET", ylab="EX", main="Confidence and prediction bands (parametric)")
points(data.ordered$MET, fitted_data, col = "red")
lines(data.ordered$MET, boot.param.cb$point[1, ], col="blue")
lines(data.ordered$MET, boot.param.cb$point[2, ], col="blue")
lines(data.ordered$MET, boot.param.pb$point[1, ], col="orange")
lines(data.ordered$MET, boot.param.pb$point[2, ], col="orange")
legend("top", legend=c("Confidence bands", "Prediction bands"),
       col=c("blue", "orange"), lty=1, cex=0.8)
```



As we can see the prediction bands does seem to cover more than 95% of the observations. But since the histogram of the residuals earlier showed that the distribution did not look Gaussian, it is probably more suitable to use the non-parametric bootstrapping.

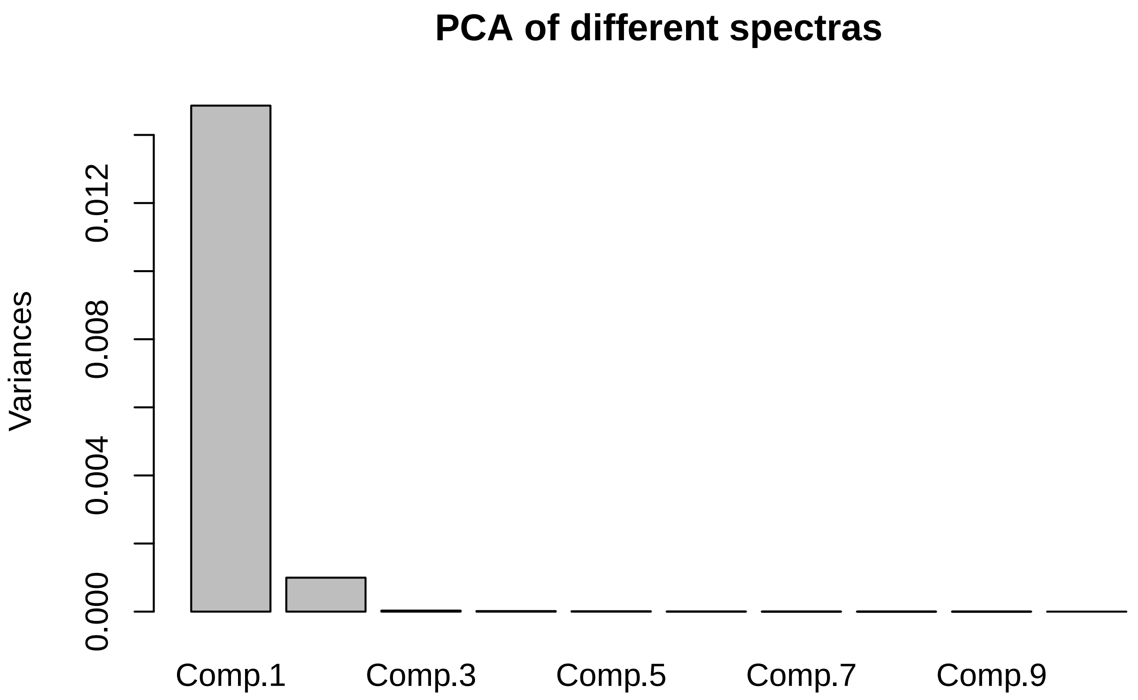
3 Assignment 4

In the final assignment, we are examining a dataset containing near-infrared spectra and viscosity levels for a collection of diesel fuels.

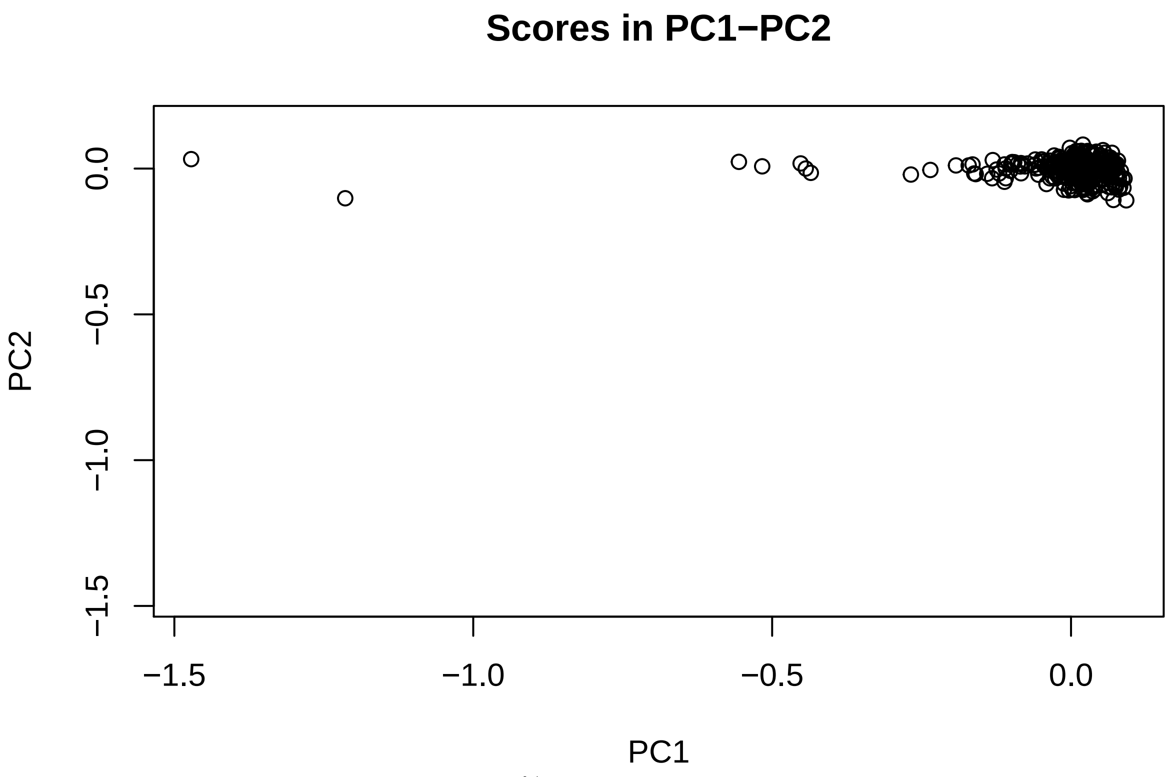
3.1 Principle component analysis

We begin by doing a *principle component analysis* (PCA), where we find which components that explain most of the variance seen in the data.

```r
set.seed(12345)
data = read.csv("NIRSpectra.csv", header = TRUE,
                sep = ";", dec = ",", fill = TRUE)
features = princomp(data[, -ncol(data)])
plot(features, main="PCA of different spectras")
```



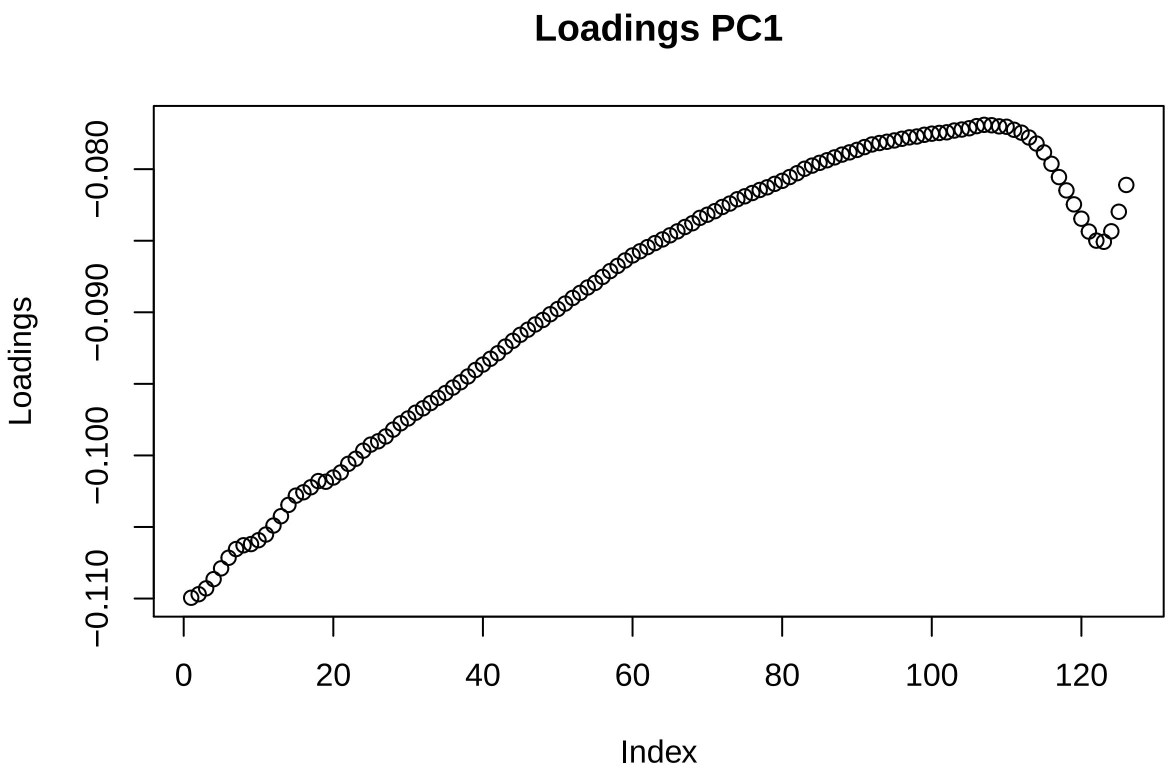
```r
# proportion of variance in PC1 + PC2 = 0.933 + 0.062 = 0.996 > 0.99
# -- we choose PC1 and PC2.

# plot of PC1 vs PC2
x_lim = c(min(features$scores[ ,1]), max(features$scores[ ,1]))
y_lim = c(min(features$scores[ ,1]), 0.15)
plot(features$scores[ ,1], features$scores[ ,2], lim=x_lim, ylim=y_lim,
     main="Scores in PC1-PC2", xlab="PC1", ylab="PC2")
```



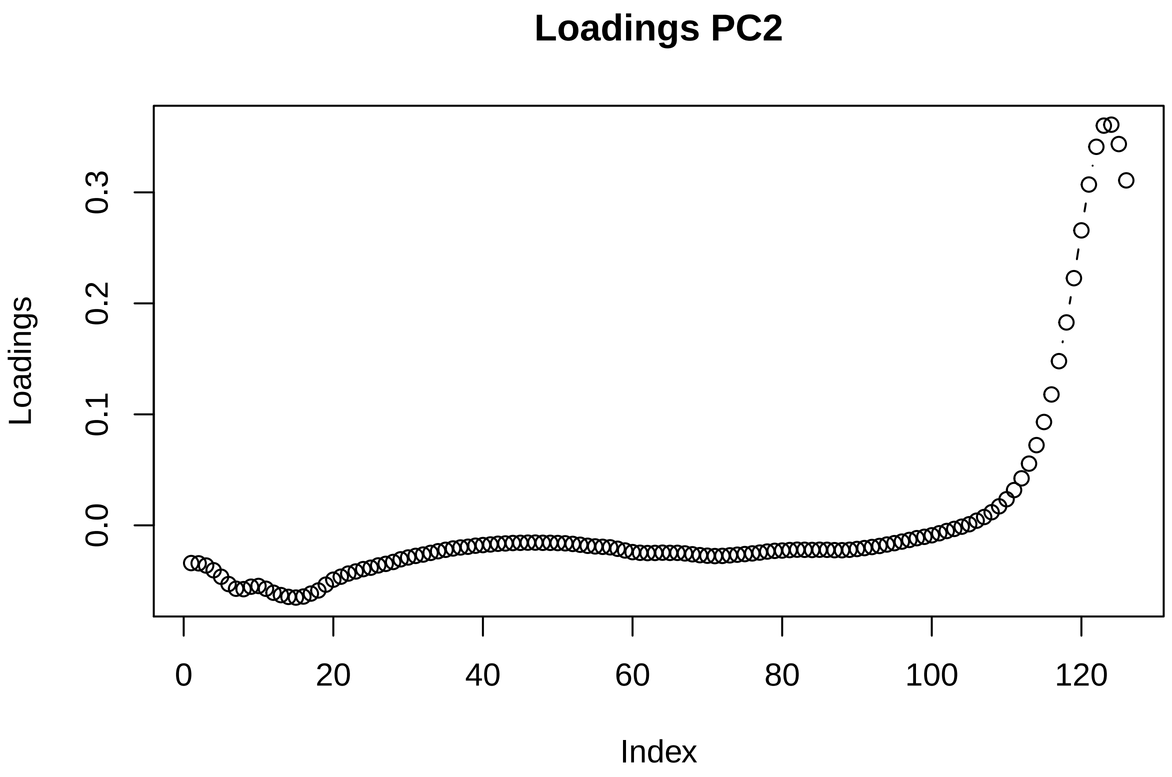
Since PC1 and PC2 together explain over 99% of the variance seen in the data, we select these two principal components. From the plot we can also make out some outliers of unsual diesel fuels.

Next we make trace plots of the loadings of PC1 and PC2, in order to examine which features contribute the most to the variance seen in the principle component.

```
# loads, how much each feature contribute to the variance seen in a PC
loads = loadings(features)
plot(loads[, 1], type="b", ylab="Loadings", main="Loadings PC1")
```



```
plot(loads[, 2], type="b", ylab="Loadings", main= "Loadings PC2")
```



We can see that PC2 has many loadings at values at or close to 0, which tells us that there are many features that doesn't contribute to the variance seen in PC2. The few points at the top of the graph suggests that there are a few features that account for most of the variance in PC2.

3.2 Independent component analysis

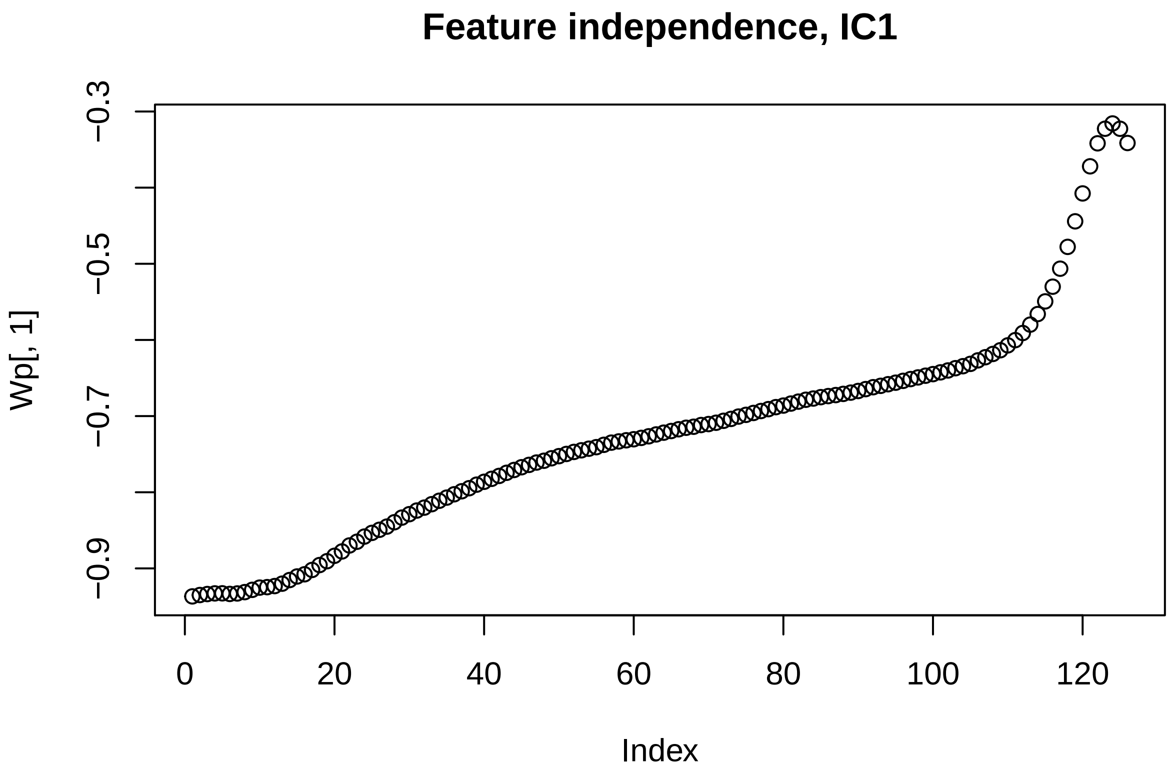
Next we perform an independent component analysis using the same number of components (two) as earlier. In contrast to the principle component analysis, we assume that the components are independent. We calculate the independence of the features within the independent components and plot the traces of the results.

```
res = fastICA(data[ ,-ncol(data)], 2, fun = "logcosh", alpha = 1.0,
              row.norm = FALSE, maxit = 200, tol = 0.0001, verbose = TRUE)
```

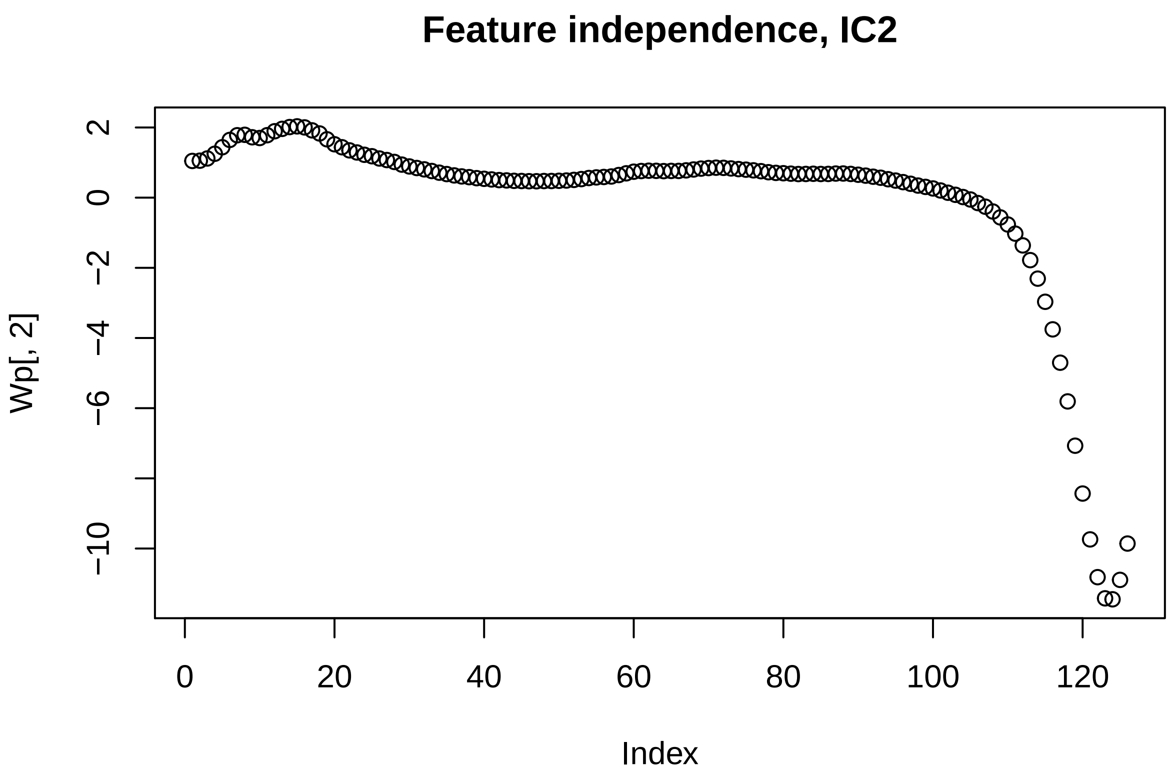
```
## Centering

## Whitening

## Symmetric FastICA using logcosh approx. to neg-entropy function

## Iteration 1 tol = 0.01930239

## Iteration 2 tol = 0.01303959

## Iteration 3 tol = 0.002393582

## Iteration 4 tol = 0.0006708454

## Iteration 5 tol = 0.0001661602

## Iteration 6 tol = 3.521604e-05
```

```
# K = pre-whitening matrix that projects data onto the first 2 principal components.
# W = estimated un-mixing matrix
Wp = res$K %*% res$W
```

```
# plot of feature independence in the two ICs
plot(Wp[, 1], main="Feature independence, IC1")
```



```r
plot(Wp[, 2], main="Feature independence, IC2")
```

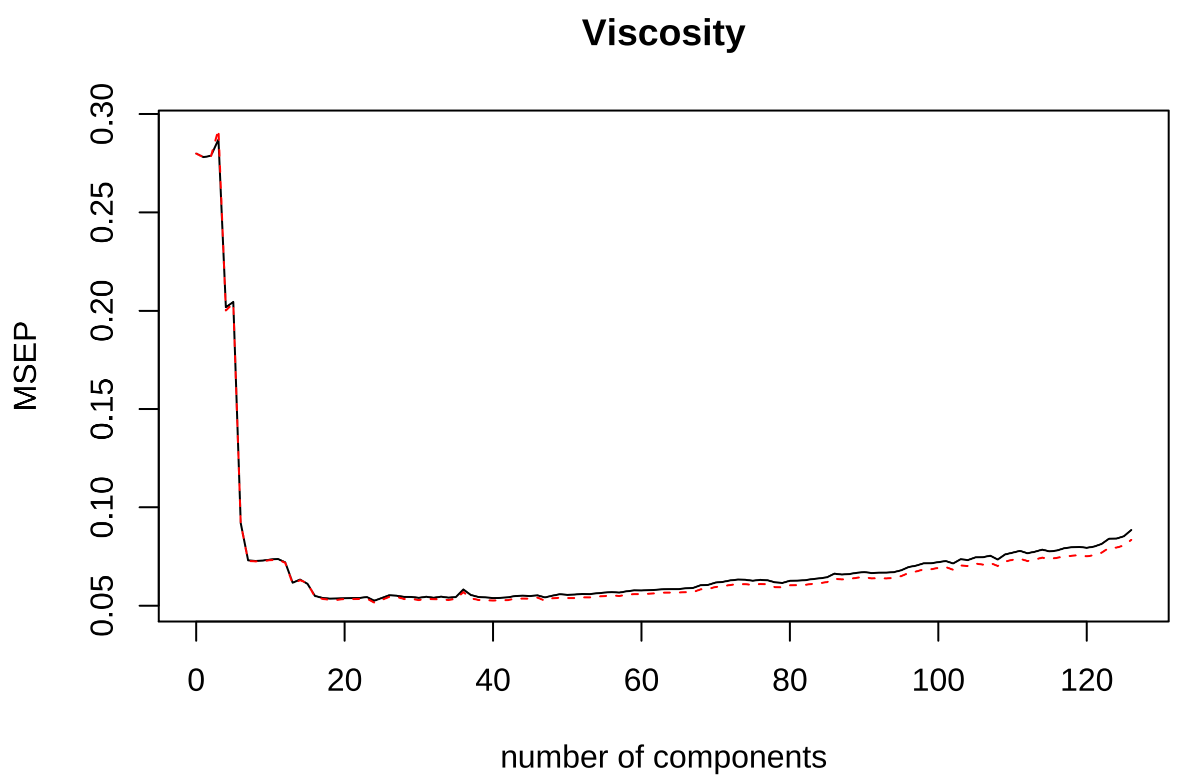


If we compare the trace plots with those of the loadings of our PCA, we can see that they look like the inverse of each other. This is because ICA measures the independence of features while PCA measures their correlation, which is why the results look like mirror opposites.

3.3 Principal component regression

Finally, we fit a PCR model in which the number of components is selected by cross validation to the data.

```r
# fit a PCR model
pcr.fit = pcr(formula = Viscosity ~ .,
              data = data,
              validation = "CV")
# plot predicted mean squared error dependent on number of components
validationplot(pcr.fit, val.type = "MSEP")
```



According to the PCR model, the optimal number of principle components for the lowest predicted MSE looks to be around 20.