Lab Report in Machine Learning

Laboration 1

TDDE01

David Grönberg
Davgr686

22-11-2019

Contents

List of Figures

List of Tables

1. Assignments

1.1 Assignment 1

The data file spambase.xlsx contains information about 2740 emails and their word frequency, consisting of 48 columns representing words and one column representing if the document was classified as spam (1) or not (0). The data is first divided into a training set and a test set. The Spam column is changed to a categorical value.

```
data <- read_excel("spambase.xlsx")
# make spam categorical
data$Spam <- as.factor(data$Spam)

# Split 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,]
```

The *glm()* function is used to fit a logistic regression model on the train set. Here, the family is set to "binomial" to indicate that a logistic regression model is to be fit.

```
logRegModel <- glm(Spam ~ ., data = train, family = binomial)
```

After the model is fitted, the *predict()* function is used to predict the target (Spam column) in the train and test set. The classification principle

$$\hat{Y} = 1 \; if \; p(Y = 1|X) > 0.5, \; otherwise \; \hat{Y} = 0$$

is used to classify Spam as a "1" if the probability is more than 0.5 and to classify Spam as a "0" if the probability is equal or less than 0.5. The prediction is done on both the train and test set. The confusion matrices and the miss-classification rates are presented.

```
train_probs <- predict(logRegModel, train, type = "response")
train_pred <- ifelse(train_probs > 0.5, "1", "0")

test_probs <- predict(logRegModel, test, type = "response")
test_pred <- ifelse(test_probs > 0.5, "1", "0")
```

Table 1.1: Confusion Matrix for train set.

	0	1
0	803	142
1	81	344

Table 1.2: Confusion Matrix for test set.

	0	1
0	791	146
1	97	336

Table 1.3: Misclassification rates for train and test set

Train	Test
0.1627737	0.1773723

The confusion matrices summarize the prediction results and shows the number of correct and incorrect predictions for each class. It gives insight into the errors being made by the classifier and the types of errors that are being made. The misclassification rate is the percentage of misclassified data from a given data set.

The logistic regression model has a slighly higher accuracy on the train set, compared to the test set. The model's misclassification rate is higher when predicting the data as spam ("1").

The classification principle is altered to:

$$\hat{Y} = 1 \; if \; p(Y = 1|X) > 0.8, \; otherwise \; \hat{Y} = 0$$

which makes the model predict "0" to a greater extent. The confusion matrix show that the model is more accurate at classifying not spam, but worse at classifying true spam. Because the model is now significantly worse at classifying true spam, the missclassification rates grow greater for both the train and test set.

Table 1.4: Confusion Matrix for train set with cutoff 0.8.

	0	1
0	893	52
1	400	25

Table 1.5: Confusion Matrix for test set with cutoff 0.8.

	0	1
0	926	11
1	367	66

Table 1.6: Misclassification rates for train and test set with cutoff 0.8

Train	Test
0.329927	0.2759124

A new model is fitted on the same train and test set using a weighted nearest neighbor classifier. Weighted nearest neighbor classifiers gives the nearest k neighbors a weight using a kernel function. It gives more weight to the neighbors which are nearby and less weight to the neighbors which are farther away.

```
kknn_classifier = kknn(Spam ~ ., train, train, k=30)
fit <- fitted(kknn_classifier)

kknn_classifier = kknn(Spam ~ ., train, test, k=30)
fit <- fitted(kknn_classifier)
```

Table 1.7: Confusion Matrix for train set using kknn (k=30).

	0	1
0	779	152
1	77	362

Table 1.8: Confusion Matrix for test set using kknn (k=30).

	0	1
0	702	249
1	180	239

Table 1.9: Misclassification rates for train and test set using kknn (k=30)

Train	Test
0.1671533	0.3131387

The weighted nearest neighbor classifier performs well on the train set, having a similar misclassification rate as the logistic regression model with 0.5 as the cutoff value. However, the misclassification rate almost doubles for the test set.

The k variable in the kknn model is altered from 30 to 1. Meaning that the model only considers one neighbor when making predictions. The confusion matrices and misclassification rates for the train and test set are shown below.

Table 1.10: Confusion Matrix for train set using kknn (k=1).

	0	1
0	931	0
1	0	439

Table 1.11: Confusion Matrix for test set using kknn (k=1).

	0	1
0	644	307
1	185	234

Table 1.12: Misclassification rates for train and test set using kknn (k=1)

Train	Test
0	0.3591241

The confusion matrix for the train set shows that the model is 100% correct in all cases, leading to a misclassification rate of 0. Because the model only considers one nearest neighbor, and is both fit and validated on the train set, it will predict the right answer every time. However, when the model is fit on the train set and validated on the test set, the model performs slightly worse, when compared to the previous model using its 30 nearest neighbors.

1.2 Assignment 2

The Machines data set consists of 48 rows containing one column (Length) that describe the life time of a machine. The following probability model is assumed for the Length variable:

$$p(x|\theta) = \theta e^{-\theta x}$$

which has an exponential distribution and is verified when looking at the density distribution graph.

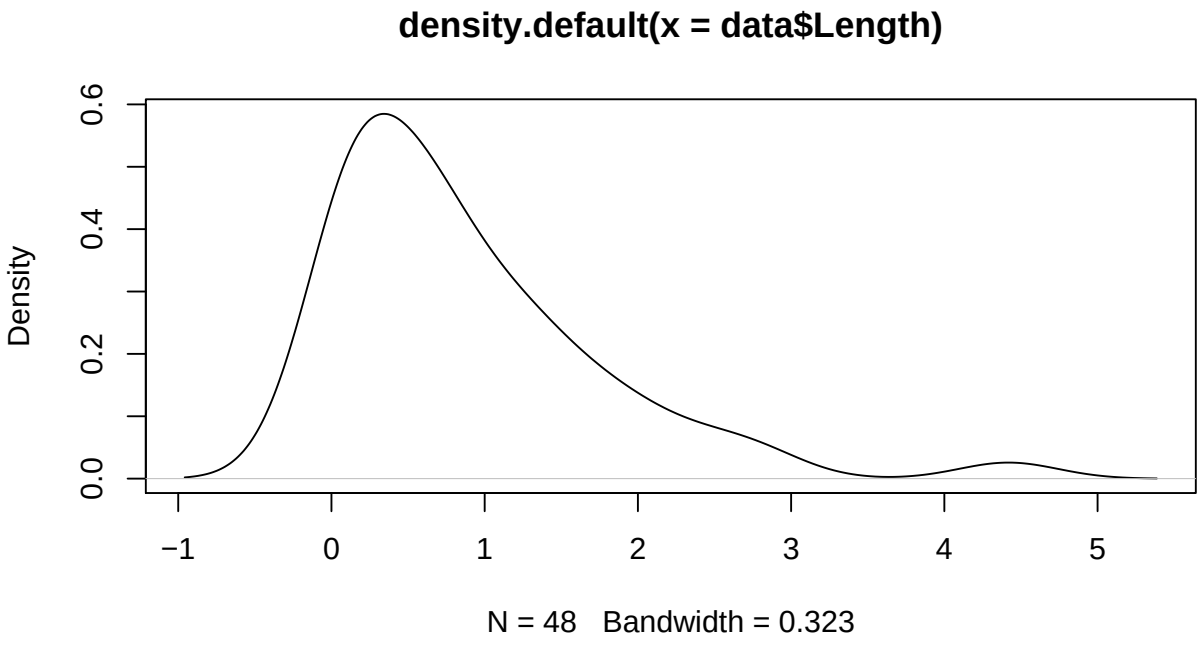

Figure 1.1: Density Distribution

A function that computes the log-likelihood for a given θ and a given vector x is created.

```
loglike <- function(theta, vector) {
  return (length(vector)*log(theta) - theta*sum(vector))
}
```

and the dependence of log-likelihood on θ is graphed. The graph 1.2 show how the log-likelihood varies when θ varies between 0 and 5. The log-likelihood is at its maximum when θ is around 1.

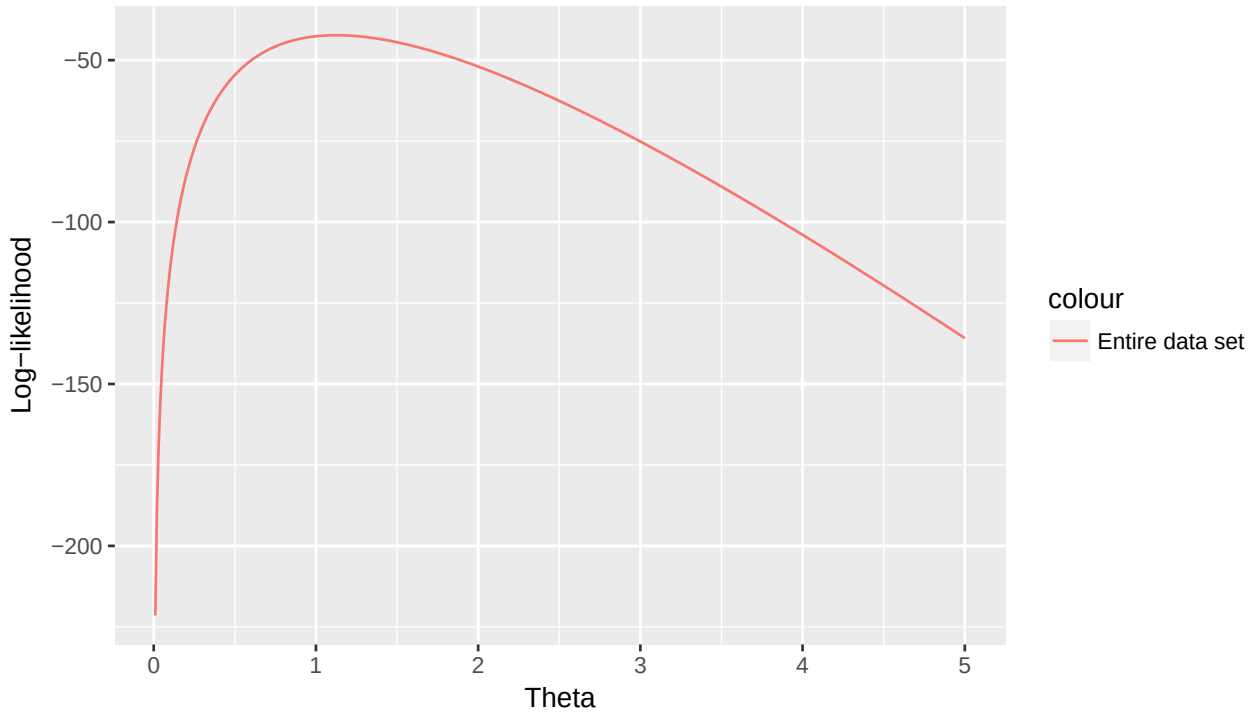

Figure 1.2: Log-likelihood dependency on Theta

Figure 1.3 shows a comparison of the log-likelihood dependency on θ when using the entire data set and when only using the six first observations. The graph shows that the maximum likelihood does not vary as much for the different θ values, compared to when using the entire data set. The maximum likelihood seems to be when θ is between 1 and 2, similarly to the maximum likelihood for the entire data set.

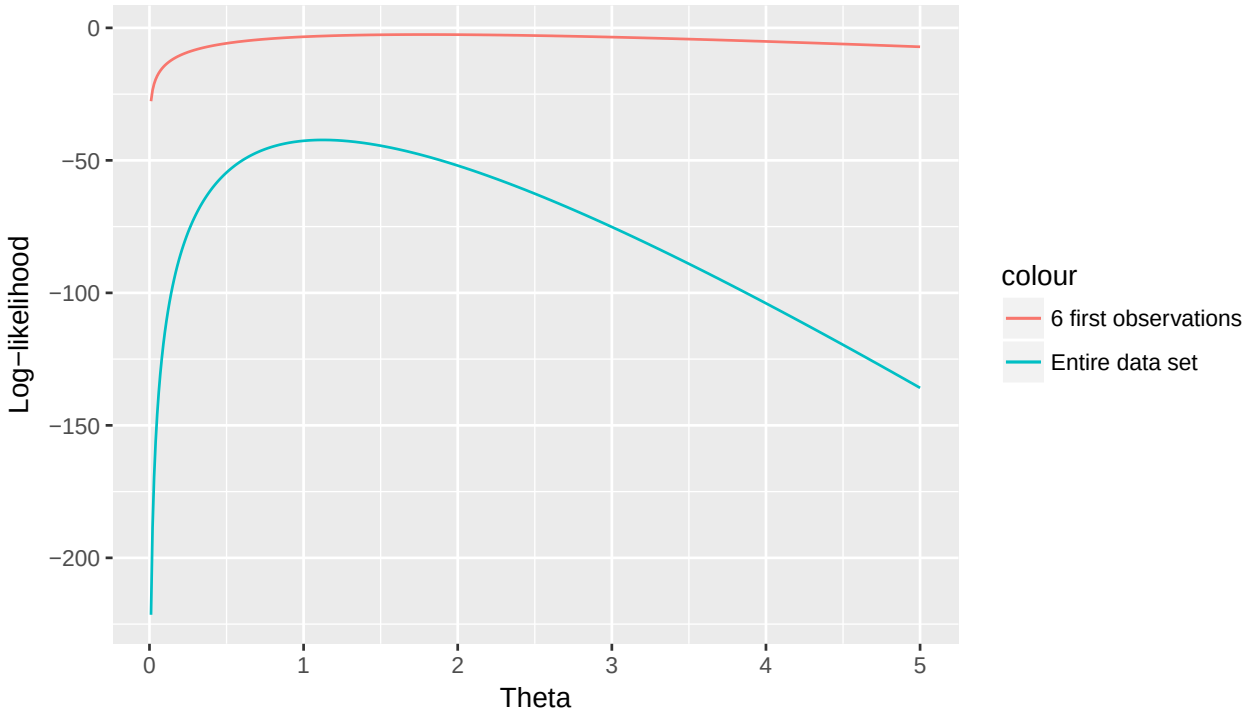

Figure 1.3: Log-likelihood dependency on Theta for both data sets

A Bayesian model with probability: $p(x|\theta) = \theta e^{-\theta x}$ and a prior: $p(\theta) = e^{-x}, = 10$ is assumed. A function computing $l(\theta) = log(p(x|\theta)p(\theta))$ is written that computes the log of posterior for a given θ.

```
l_theta <- function(theta, vector) {
  lambda <- 10
  prior <- lambda*exp(-lambda*theta)
  return (log(prior) + loglike(theta, vector))
}
```

Figure 1.4 shows the log-likelihood dependency on θ for all three models. The graph shows that the log-likelihood curve for the Bayesian model is very similar to the log-likelihood curve for the model using the whole data set. However, the Bayesian model's curve seem to find its maximum log-likelihood for lower values of θ. This is the result of incorporating a prior in our Bayesian model, which means that the likelihood is weighted by the prior, thus giving us different results compared to the first model.

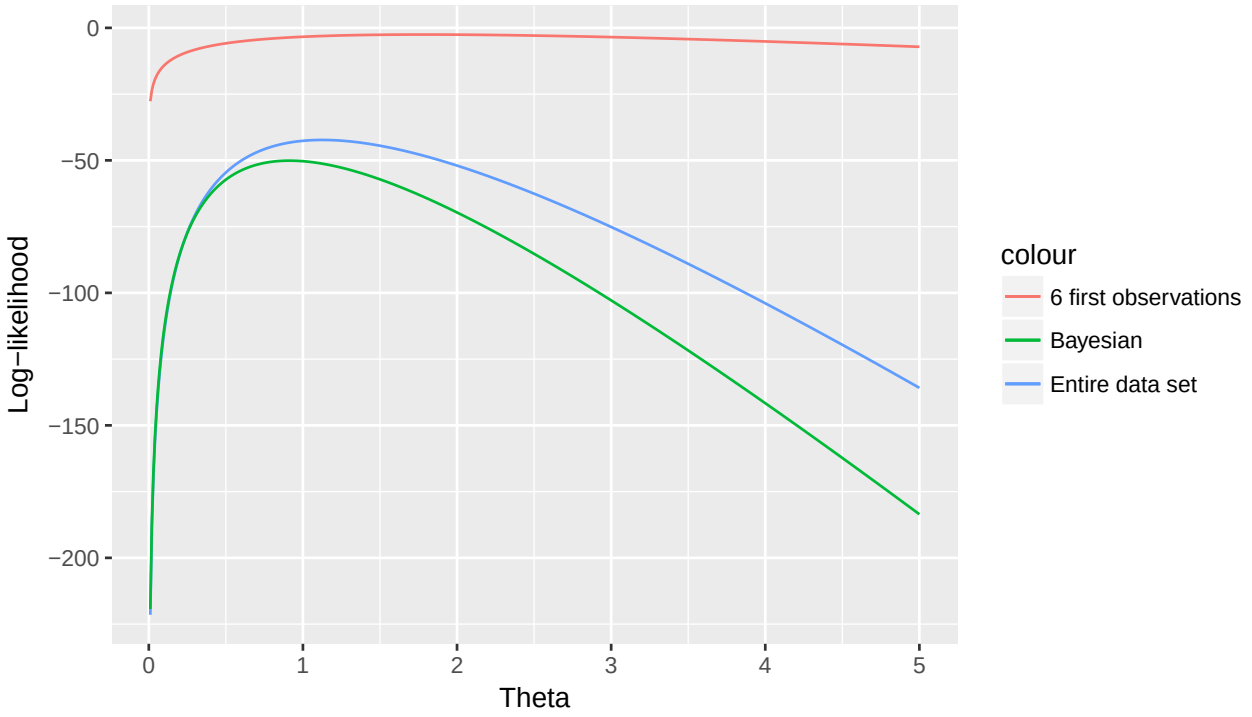

Figure 1.4: Log-likelihood dependency on Theta for all three models

In order to generate new values from the exponential probability distribution $p(x|\theta) = \theta e^{-\theta x}$, the *rexp(n, rate)* function is used, where n represents the number of observations to generate and *rate* represents the rate of occurrence (in this case θ).

```
random_nmbrs <- rexp(n=50, rate = 1.13)
```

Figure 1.5 shows the distribution of the generated values and the original values. The generated values have a very similar distribution compared to the original values, showing that the exponential distribution fits the original data well.

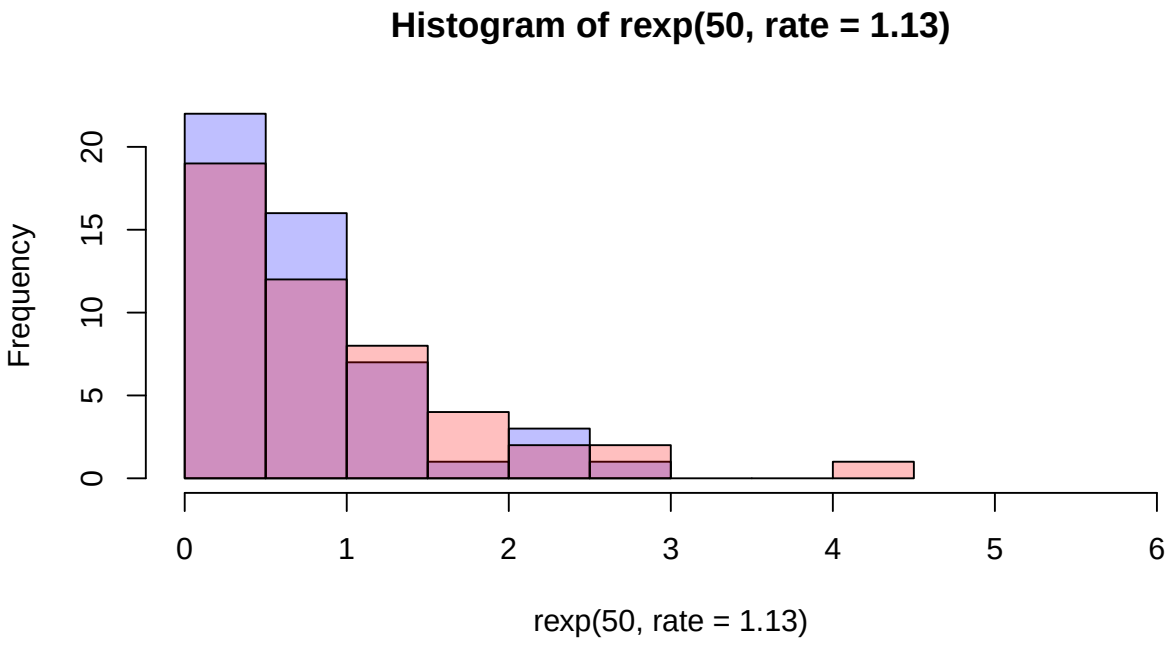

Figure 1.5: Histogram of generated and original values

1.3 Assignment 4

The tecator data set contains data about meat samples and their infrared absorbance spectrum. Each row consists of the absorbance spectrum on the corresponding channels and the levels of moisture, fat and protein. Figure 1.6 graphs a scatter distribution of the Protein and Moisture variable for each row in the tecator data. The graphs shows that Protein seems to generally increase linearly when Moisture does - indicating a linear relationship. Thus, A linear model describes this data well.

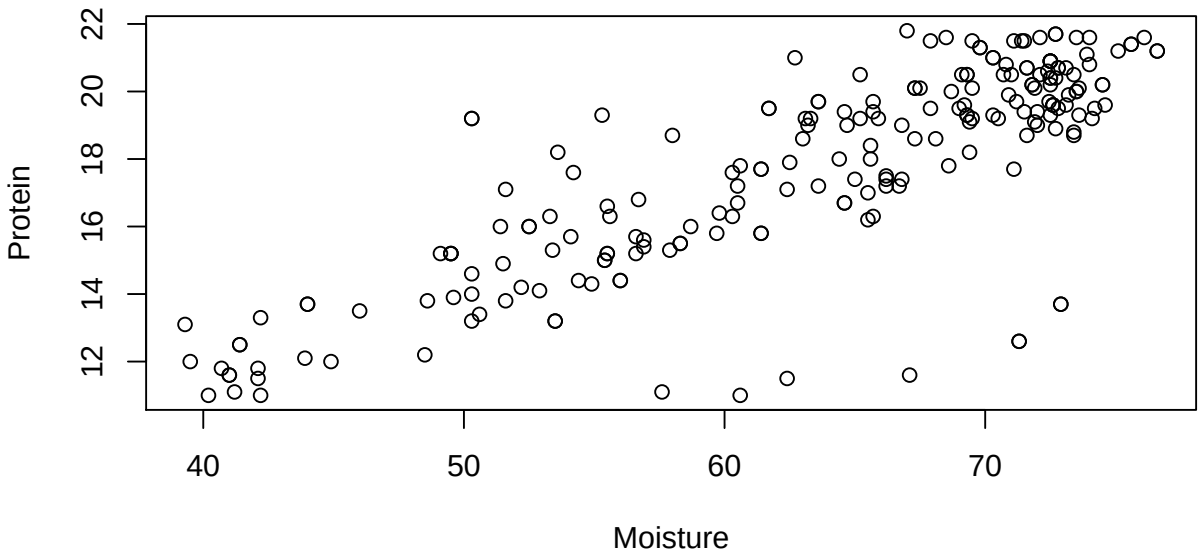

Figure 1.6: Moisture versus Protein

Consider model M_i in which Moisture is normally distributed, and the expected Moisture is a polynomial function of Protein including the polynomial terms up to power i:

$$E(m) = b_0 + b_1 p^1 + b_2 p^2 + ... + b_I p^I + \epsilon \tag{1.1}$$

Where m is the Moisture and p is the Protein.

The data is first divided into a training and test set and then 6 models M_1 to M_6 are fitted using the *lm()* function.

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


# Fit the 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), data = train)
```

Then each model makes predictions on the training and test set and their corresponding Mean Square Error (MSE) is recorded. Mean Square error is a simple and common metric for regression evaluation and defined as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{1.2}$$

where $\hat{y}_i$ is the predicted value and y_i is the actual value. MSE is appropriate to use in regression problems because The MSE tells you how close the regression line is to the set of points in the data. It does this by taking the distances from the points to the regression line and squaring them. It also penalizes larger differences, thus putting more weight on larger errors.

```
# Predict M1
M1.test_preds <- predict(M1, test, type = "response")
M1.train_preds <- predict(M1, train, type = "response")
M1.test_mse <- mean((M1.test_preds - test$Moisture)^2)
M1.train_mse <- mean((M1.train_preds - train$Moisture)^2)

# Predict M2
M2.test_preds <- predict(M2, test, type = "response")
M2.train_preds <- predict(M2, train, type = "response")
M2.test_mse <- mean((M2.test_preds - test$Moisture)^2)
M2.train_mse <- mean((M2.train_preds - train$Moisture)^2)

# Predict M3
M3.test_preds <- predict(M3, test, type = "response")
M3.train_preds <- predict(M3, train, type = "response")
M3.test_mse <- mean((M3.test_preds - test$Moisture)^2)
M3.train_mse <- mean((M3.train_preds - train$Moisture)^2)

# Predict M4
M4.test_preds <- predict(M4, test, type = "response")
M4.train_preds <- predict(M4, train, type = "response")
M4.test_mse <- mean((M4.test_preds - test$Moisture)^2)
M4.train_mse <- mean((M4.train_preds - train$Moisture)^2)

# Predict M5
M5.test_preds <- predict(M5, test, type = "response")
M5.train_preds <- predict(M5, train, type = "response")
M5.test_mse <- mean((M5.test_preds - test$Moisture)^2)
M5.train_mse <- mean((M5.train_preds - train$Moisture)^2)

# Predict M6
M6.test_preds <- predict(M6, test, type = "response")
M6.train_preds <- predict(M6, train, type = "response")
M6.test_mse <- mean((M6.test_preds - test$Moisture)^2)
M6.train_mse <- mean((M6.train_preds - train$Moisture)^2)
```

Figure 1.7 shows the Mean Squared Error for the training and test set for the different values of i in M_i. When evaluating on the training data, the linear model yields the highest MSE. When i is increased, the MSE decreases. The most complex model (i.e. M_6 yields the lowest MSE. However, when we evaluate the models on the test set, the linear model yields the lowest MSE, and the model's MSE increases with the complexity. This can be explained by the concept of overfitting. Overfitting is when a model learns the detail and noise of the training data to the extent that it looses its ability to make adequate predictions on generalized or unseen data, and is said to have a high variance. However, underfitting (i.e. using a too simple model that fails to catch the relevant relationships of the data) is also undesirable. A good model both catches the underlying relationships and performs well on generalized data. However, it is difficult to do both. According to the plot, model M_5 is best because it has the lowest combined MSE of the training and test.

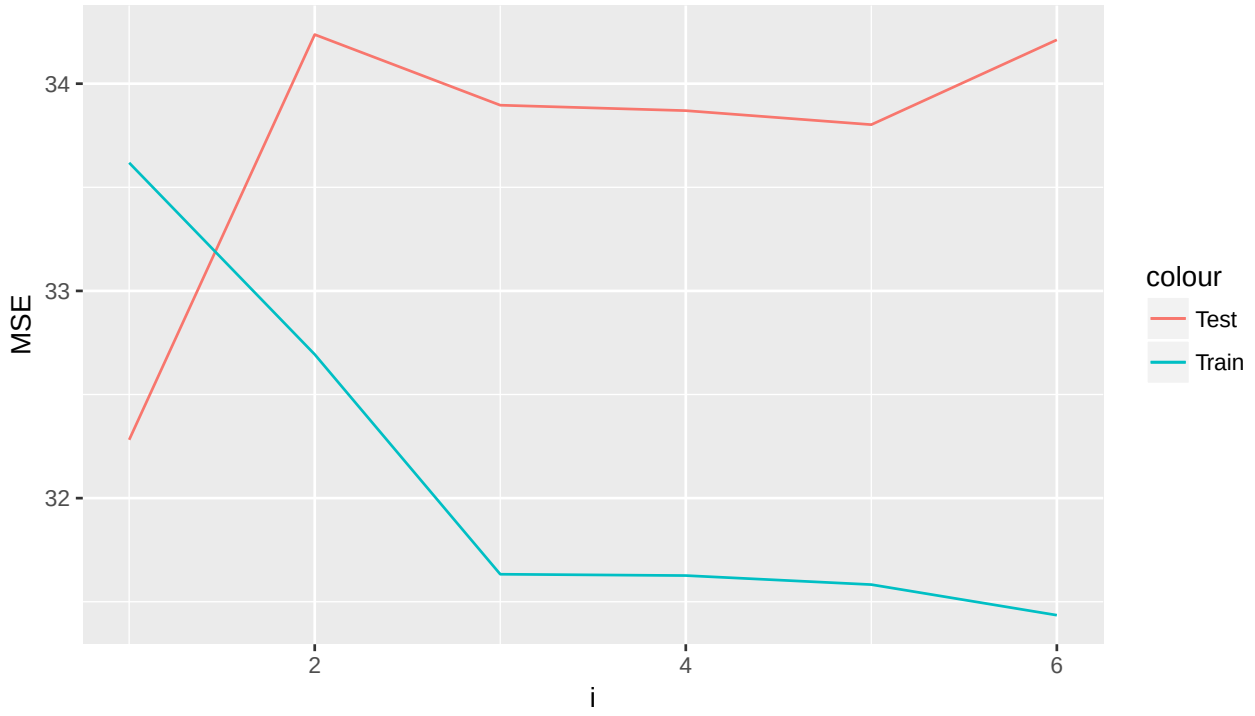

Figure 1.7: MSE dependency on i

the *stepAIC()* function is then used to do variable selection on a linear model where Fat is the response variable and Channel 1-100 is are predictors.

```
linearModel <- lm(Fat ~ . - Protein - Moisture - Sample, data = data )
aic <- stepAIC(linearModel, trace = FALSE)
```

The *stepAIC()* function presents a final model, consisting of 63 variables.

Next, a Ridge regression model is fit using the same response and predictors as in the previous linear model. Ridge regression uses L2 regularisation to penalise residuals when the parameters of a regression model are being learned. The coefficients that are the least efficient in the estimation will "shrink" the fastest. The glmnet function provides the functionality for ridge regression when parameter $\alpha = 0$. Ridge regression involves tuning the hyperparameter λ which glmnet will generate default values for you.

```
x=model.matrix(Fat~. - Protein - Moisture - Sample,data=data)
fit.ridge=glmnet(x,data$Fat,alpha=0)
```

Figure 1.8 shows how the coefficients are shrinked towards zero using Ridge regression. Each line represents a coefficient whose value converges to zero as λ is increasing. The top axis shows the number of non-zero coefficients. When λ is very low, the shrinkage seem to be quite ragged, but when λ increases the lines even out and shrink uniformly.

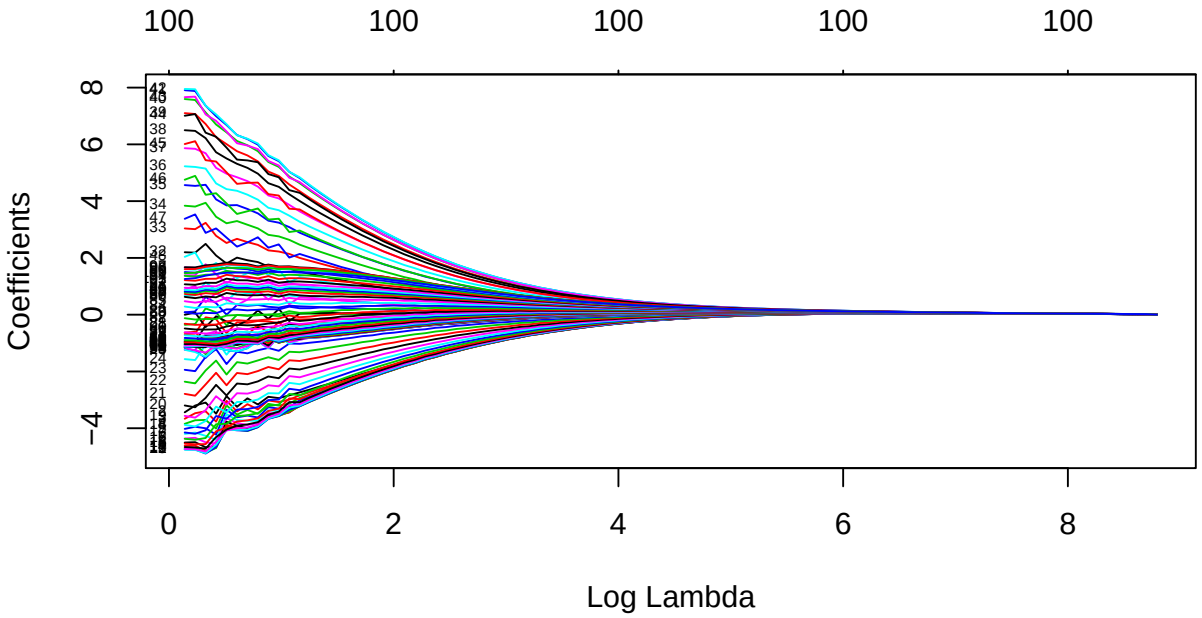

Figure 1.8: Ridge: Model coefficients dependency on lambda

When performing Lasso regression, the model is fit using the same variables as in the Ridge regression model, except *alpha* is 1, indicating that Lasso regression should be used. Lasso regression is similar to Ridge regression. It also penalize non-zero coefficients, but unlike Ridge regression which penalizes sum of squared coefficients (L2 regularisation), lasso penalizes the sum of their absolute values (L1 regularisation). This can result in coefficients being zeroed, which can not happen in Ridge regression.

```
fit.lasso=glmnet(x,data$Fat,alpha=1)
```

Figure 1.9 shows how the coefficients are shrinked towards zero using Lasso regression. Here, the coefficients shrink to zero in a less homogeneous way, compared to when using Ridge regression. Some predictors have large coefficients, while the rest are nearly zeroed. When λ increases, the number of non-zero coefficients decrease resulting in having all coefficients be put to zero when λ is 2.

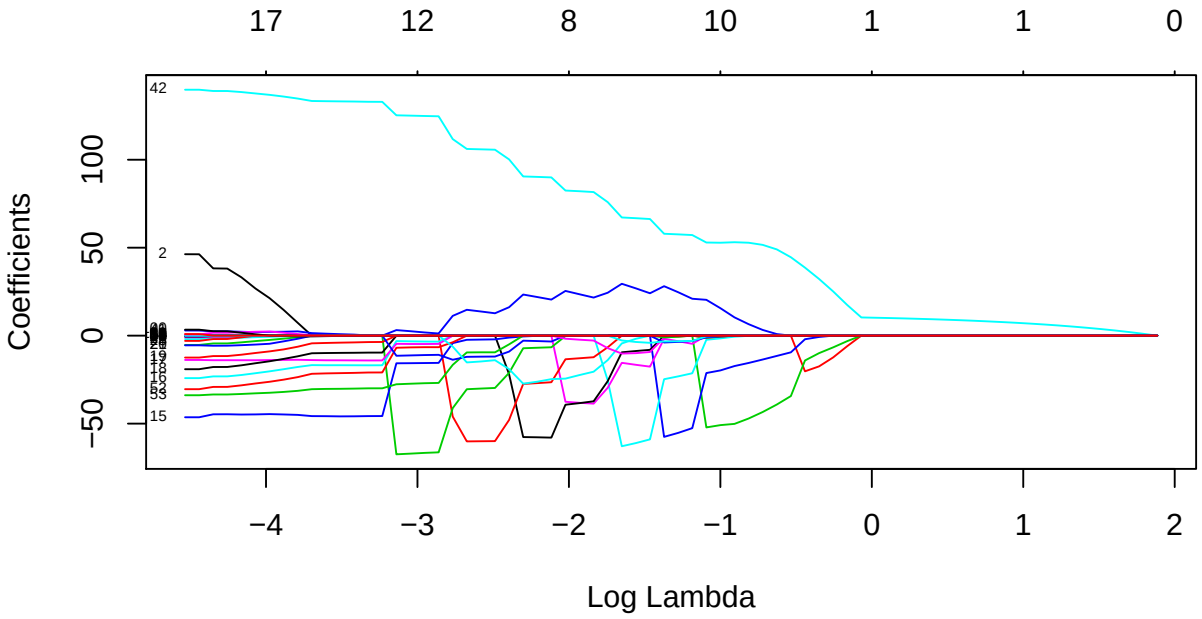

Figure 1.9: Lasso: Model coefficients dependency on lambda

Lastly, cross-validation is used to find the optimal Lasso model.

```
cv.lasso=cv.glmnet(x,data$Fat, alpha=1)
```

The λ value that gives the lowest MSE is 0.01072971 and 22 variables were chosen by the model. This is quite different from the results when using StepAIC, that chose 63 variables for the final model.

Figure 1.10 shows the MSE for each λ value. The dotted lines show the λ that gives the minimal MSE and the λ value that is within 1 standard error of the lowest error. As λ increases, the MSE also increases. When λ is between 0 and 1, the MSE is changing slowly, but when λ goes under 0, there is a rapid drop in MSE.

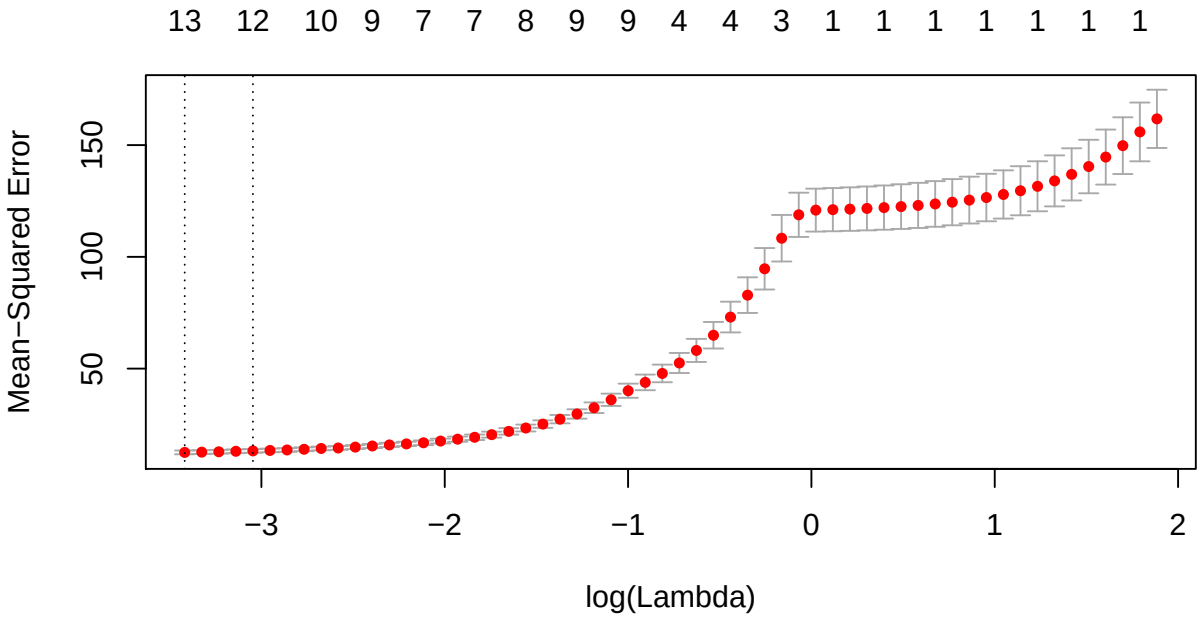

Figure 1.10: Lasso: MSE for different lambda

2. Code Appendix

2.1 Assignment 1

```r
# Loading data
library("readxl")
#library("caret")
library("xtable")
library("kknn")

data <- read_excel("spambase.xlsx")

# make spam categorical
data$Spam <- as.factor(data$Spam)

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

# Fit the model on train set and predict on train
logRegModel <- glm(Spam ~ ., data = train, family = binomial)

#predict on train set with 0.5 cutoff
train_probs <- predict(logRegModel, train, type = "response")
train_pred <- ifelse(train_probs > 0.5, "1", "0")
table("Actual" = train$Spam, "Predicted" = train_pred )
mean(train_pred != train$Spam)

#predict on test set with 0.5 cutoff
test_probs <- predict(logRegModel, test, type = "response")
test_pred <- ifelse(test_probs > 0.5, "1", "0")
table("Actual" = test$Spam, "Predicted" = test_pred )
mean(test_pred != test$Spam)

#predict on train set with 0.8 cutoff
train_pred08 <- ifelse(train_probs > 0.8, "1", "0")
table("Actual" = train$Spam, "Predicted" = train_pred08 )
mean(train_pred08 != train$Spam)

#predict on test set with 0.8 cutoff
test_pred08 <- ifelse(test_probs > 0.8, "1", "0")
table("Actual" = test$Spam, "Predicted" = test_pred08 )
mean(test_pred08 != test$Spam)

## KKNN with k = 30

kknn_classifier = kknn(Spam ~ ., train, train, k=30)
fit <- fitted(kknn_classifier)
```

```r
table(train$Spam, fit)
mean(fit != train$Spam)

kknn_classifier = kknn(Spam ~ ., train, test, k=30)
fit <- fitted(kknn_classifier)
table(test$Spam, fit)
mean(fit != test$Spam)

## KKNN with k = 1
kknn_classifier = kknn(Spam ~ ., train, train, k=1)
fit <- fitted(kknn_classifier)
table(train$Spam, fit)
mean(fit != train$Spam)

kknn_classifier = kknn(Spam ~ ., train, test, k=1)
fit <- fitted(kknn_classifier)
table(test$Spam, fit)
mean(fit != test$Spam)

##CLEAN UP
rm(list = ls())
gc()
cat("\014")
```

2.2 Assignment 2

```r
# Loading data
library("readxl")
require(ggplot2)
#install.packages('kknn', dependencies=TRUE)
data <- read_excel("machines.xlsx")
head(data)
d <- density(data$Length) # returns the density data
plot(d) # plots the results

loglike <- function(theta, vector) {
  return (length(vector)*log(theta) - theta*sum(vector))
}

l_theta <- function(theta, vector) {
  lambda <- 10
  prior <- lambda*exp(-lambda*theta)
  return (log(prior) + loglike(theta, vector))
}

theta <- seq(from=0.01, to=5.00, by=0.01)
log_likelihood <- loglike(theta, data$Length)
log_likelihood1 <- loglike(theta, data$Length[0:6])


log_res <-
  data.frame(
    var0 = log_likelihood,
    var1 = log_likelihood1,
    x = theta
  )

ggplot(log_res, aes(x)) +
  geom_line(aes(y = var0, colour = "Entire data set")) +
  geom_line(aes(y = var1, colour = "6 first observations")) +
  ylab('Log-likelihood')+xlab('Theta')
```

```r

# PART 4

bayesian <- l_theta(theta, data$Length)

log_res <-
  data.frame(
    var0 = log_likelihood,
    var1 = log_likelihood1,
    var2 = bayesian,
    x = theta
  )

ggplot(log_res, aes(x)) +
  geom_line(aes(y = var0, colour = "Entire data set")) +
  geom_line(aes(y = var1, colour = "6 first observations")) +
  geom_line(aes(y = var2, colour = "Bayesian")) +
  ylab('Log-likelihood')+xlab('Theta')

# PART 5

random_nmbrs <- hist(rexp(50, rate = 1.13), plot = FALSE)
orig_vals <- hist(data$Length, plot = FALSE)

plot( random_nmbrs, col=rgb(0,0,1,1/4), xlim=c(0,6))  # first histogram
plot( orig_vals, col=rgb(1,0,0,1/4), xlim=c(0,6), add=T)


##CLEAN UP
rm(list = ls())
gc()
cat("\014")
```

2.3 Assignment 4

```r
## PART 1: Load data
library("readxl")
#install.packages("glmnet", dependencies = TRUE)
library(Metrics)
library(ggplot2)
library(MASS)
require(glmnet)
data <- read_excel("tecator.xlsx")
head(data)
plot(data$Moisture, data$Protein, xlab = "Moisture", ylab = "Protein")

## PART 2:
#P(y) = b0 + b1x + ... + bIx^I + e


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

## Create models
M1 <- lm(Moisture ~ Protein, data = train )
M2 <- lm(Moisture ~ Protein + I(Protein^2), data = train )
```

```r
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), data = train )

# Predict M1
M1.test_preds <- predict(M1, test, type = "response")
M1.train_preds <- predict(M1, train, type = "response")
M1.test_mse <- mean((M1.test_preds - test$Moisture)^2)
M1.train_mse <- mean((M1.train_preds - train$Moisture)^2)

# Predict M2
M2.test_preds <- predict(M2, test, type = "response")
M2.train_preds <- predict(M2, train, type = "response")
M2.test_mse <- mean((M2.test_preds - test$Moisture)^2)
M2.train_mse <- mean((M2.train_preds - train$Moisture)^2)

# Predict M3
M3.test_preds <- predict(M3, test, type = "response")
M3.train_preds <- predict(M3, train, type = "response")
M3.test_mse <- mean((M3.test_preds - test$Moisture)^2)
M3.train_mse <- mean((M3.train_preds - train$Moisture)^2)

# Predict M4
M4.test_preds <- predict(M4, test, type = "response")
M4.train_preds <- predict(M4, train, type = "response")
M4.test_mse <- mean((M4.test_preds - test$Moisture)^2)
M4.train_mse <- mean((M4.train_preds - train$Moisture)^2)

# Predict M5
M5.test_preds <- predict(M5, test, type = "response")
M5.train_preds <- predict(M5, train, type = "response")
M5.test_mse <- mean((M5.test_preds - test$Moisture)^2)
M5.train_mse <- mean((M5.train_preds - train$Moisture)^2)

# Predict M6
M6.test_preds <- predict(M6, test, type = "response")
M6.train_preds <- predict(M6, train, type = "response")
M6.test_mse <- mean((M6.test_preds - test$Moisture)^2)
M6.train_mse <- mean((M6.train_preds - train$Moisture)^2)

MSE.test<- c(M1.test_mse,M2.test_mse, M3.test_mse, M4.test_mse, M5.test_mse, M6.test_mse)
MSE.train<- c(M1.train_mse,M2.train_mse, M3.train_mse, M4.train_mse, M5.train_mse, M6.train_mse)
x <- seq(1, 6, by=1)

MSE_res <-
  data.frame(
    var0 = MSE.test,
    var1 = MSE.train,
    x = x
  )

ggplot(MSE_res, aes(x)) +
  geom_line(aes(y = var0, colour = "Test")) +
  geom_line(aes(y = var1, colour = "Train")) +
  ylab('MSE')+xlab('i')


## Part 4
```

```r
linearModel <- lm(Fat ~ . - Protein - Moisture - Sample, data = data )
aic <- stepAIC(linearModel)
aic$anova

## Part 5
x=model.matrix(Fat~. - Protein - Moisture - Sample,data=data)

fit.ridge=glmnet(x,data$Fat,alpha=0)
plot(fit.ridge,xvar="lambda",label=TRUE)

fit.lasso=glmnet(x,data$Fat,alpha=1)
plot(fit.lasso,xvar="lambda",label=TRUE)

## cross-validation
cv.lasso=cv.glmnet(x,data$Fat, alpha=1)
cv.lasso$lambda.min
coef(cv.lasso, s="lambda.min")
plot(cv.lasso)

##CLEAN UP
rm(list = ls())
gc()
cat("\014")
```