Computer Lab 1 (732A99 Machine Learning)

Lennart Schilling (lensc874)

22 November 2018

Assignment 1

1.1

At first, the data from the Excel file *spambase.xlsx* will be imported and splitted into train and test data (50%:50%)

```
# Importing data
library(readxl)
data = read_excel("spambase.xlsx")

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

1.2

Using the train data, a logistic regression model will be created. Analysing the p-values of the coefficients, it can be seen which independent variables have a significant influence on the dependent variable *Spam*. For the sake of clarity, this is not explicitly stated in this report.

```
# Fitting model
logitModel = suppressWarnings(glm(Spam ~ ., data = train, family = binomial))
summary(logitModel)
```

In the next step, the *logitModel* will be used to classify emails of the training and test data. To prevent duplicate code in 1.3, the *classificationLogit*-function was coded. Giving data and a threshold as an input, a list with the specified threshold to decide which probabilities lead to a spam classification, the confusion matrix and the misclassification rate will be returned.

```
# Classifying & evaluating results
classificationLogit = function(data, threshold = 0.5) {
  # Classifying emails with the model
  yFit = predict(logitModel,
                 newdata = data[,!colnames(data) %in% "Spam"],
                 type='response')
  yFit = ifelse(yFit > threshold, 1, 0)
  # Evaluating classification results
  confusionMatrix = table(y = data$Spam, yFit)
  misclassificationRate <- mean(yFit != data$Spam)
  # Returning results
  return(
    list(
      threshold = threshold,
      confusionMatrix = confusionMatrix,
      misclassificationRate = misclassificationRate
```

```
    )
  )
}
```

Using the train data and the default threshold (0.5) as the input leads to the following confusion matrix and misclassification rate.

```
classificationLogitTrain = classificationLogit(data = train)
classificationLogitTrain$confusionMatrix
```

```
##    yFit
## y     0   1
##   0 803 142
##   1  81 344
```

```
classificationLogitTrain$misclassificationRate
```

```
## [1] 0.1627737
```

Instead, using the test data and the default threshold (0.5) as the input leads to the following confusion matrix and misclassification rate.

```
classificationLogitTest = classificationLogit(data = test)
classificationLogitTest$confusionMatrix
```

```
##    yFit
## y     0   1
##   0 791 146
##   1  97 336
```

```
classificationLogitTest$misclassificationRate
```

```
## [1] 0.1773723
```

It can be seen that the model performs about equally well for both data. The misclassification rate is slightly better for the training data (16.2%) than for the training data (17.7%). This is an indication that there is not too much overfitting on the training data.

1.3

Now we are changing the classification principle and therefore the input threshold from 0.5 to 0.9.

Using the train data and the threshold = 0.9 as the input leads to the following confusion matrix and misclassification rate.

```
classificationLogitTrainAdjThreshold = classificationLogit(data = train, threshold = 0.9)
classificationLogitTrainAdjThreshold$confusionMatrix
```

```
##    yFit
## y     0   1
##   0 944   1
##   1 419   6
```

```
classificationLogitTrainAdjThreshold$misclassificationRate
```

```
## [1] 0.3065693
```

Using the test data and the threshold = 0.9 as the input leads to the following confusion matrix and misclassification rate.

```
classificationLogitTestAdjThreshold = classificationLogit(data = test, threshold = 0.9)
classificationLogitTestAdjThreshold$confusionMatrix
```

```
##    yFit
## y     0   1
##   0 936   1
##   1 427   6
```

```
classificationLogitTestAdjThreshold$misclassificationRate
```

```
## [1] 0.3124088
```

In both cases it can be seen that the classification quality decreases a lot (misclassification rates about 30%). Because the threshold is now much higher than before, the number of false negative predictions has increased strongly.

1.4

In the following, the standard classifier *kknn()* was used to predict spam mails. Again, to prevent duplicate code in 1.5, the *classificationKknn*-function was coded. Giving data, number of k and a threshold as an input, a list with the same elements as the *classificationLogit*-function is returned.

```
library(kknn)
# Classifying & evaluating results
classificationKknn = function(data, k, threshold = 0.5) {
  # Classifying emails
  kknnModel <- kknn(formula = Spam ~ .,
                    train = train,
                    test = data,
                    k = k)
  kknnModel$fitted.values = ifelse(kknnModel$fitted.values > threshold, 1, 0)
  # Evaluating classification results
  confusionMatrix = table(y = data$Spam, yFit = kknnModel$fitted.values)
  misclassificationRate <- mean(kknnModel$fitted.values != data$Spam)
  # Returning results
  return(
    list(
      threshold = threshold,
      confusionMatrix = confusionMatrix,
      misclassificationRate = misclassificationRate
    )
  )
}
```

Using the train data and k = 30 as the input leads to the following misclassification rate.

```
classificationKnnTrain = classificationKknn(data = train, k = 30)
classificationKnnTrain$misclassificationRate
```

```
## [1] 0.1722628
```

Instead, using the test data and k = 30 as the input leads to the following misclassification rate.

```
classificationKnnTest = classificationKknn(data = test, k = 30)
classificationKnnTest$misclassificationRate
```

```
## [1] 0.329927
```

Here, a big difference between the prediction power of the model related to the train data (misclassification rate: 17.2%) and the test data (misclassification rate: 32.9%) can be observed. This leads to the assumption that the model is overfitting on the training data. Compared to the results of the logistic regression model with the threshold = 0.5, this model does not deliver such accurate predictions.

1.5

Now we are changing the k from 30 to 1.

Using the train data and k = 1 as the input leads to the following misclassification rate.

```
classificationKnnTrain = classificationKknn(data = train, k = 1)
classificationKnnTrain$misclassificationRate
```

```
## [1] 0
```

Using the test data and k = 1 as the input leads to the following misclassification rate.

```
classificationKnnTest = classificationKknn(data = test, k = 1)
classificationKnnTest$misclassificationRate
```

```
## [1] 0.3459854
```

This example shows very clearly that the model is strongly overfitted on the training data. While it classifies every mail for the training data correctly, the misclassification rate for the test data is almost 35%. With k = 1, the classification depends only on the nearest neighbor (the value of the dependent variable of this observation in the training data where the independent variables have the lowest distance to the obsvervation which shall be classified) which leads to a much higher dependency on the training data.

Assignment 3

3.1

```
featureSelection = function(X, Y, Nfolds) {
  # Checking input
  if (!is.matrix(X)) {
    stop("X must be of class 'matrix'.")
  }
  if (!is.numeric(Y)) {
    stop("Y must be a numeric vector.")
  }
  if (!is.numeric(Nfolds) | length(Nfolds) != 1) {
    stop("Nfolds must be a numeric of length 1.")
  }
  if (nrow(X) != length(Y)) {
    stop("X and Y must have same number of observations.")
  }
  # Adding intercept variable as last column to X
  X = cbind(X, Intercept = 1)
  # Creating folds
  set.seed(12345)
  obs = 1:nrow(X)
  folds = list()
    # Allocating observations equally to folds
    for (i in 1:Nfolds) {
      folds[[i]] = sample(obs, floor(nrow(X)/Nfolds))
```

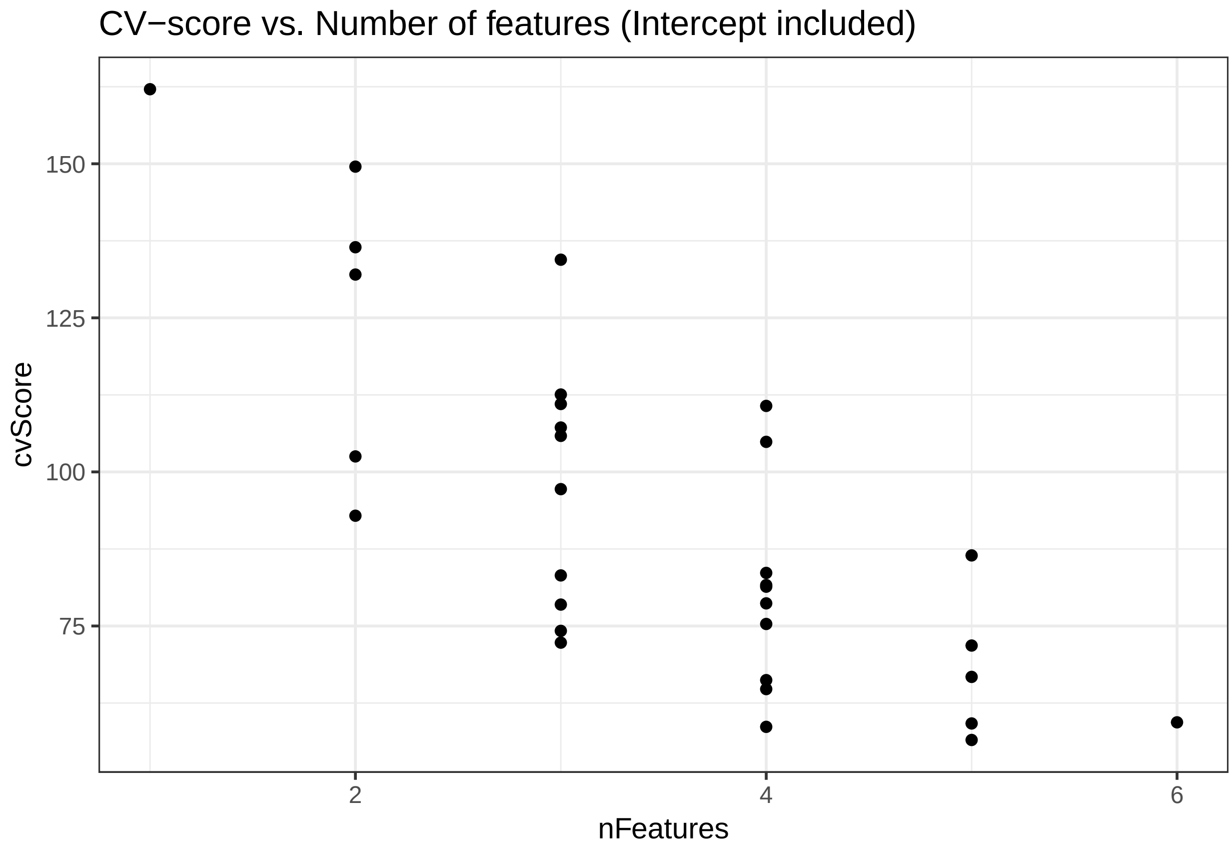
```
    obs = obs[!obs %in% folds[[i]]]
  }
  # Creating equal sized folds could lead to unallocated observations
  # which will be allocated one by one iteratively
  i = 1
  while (length(obs) > 0) {
    folds[[i]] = c(folds[[i]], obs[1])
    obs = obs[-1]
    i = i+1
    if (i > Nfolds) {
      i = 1
    }
  }
# Identifying all possible subsets
i = 1
allSubsets = list()
for (nX in 0:(ncol(X)-1)) {
  combinations = combn(1:(ncol(X)-1), nX)
  for (comb in 1:ncol(combinations)) {
    allSubsets[[i]] = c(combinations[,comb], ncol(X)) # to each subset the intercept will be also add
    i = i+1
  }
}
# Selecting best subset for linear regression
# Setting up list object with CV-score (default = 0) for each subset
subsetCVScore = list()
for (i in 1:length(allSubsets)) {
  subsetCVScore[[i]] = 0
}
i = 1
# Calculating CV-score for each subset applying k-cross-validation
for (subset in allSubsets) {
  subsetX = as.matrix(X[,subset])
  for (validationSet in folds) {
    # Creating train and validation set
    trainX = subsetX[-validationSet,]
    trainY = Y[-validationSet]
    validationX = subsetX[validationSet,]
    validationY = Y[validationSet]
    # Getting beta estimates with train data
    betaEstimates = (solve(t(trainX) %*% trainX)) %*% t(trainX) %*% trainY
    # Estimating validationY
    validationYFit = validationX %*% betaEstimates
    # Calculating MSE
    mse = mean((validationY - validationYFit)^2)
    # Adding MSE to subsetCVScore
    subsetCVScore[[i]] = subsetCVScore[[i]] + mse
  }
  # calculating CV-score for each subset
  subsetCVScore[[i]] = subsetCVScore[[i]]/Nfolds
  i = i+1
}
# Identifying subset with least CV-score
```

```r
  leastCVScore = 99999999
  for (i in 1:length(subsetCVScore)) {
    if (subsetCVScore[[i]] < leastCVScore) {
      leastCVScore = subsetCVScore[[i]]
      bestSubset = i
    }
  }
  # Calculating data frame with CV-score per number of features for plot
  nFeaturesCVScore = setNames(data.frame(matrix(ncol = 2, nrow = 0)), c("nFeatures", "cvScore"))
  i = 1
  for (subset in allSubsets) {
    nFeaturesCVScore[i,1] = length(allSubsets[[i]])
    nFeaturesCVScore[i,2] = subsetCVScore[[i]]
    i = i+1
  }
  # Returning elements
  print("Best subset:")
  print(sort(colnames(X[,allSubsets[[bestSubset]]])))
  print(paste("CV score: ", round(subsetCVScore[[bestSubset]], 2), sep = ""))
  ggplot2::ggplot(nFeaturesCVScore, ggplot2::aes(nFeatures, cvScore)) +
    ggplot2::geom_point() +
    ggplot2::theme_bw() +
    ggplot2::ggtitle("CV-score vs. Number of features (Intercept included)")
}
```

3.2

```r
data(swiss)
featureSelection(X = as.matrix(swiss[,-1]),
                 Y = swiss[,1],
                 Nfolds = 5)
```

```
## [1] "Best subset:"
## [1] "Agriculture"      "Catholic"         "Education"
## [4] "Infant.Mortality" "Intercept"
## [1] "CV score: 56.51"
```



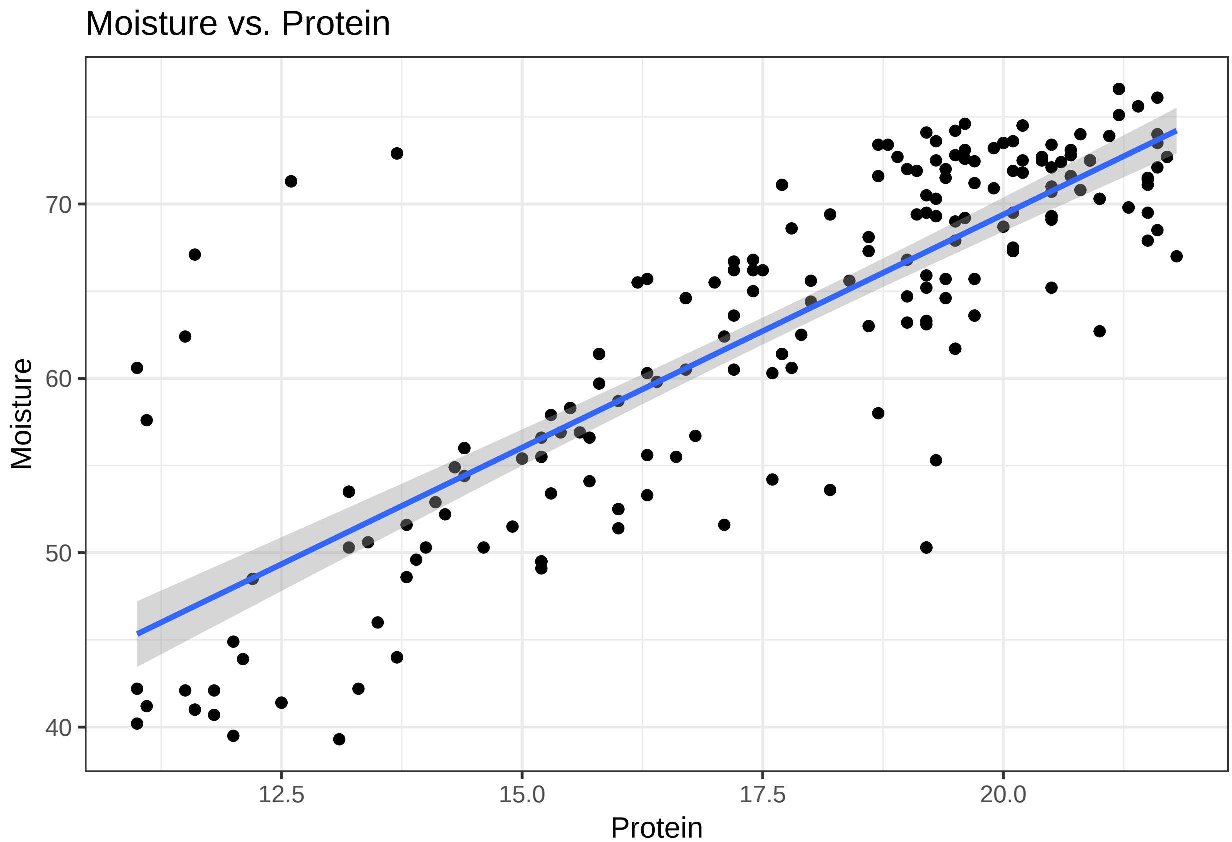
It can be seen that with an increasing number of features, the CV score approximately deacreases linearly. While the CV score is very high using only the *intercept* as a predictor (nFeatures = 1), the minimum CV score of 56.51 can be found for nFeatures = 5 (which are the variables *Agriculture*, *Catholic*, *Education*, *Infant.Mortality* and the *intercept*).

Linear regression identified a linear relationship between the independent variables and the dependent variable *Fertility*. To say that it is reasonable that the independent variables have largest impact on the target, causation needs to checked. It indicates that one event is the result of the occurrence of the other event.

Assignment 4

4.1

```r
# Importing data
library(readxl)
data = read_excel("tecator.xlsx")

# Plotting moisture versus protein
library(ggplot2)
ggplot(data = data, aes(x = Protein, y = Moisture)) +
  geom_point() +
  geom_smooth(method = "lm") +
  theme_bw() +
  labs(title = "Moisture vs. Protein")
```



Since the blue line indicates a linear model which assumes that there is a positive relationship between Moisture and Protein, the data can be presented by a linear model. However, the model would not predict the outlier values which can be observed in the top left corner of the plot.

4.2

The probabilistic model for M_i can be described as

$$M_i = \beta_0 + \beta_1 Protein + \beta_2 Protein^2 + ... + \beta_i Protein^i + \epsilon.$$

Since we know that Moisture is normally distributed and

$$\epsilon \sim N\left(0, \sigma^2\right),$$

$$M_i \sim N\left(\sum_{j=0}^{i} \beta_j Protein^j, \sigma^2\right).$$

Since

$$\epsilon = \sum_{j=0}^{i} (y_j - \hat{y_j})$$

and

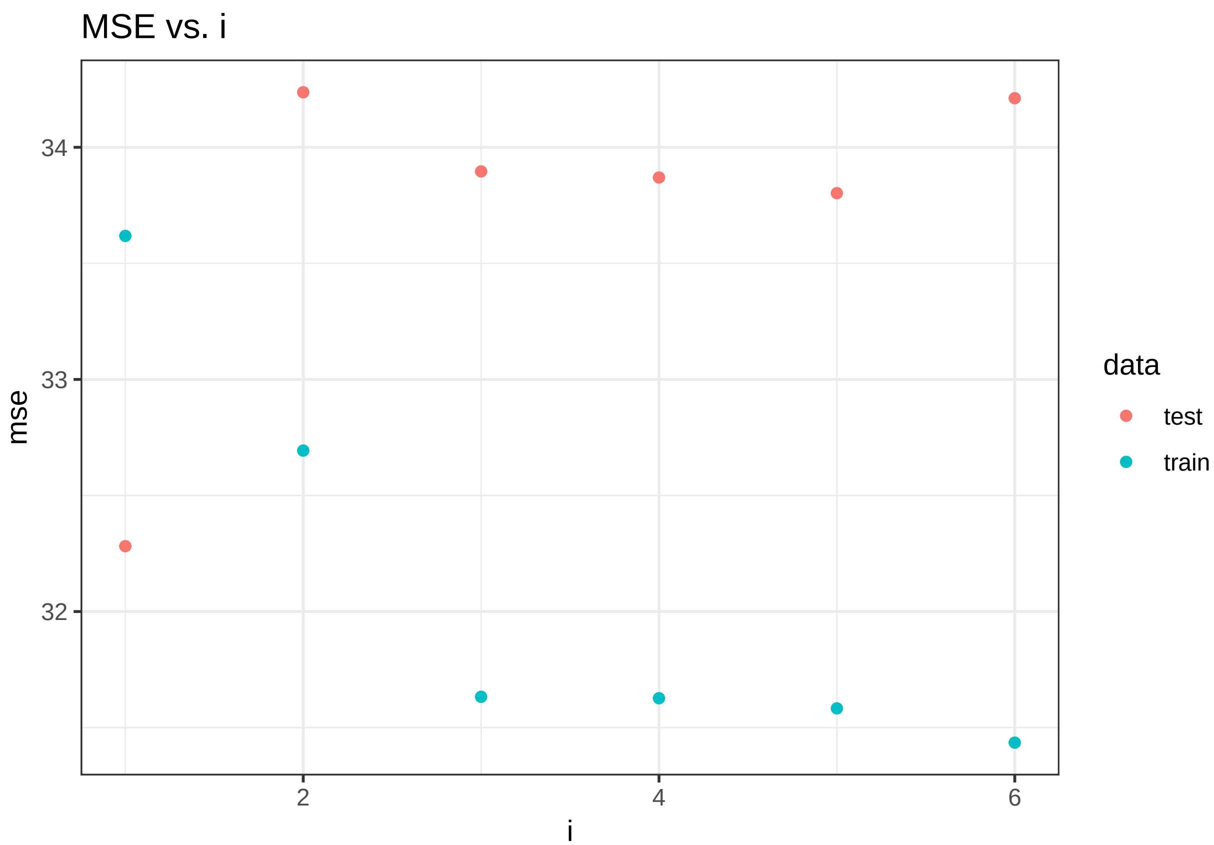
$$MSE = \frac{1}{n} \sum_{j=0}^{i} (y_j - \hat{y_j})^2,$$

it becomes clear that the calculation of the MSE is very similar to the caluclation of the error term. That shows that the MSE is a good indicator for how well the model fits the data, because it quantifies the difference between the predicted and actual values.

4.3

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

# Fitting models
m1 = lm(Moisture ~ Protein, data = train)
m2 = lm(Moisture ~ Protein + I(Protein^2), data = train)
m3 = lm(Moisture ~ Protein + I(Protein^2) + I(Protein^3), data = train)
m4 = lm(Moisture ~ Protein + I(Protein^2) + I(Protein^3) + I(Protein^4), data = train)
m5 = lm(Moisture ~ Protein + I(Protein^2) + I(Protein^3) + I(Protein^4) + I(Protein^5), data = train)
m6 = lm(Moisture ~ Protein + I(Protein^2) + I(Protein^3) + I(Protein^4) + I(Protein^5) + I(Protein^6), d

# Predicting moisture values using the fitted models
  # Train data
m1yFitTrain = predict(m1, train)
m2yFitTrain = predict(m2, train)
m3yFitTrain = predict(m3, train)
m4yFitTrain = predict(m4, train)
m5yFitTrain = predict(m5, train)
m6yFitTrain = predict(m6, train)
  # Test data
m1yFitTest = predict(m1, test)
m2yFitTest = predict(m2, test)
m3yFitTest = predict(m3, test)
m4yFitTest = predict(m4, test)
m5yFitTest = predict(m5, test)
m6yFitTest = predict(m6, test)

# Calculating MSE
  # Train data
m1MSETrain = mean((train$Moisture - m1yFitTrain)^2)
m2MSETrain = mean((train$Moisture - m2yFitTrain)^2)
m3MSETrain = mean((train$Moisture - m3yFitTrain)^2)
m4MSETrain = mean((train$Moisture - m4yFitTrain)^2)
m5MSETrain = mean((train$Moisture - m5yFitTrain)^2)
m6MSETrain = mean((train$Moisture - m6yFitTrain)^2)
  # Test data
m1MSETest = mean((test$Moisture - m1yFitTest)^2)
m2MSETest = mean((test$Moisture - m2yFitTest)^2)
m3MSETest = mean((test$Moisture - m3yFitTest)^2)
m4MSETest = mean((test$Moisture - m4yFitTest)^2)
m5MSETest = mean((test$Moisture - m5yFitTest)^2)
m6MSETest = mean((test$Moisture - m6yFitTest)^2)

# Combining restults in a data frame
finalDf = setNames(data.frame(matrix(ncol = 3, nrow = 12)), c("i", "data", "mse"))
finalDf$i = c(1:6,1:6)
finalDf$data = c(rep("train",6),rep("test",6))
finalDf$mse = c(m1MSETrain, m2MSETrain, m3MSETrain, m4MSETrain, m5MSETrain, m6MSETrain,
```

```
                m1MSETest, m2MSETest, m3MSETest, m4MSETest, m5MSETest, m6MSETest)

# Plotting results
ggplot(data = finalDf, aes(x = i, y = mse, color = data)) +
  geom_point() +
  theme_bw() +
  labs(title = "MSE vs. i")
```



Which model is best according to the plot? How do the MSE values change and why? Interpret this picture in terms of bias-variance tradeoff.

Since the least difference between the test and train MSE can be observed for i=1, model M_1 is the best according to the plot. The test MSE is even lower than the training MSE, which even indicates little underfitting. A big challenge in fitting a model is to avoid overfitting. As model flexibility increases, training MSE will decrease, but the test MSE may increase as shown in the plot as well. This happens if the statistical model identifies patterns in the training data which are just caused by random chance.

4.4

```
library(MASS)
model = lm(Fat ~ ., data = data[,2:102])
stepwiseSelection = stepAIC(model, direction = "both", trace = FALSE)
summary(stepwiseSelection)
```

```
##
## Call:
## lm(formula = Fat ~ Channel1 + Channel2 + Channel4 + Channel5 +
```

```
##     Channel7 + Channel8 + Channel11 + Channel12 + Channel13 +
##     Channel14 + Channel15 + Channel17 + Channel19 + Channel20 +
##     Channel22 + Channel24 + Channel25 + Channel26 + Channel28 +
##     Channel29 + Channel30 + Channel32 + Channel34 + Channel36 +
##     Channel37 + Channel39 + Channel40 + Channel41 + Channel42 +
##     Channel45 + Channel46 + Channel47 + Channel48 + Channel50 +
##     Channel51 + Channel52 + Channel54 + Channel55 + Channel56 +
##     Channel59 + Channel60 + Channel61 + Channel63 + Channel64 +
##     Channel65 + Channel67 + Channel68 + Channel69 + Channel71 +
##     Channel73 + Channel74 + Channel78 + Channel79 + Channel80 +
##     Channel81 + Channel84 + Channel85 + Channel87 + Channel88 +
##     Channel92 + Channel94 + Channel98 + Channel99, data = data[,
##     2:102])
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.82961 -0.57129 -0.00696  0.58152  2.86375
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)      7.093      1.453   4.882 2.64e-06 ***
## Channel1     10559.894   2333.430   4.525 1.21e-05 ***
## Channel2    -12636.967   3467.995  -3.644 0.000369 ***
## Channel4      8489.323   4637.993   1.830 0.069164 .
## Channel5    -10408.967   4771.350  -2.182 0.030689 *
## Channel7     -5376.018   3851.782  -1.396 0.164847
## Channel8      7215.595   4246.489   1.699 0.091342 .
## Channel11    -9505.520   5721.115  -1.661 0.098692 .
## Channel12    37240.918  12290.648   3.030 0.002878 **
## Channel13   -41564.547  15892.375  -2.615 0.009817 **
## Channel14    34938.179  13290.454   2.629 0.009454 **
## Channel15   -23761.451   6584.006  -3.609 0.000417 ***
## Channel17     4296.572   3189.730   1.347 0.179998
## Channel19    14279.808   5017.407   2.846 0.005042 **
## Channel20   -23855.616   5153.161  -4.629 7.85e-06 ***
## Channel22    18444.906   3381.683   5.454 1.97e-07 ***
## Channel24   -20138.426   4946.417  -4.071 7.52e-05 ***
## Channel25    18137.432   5374.094   3.375 0.000938 ***
## Channel26    -7670.318   3859.006  -1.988 0.048660 *
## Channel28    20079.898   4991.631   4.023 9.06e-05 ***
## Channel29   -36351.014   7655.223  -4.749 4.72e-06 ***
## Channel30    18071.276   5863.802   3.082 0.002446 **
## Channel32     3838.013   2722.862   1.410 0.160729
## Channel34    -9242.884   2225.926  -4.152 5.48e-05 ***
## Channel36     8070.938   3317.588   2.433 0.016152 *
## Channel37    -9045.588   3536.621  -2.558 0.011522 *
## Channel39    18664.454   5986.730   3.118 0.002183 **
## Channel40   -20069.709  10701.902  -1.875 0.062677 .
## Channel41    22257.776  11122.533   2.001 0.047169 *
## Channel42   -21760.853   5833.811  -3.730 0.000270 ***
## Channel45    18145.804   2985.416   6.078 9.50e-09 ***
## Channel46    -8225.696   3715.367  -2.214 0.028330 *
## Channel47    -4986.549   2558.694  -1.949 0.053165 .
## Channel48     2876.075   2014.985   1.427 0.155546
```

```
## Channel50   -13009.410   4535.797  -2.868 0.004720 **
## Channel51    29251.161   6554.297   4.463 1.57e-05 ***
## Channel52   -26833.976   4389.473  -6.113 7.97e-09 ***
## Channel54    30954.862   4392.339   7.047 6.06e-11 ***
## Channel55   -35183.287   5646.314  -6.231 4.39e-09 ***
## Channel56    14912.986   2810.889   5.305 3.93e-07 ***
## Channel59    -8030.278   1887.431  -4.255 3.66e-05 ***
## Channel60    13071.416   2629.374   4.971 1.79e-06 ***
## Channel61    -7850.189   2246.864  -3.494 0.000625 ***
## Channel63    15059.275   3231.692   4.660 6.90e-06 ***
## Channel64   -19909.466   4727.696  -4.211 4.35e-05 ***
## Channel65     4190.184   3486.766   1.202 0.231346
## Channel67    13850.508   3909.121   3.543 0.000526 ***
## Channel68   -25873.365   5304.223  -4.878 2.69e-06 ***
## Channel69    18362.385   3331.483   5.512 1.50e-07 ***
## Channel71    -9223.910   1558.752  -5.917 2.11e-08 ***
## Channel73    12456.498   2386.255   5.220 5.82e-07 ***
## Channel74    -5624.411   1933.590  -2.909 0.004177 **
## Channel78    -7927.105   2176.860  -3.642 0.000372 ***
## Channel79    15473.188   3812.200   4.059 7.89e-05 ***
## Channel80   -22391.895   4490.714  -4.986 1.67e-06 ***
## Channel81    13852.453   3105.934   4.460 1.59e-05 ***
## Channel84   -11442.630   3457.064  -3.310 0.001167 **
## Channel85    20228.671   4081.863   4.956 1.91e-06 ***
## Channel87   -15938.315   4102.273  -3.885 0.000153 ***
## Channel88     5647.072   3236.286   1.745 0.083033 .
## Channel92     6595.995   1864.595   3.537 0.000537 ***
## Channel94    -5497.846   1847.113  -2.976 0.003397 **
## Channel98    -8728.596   2489.314  -3.506 0.000598 ***
## Channel99     8554.587   1898.010   4.507 1.31e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.107 on 151 degrees of freedom
## Multiple R-squared:  0.9947, Adjusted R-squared:  0.9925
## F-statistic: 447.9 on 63 and 151 DF,  p-value: < 2.2e-16
```

```r
nVariables = length(stepwiseSelection$coefficients)-1
paste0("selected variables: ", nVariables)
```

```
## [1] "selected variables: 63"
```

4.5

```r
library(glmnet)
```

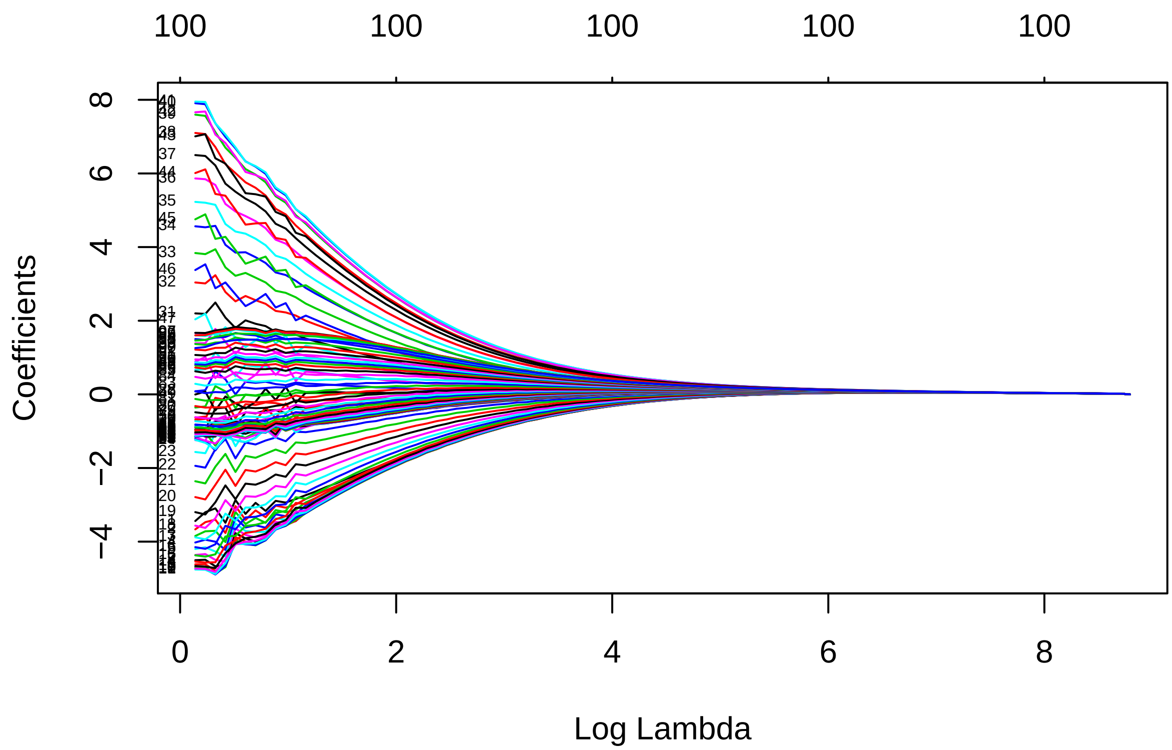
```
## Loading required package: Matrix
```

```
## Loading required package: foreach
```

```
## Loaded glmnet 2.0-16
```

```r
covariates = data[,2:101]
response = data[,102]
modelRidge = glmnet(as.matrix(covariates),
                    as.matrix(response),
```

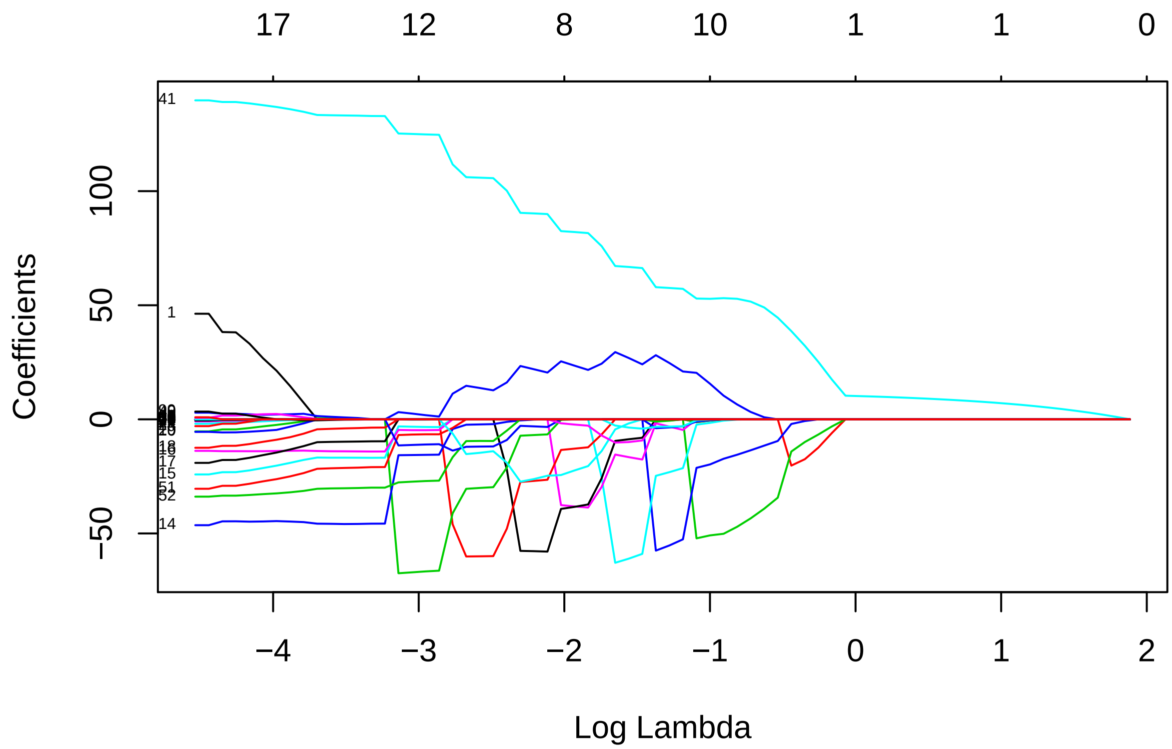
```
                     alpha=0,
                     family="gaussian")
plot(modelRidge, xvar="lambda", label=TRUE)
```



All coefficients converge to zero as Log Lambda increases.

4.6

```
modelLasso = glmnet(as.matrix(covariates),
                     as.matrix(response),
                     alpha=1,
                     family="gaussian")
plot(modelLasso, xvar = "lambda", label = TRUE)
```



Compared to the Ridge regression, using Lasso regression, the coefficients converge much faster towards zero as Log Lambda increases.

4.7

```
modelLassoCV = cv.glmnet(as.matrix(covariates),
                         as.matrix(response),
                         alpha = 1,
                         family = "gaussian",
                         lambda = seq(0,1,0.001))
coef(modelLassoCV, s = "lambda.min")
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
##                           1
## (Intercept)   1.218582e+01
## Channel1      2.243351e+01
## Channel2     -5.477212e+01
## Channel3      2.028288e+01
## Channel4      1.790966e+01
## Channel5      1.599792e+01
## Channel6      1.496901e+01
## Channel7      4.930552e+01
## Channel8      1.410886e+01
## Channel9      1.100549e+01
## Channel10    -3.946699e+01
## Channel11    -3.413441e+01
## Channel12    -3.361613e+00
```

```
## Channel13    1.501427e+02
## Channel14    2.372668e-01
## Channel15   -4.867299e+01
## Channel16   -4.549011e+01
## Channel17   -3.891421e+01
## Channel18   -3.095813e+01
## Channel19   -2.488455e+01
## Channel20   -2.152392e+01
## Channel21   -1.721212e+01
## Channel22   -1.288916e+01
## Channel23   -7.467292e+00
## Channel24    3.858146e-01
## Channel25    6.141436e+00
## Channel26    7.339759e+00
## Channel27    4.952231e+00
## Channel28    9.256704e-01
## Channel29   -1.845498e+00
## Channel30   -6.201764e+00
## Channel31   -1.240428e+01
## Channel32   -1.608200e+01
## Channel33   -1.533188e+01
## Channel34   -8.878726e+00
## Channel35   -1.222622e+00
## Channel36    4.986875e+00
## Channel37    8.663342e+00
## Channel38    9.507388e+00
## Channel39    5.835387e+00
## Channel40    5.486958e+00
## Channel41    1.437055e+02
## Channel42    2.782707e+01
## Channel43    9.118839e+00
## Channel44   -9.479487e+00
## Channel45   -2.436273e+00
## Channel46   -7.205962e+00
## Channel47   -7.485645e-04
## Channel48   -1.494150e+01
## Channel49   -1.826387e+01
## Channel50   -2.915222e+01
## Channel51   -6.977177e+01
## Channel52   -1.219494e+01
## Channel53    7.071090e+00
## Channel54   -9.397099e+00
## Channel55   -3.634639e-01
## Channel56    4.870685e+00
## Channel57    3.943612e+01
## Channel58    1.141937e+01
## Channel59    4.812258e+00
## Channel60    5.489478e+00
## Channel61    1.251985e+01
## Channel62    2.021891e+01
## Channel63   -7.405961e+00
## Channel64   -2.464353e+01
## Channel65    3.321800e+01
## Channel66   -9.834681e+00
```

```
## Channel67    1.262018e+01
## Channel68   -3.217675e+00
## Channel69    8.903740e+00
## Channel70    9.503287e-01
## Channel71    3.949972e+00
## Channel72    1.789541e+01
## Channel73   -2.043606e+01
## Channel74   -1.276925e+01
## Channel75   -8.831294e+00
## Channel76   -2.240484e+01
## Channel77    8.138153e-01
## Channel78   -2.734175e+01
## Channel79   -2.300507e+01
## Channel80    2.379017e-04
## Channel81    1.391527e-04
## Channel82    9.866988e-05
## Channel83    1.303538e-06
## Channel84   -8.303980e-05
## Channel85   -4.297569e-05
## Channel86   -3.530336e-05
## Channel87    5.470530e-05
## Channel88    8.946494e-05
## Channel89    9.648548e-05
## Channel90   -2.640082e+01
## Channel91    6.485393e-01
## Channel92    7.039651e-01
## Channel93   -1.993822e+01
## Channel94    2.455602e+01
## Channel95    2.755495e+01
## Channel96    4.056484e-02
## Channel97    8.631612e+00
## Channel98    4.537254e+00
## Channel99    5.168688e+00
## Channel100   1.640315e+01
```

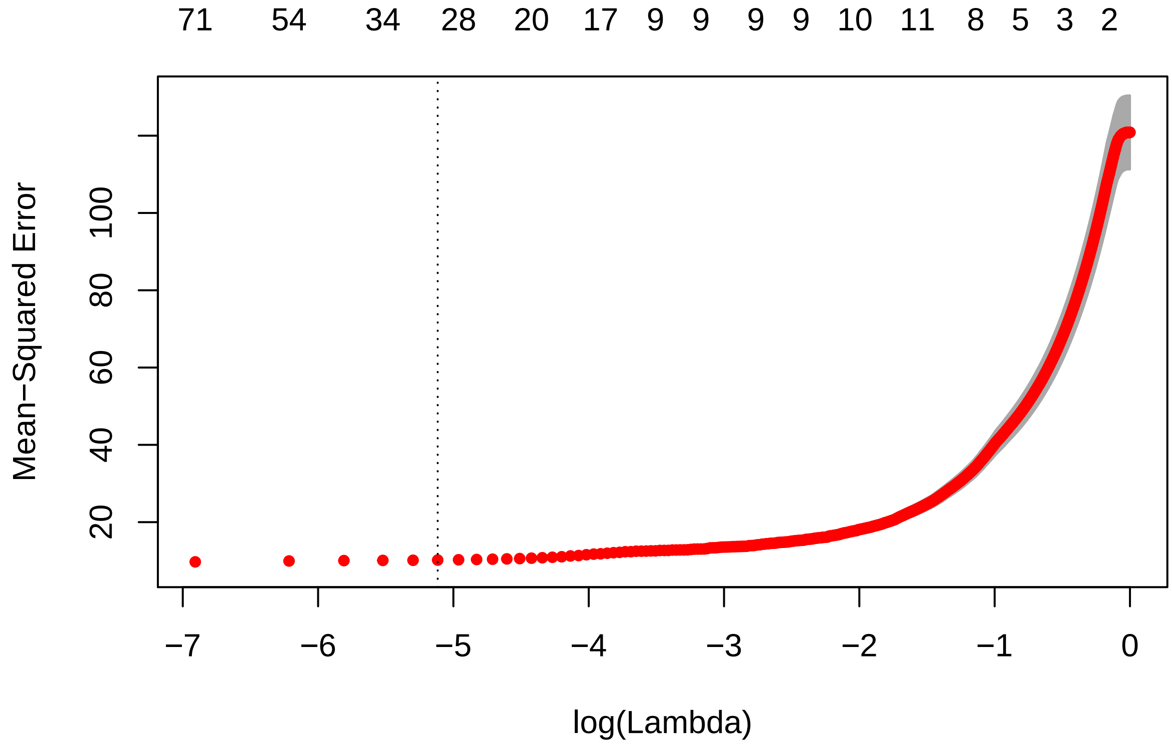
```r
paste0("Selected variables: ", length(coef(modelLassoCV, s = "lambda.min"))-1) # all variables used
```

```
## [1] "Selected variables: 100"
```

```r
paste0("Optimal Lambda: ", modelLassoCV$lambda.min)
```

```
## [1] "Optimal Lambda: 0"
```

```r
plot(modelLassoCV)
```



4.8

Comparing the two different methods, the number of variables selected differs strongly. While as a result of 4.4, 63 variables were selected, in 4.7 all 100 independent variables were selected to predict the dependent variable.