Laboration 1 - TDDE01

William Bergekrans, Anna Dahlsjö, Uno Österman

2020-11-23

Assignment 1

Responsible: William Bergekrans.

Assignment 1 is about handwritten digit recognition with K-means.

1. Split the Data

We started with dividing the data into three sets, training-, validation- and a test-set. The training set is 50% of the data, validation and test sets are 25% of data each.

2. Fitting with kknn

First, a 30-nearest neighbor classifier is fitted using the kknn() function from the kknn package. The rectangular kernel is used which is a standard unweighted knn.

Confusion Table

The resulting confusion table when we predict using the test-set is as follows:

Table 1: Confusion Table

	0	1	2	3	4	5	6	7	8	9
0	77	0	0	0	0	0	0	0	0	0
1	0	81	0	0	0	1	0	0	7	1
2	0	2	98	0	0	1	0	0	0	1
3	0	0	0	107	0	0	0	1	1	1
4	1	0	0	0	94	0	0	0	0	0
5	0	0	0	2	0	93	0	0	0	0
6	0	0	0	0	2	2	90	0	0	0
7	0	0	0	0	6	1	0	111	0	1
8	0	0	3	1	2	0	0	0	70	0
9	0	3	0	1	5	5	0	0	0	85

X-axis is the true values and Y-axis is the predictions.

The number that the model is best at predicting is zero as all the predictions of zero are true. The worst performances are on 1, 7, 8 and 9. In the training set there are more data for low and high numbers. The distributon of the target data in the training set is:

Var1	Freq
0	202
1	195
2	192
3	188
4	175
5	181
6	192
7	182
8	204
9	200

The test set show the opposite distribution and has more data for targets that the training set has less data for.

Var1	Freq
0	78
1	86
2	101
3	111
4	109
5	103
6	90
7	112
8	78
9	89

The results indicates that certain digits are naturally more alike and therefore more difficult to predict. If there was large faults in the model we should see more bad predictions for all targets.

Misclassification Rate

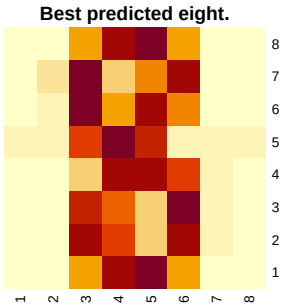
The misclassification error is calculated with the formula (1/N)*Sum(y!=y-hat) for every prediction in the test set. The misclassification rate for the training set is 0.0450026. The misclassification rate for the test set is 0.0532915. This means that 5 percent of the predictions made by the model on the test set are wrong and 4.5 percet for the training set. As the two error rates are very similar and that the error for the training set is not close to 0 we can draw the conclusion that the model is not overfitted. An overfitted model would perform much better on the training set than on the test set.

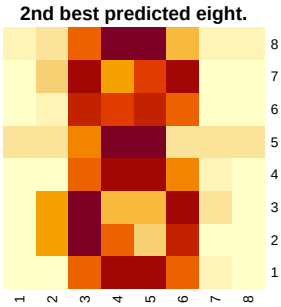
If this is a good misclassification rate depends on the cost of misclassification. If the model is used in banking such an error is probably too large because it can have huge consequences. For uses where the cost of misclassification is low, this model is probably good enough. It performs better than guessing.

3. Hard numbers to classify

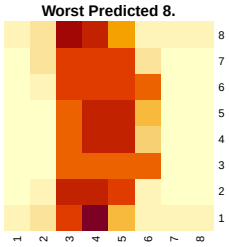
For the number 8 there are several data observations that has the probability one of being correct. Which means that the model is absolutely sure it is correct. However, there are also cases where the probability of the model labeling the data correctly as eight are very low. The three worst probabilites are: 0.1, 0.1333333, 0.2666667.

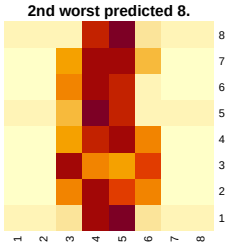
The two highest probabilities for 8 look as follows:

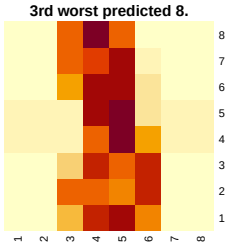




In the two examples above the shape of an 8 can clearly be seen.



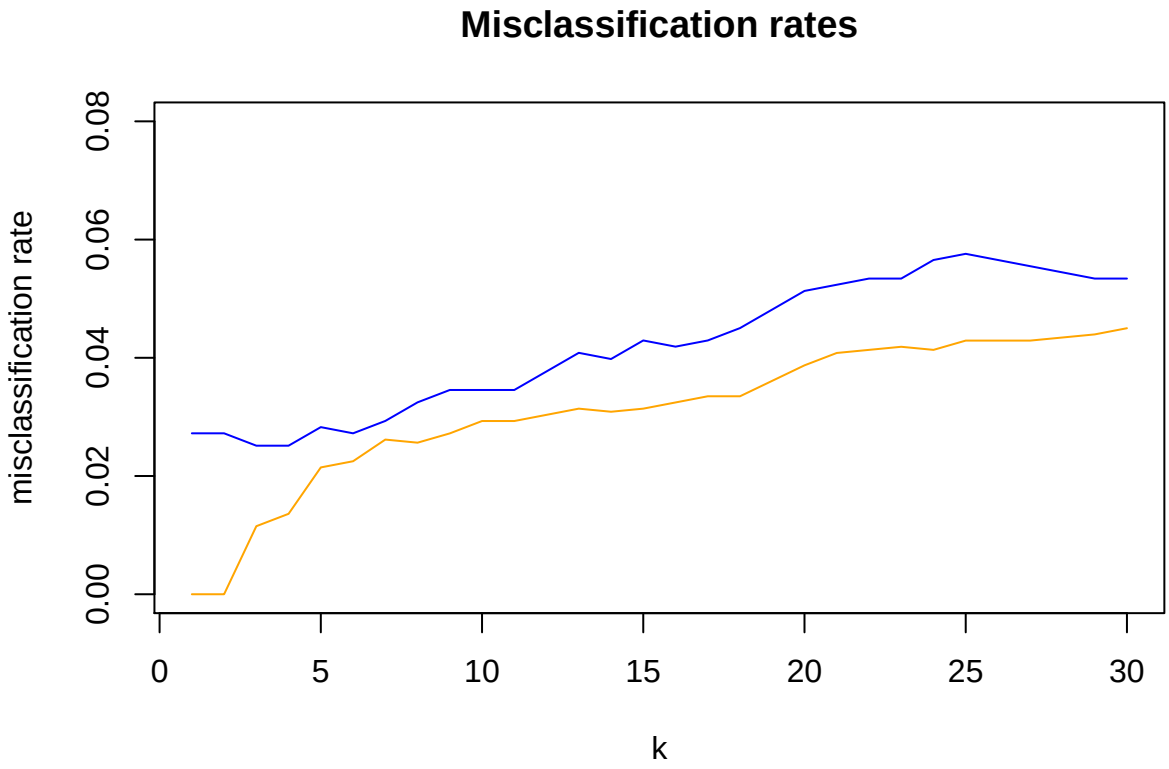




Above is the heatmaps of the eights with the lowest probability of being labeled as an eight. They are not recognizable by a naked eye as being and eight compared to the best cases which clearly can be identified as eights based on their heatmaps.

4. KNN fitting with different K-values

The misclassification for the training (orange) and validation (blue) sets are:



Orange is for the training set, blue is the validation set.

In knn having a small k means that the model basically memorizes the date and the training misclassifation error will therefore be zero. The model complexity increases whith higher Ks as more neighbors are taken into account. For the validation set the same trend can be seen but for low k the model will generally predict wrong where the training and validation set differs. For higher Ks we see that they follow quite a similar trend with a quite constant difference between the two sets, where the performance on the validation set is worse which is expected.

Overfitting and underfitting is strongly connected to bias and variance. Overfitting indicated that the variance is high with low bias and underfitting is the oposite. When the training set has a very low misclassification error this indicates that the model has a high variance and with a low bias for the training set. For higher Ks we see that the training and validation error increases and approach a flatter curve, with is more bias in the model but less varaince.

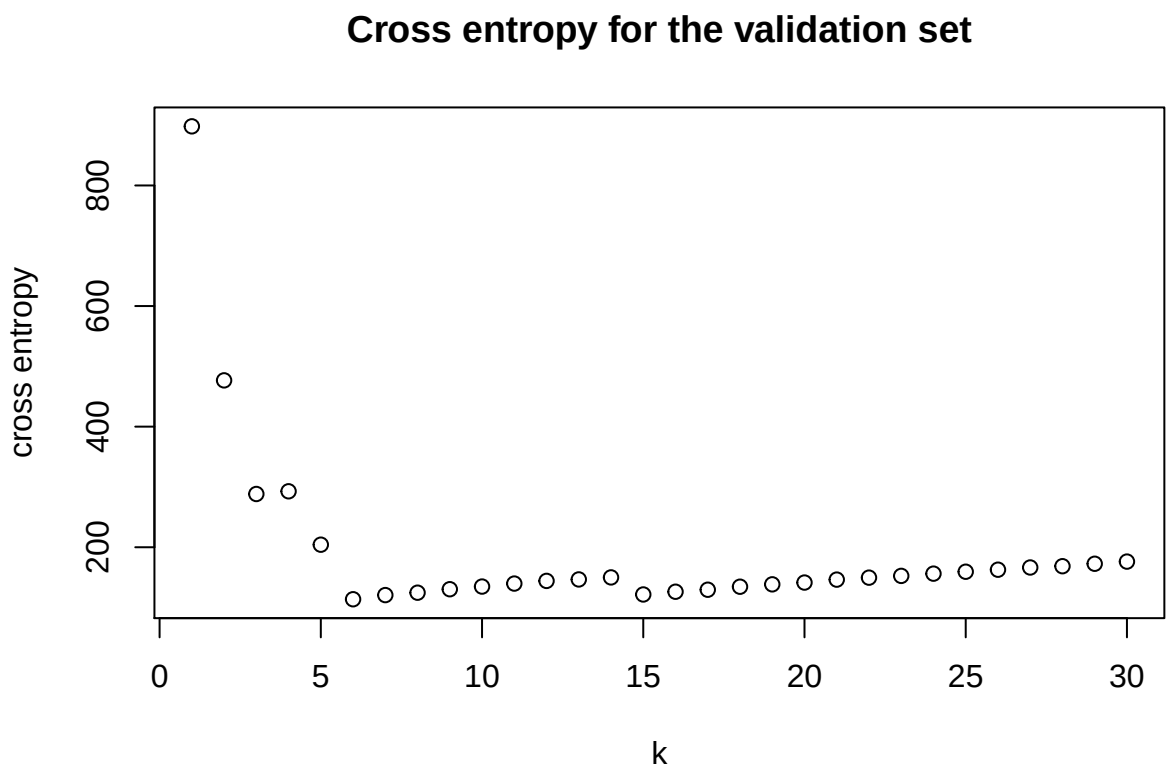
The best K according to the graphs seem to be k=4 as this gives the lowest misclassification rate for the validation set. Next we calculate the misclassification rate on the test set using the fitted model that had k=4.

```
##                         Training set Validation set   Test set
## misclassification rate   0.01360544     0.02513089 0.02507837
```

The test set and validation set has very similar misclassification rate which is a good thing and indicates that we choose a good k-value to use.

5. Cross-entropy calculation of empirical risk

We have once again fitted the knn model for k-values ranging from 1 to 30 using the kknn() function. We provided the validation set to the function and therefore got predictions for this set.



The result shows that the best predictions on the test set are still when k = 6 which is a higher k than what we would have choosen using the misclassification error. Cross-entropy may be more suited as a empirical risk function as it takes the closeness of predictions into account. The misclassification error blindly looks at the number of correct predictions. In this case where digits may look similar, how close the model was to choose the correct prediction may be very different for different Ks. The cross-entropy can reflect this and we see that a slightly less complex model is better according to the cross-entropy.

Assignment 2

Responible: Anna Dahlsjö.

This assignment is about Ridge regression and Model selection.

1. Write down the probabilistic model as a Bayesian model

P(motor_UPDRS|w,sigma) = N(motor_UPDRS|w0 + Xw, sigma$^{2I)\ w\ =\ N(0,\ (1/lambda)\text{sigma}}$2*I)

2. Scale the data

We started by reading the csv file and store it as global data set, called parkinsons_data, the we scaled the data and divided it into two sets; training and test. The training set is 60% of the data and the test set is test set.

3. Implement 4 functions

Now, we want to create the loglikelihood function which we later will use in the Ridge regression function. This will help us find the optimal parameters to predict the motor_UPDRS from the features.

The degrees of freedom for the different lambdas and the training set are: lambda = 1 -> 13.86 lambda = 100 -> 9.93 lambda = 1000 -> 5.64

4. Compute optimal w and sigma parameters and then then MSE for both training and test data

The optimal parameters will now be computed using the ridgeOpt function and we can determine, by using MSE, which model (lambda) is the most accurate to the predict the motor_UPDRS values.

The most appropriate penalty parameter for both the test and training data is lambda = 1 with a MSE of 0.099 for the training data and MSE of 0.068 for th test data. We want the MSE to be as small as possible. Mean Square Error is a more appropriate measure here because it assigns more weight to the bigger errors, and we don't want any big error in this type of prediction. We also want to take into consideration values that are both higher and lower than the real value and therefore this value is better then MAE which weights these values the same.

5. AIC

Through the Akaike's information criterion (AIC) we will compute which model is the best fit to the data in an other way.

The optimal model according to the AIC criterion is the one with the smallest value which is given by lambda = 1 for both the training and test data, 9554 and 6477 respectively. The theoretical advantage of this kind of model compared to the holdout model is that it consider in-sample risk, both the performance and its complexity - in comparison to only the complexity for the holdout model above.

Assignment 3

Responsible: Uno Österman

This part deals with linear regression using the LASSO method

Dividing the data

First we need to divide the data into training and testing data, a randomly selected 50/50 split

task 1

Make a linear regression model for fat where absorbency char. are features.

```
# --------------------
# Assignment 3
# --------------------

# import libraries
# library(readr)
library(ggplot2)
library(glmnet)
```

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

```
## Loaded glmnet 4.0-2
```

```
#Read the Data using the readr package.
tecator = read.csv("tecator.csv")
#Split the data into training and testing data
```

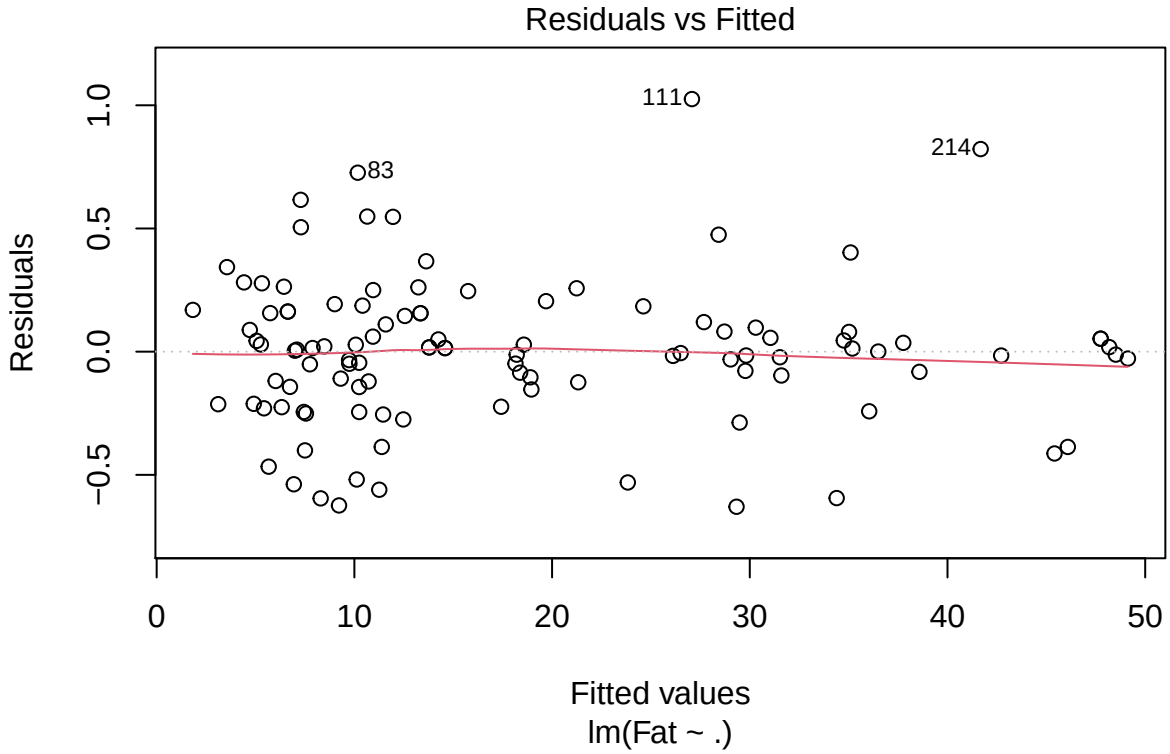
```r
n = dim(tecator)[1]
#We use set.seed() and a starting number, 12345, to generate a random sequence of number that will give
set.seed(12345)
#Divide the data in half (0.5)
id = sample(1:n, floor(n*0.5))
#assign the training data
train = tecator[-id,]
#assign the test data
test = tecator[id,]


#Fitting linear model for fat and absorbency
fat_model = lm(Fat~., data = train[,2:102])
#Produce result summary
summary(fat_model)
```

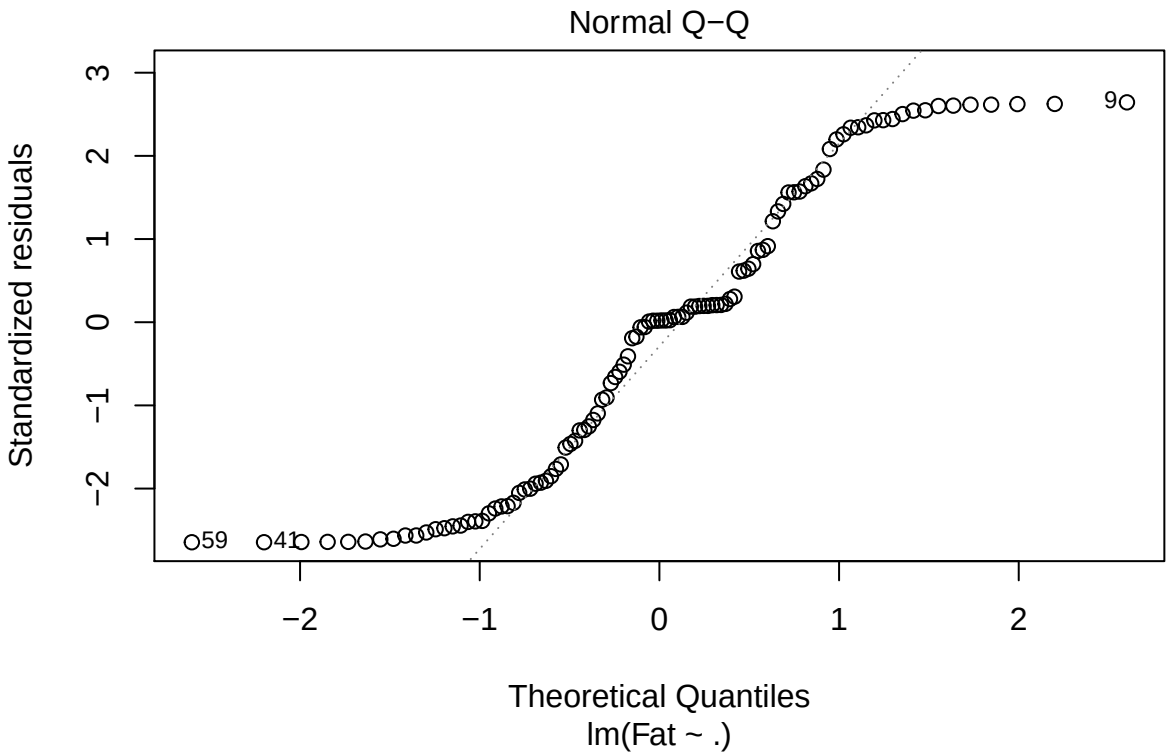
```
##
## Call:
## lm(formula = Fat ~ ., data = train[, 2:102])
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.62940 -0.14304  0.01048  0.15595  1.02521
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.103e+00  4.394e+00  -0.251    0.809
## Channel1     2.285e+04  1.899e+04   1.203    0.268
## Channel2    -3.275e+04  3.545e+04  -0.924    0.386
## Channel3    -4.294e+03  4.868e+04  -0.088    0.932
## Channel4     5.515e+04  8.929e+04   0.618    0.556
## Channel5    -6.697e+04  1.094e+05  -0.612    0.560
## Channel6     2.958e+04  8.925e+04   0.331    0.750
## Channel7    -4.738e+04  6.822e+04  -0.695    0.510
## Channel8     3.342e+04  3.479e+04   0.961    0.369
## Channel9    -2.866e+03  2.271e+04  -0.126    0.903
## Channel10    2.325e+04  5.182e+04   0.449    0.667
## Channel11   -5.137e+04  7.593e+04  -0.676    0.520
## Channel12    1.159e+05  1.023e+05   1.133    0.294
## Channel13   -4.259e+04  1.243e+05  -0.343    0.742
## Channel14   -3.076e+04  1.174e+05  -0.262    0.801
## Channel15    1.391e+04  8.790e+04   0.158    0.879
## Channel16   -3.186e+04  5.067e+04  -0.629    0.549
## Channel17   -4.284e+03  2.541e+04  -0.169    0.871
## Channel18    4.212e+04  3.440e+04   1.224    0.260
## Channel19   -6.181e+04  6.030e+04  -1.025    0.339
## Channel20    1.193e+05  1.255e+05   0.950    0.374
## Channel21   -2.272e+05  1.703e+05  -1.334    0.224
## Channel22    2.845e+05  1.802e+05   1.578    0.158
## Channel23   -2.317e+05  1.543e+05  -1.501    0.177
## Channel24    1.528e+05  1.280e+05   1.193    0.272
## Channel25   -7.904e+04  9.043e+04  -0.874    0.411
## Channel26    2.205e+04  3.115e+04   0.708    0.502
## Channel27   -3.459e+03  5.017e+04  -0.069    0.947
```

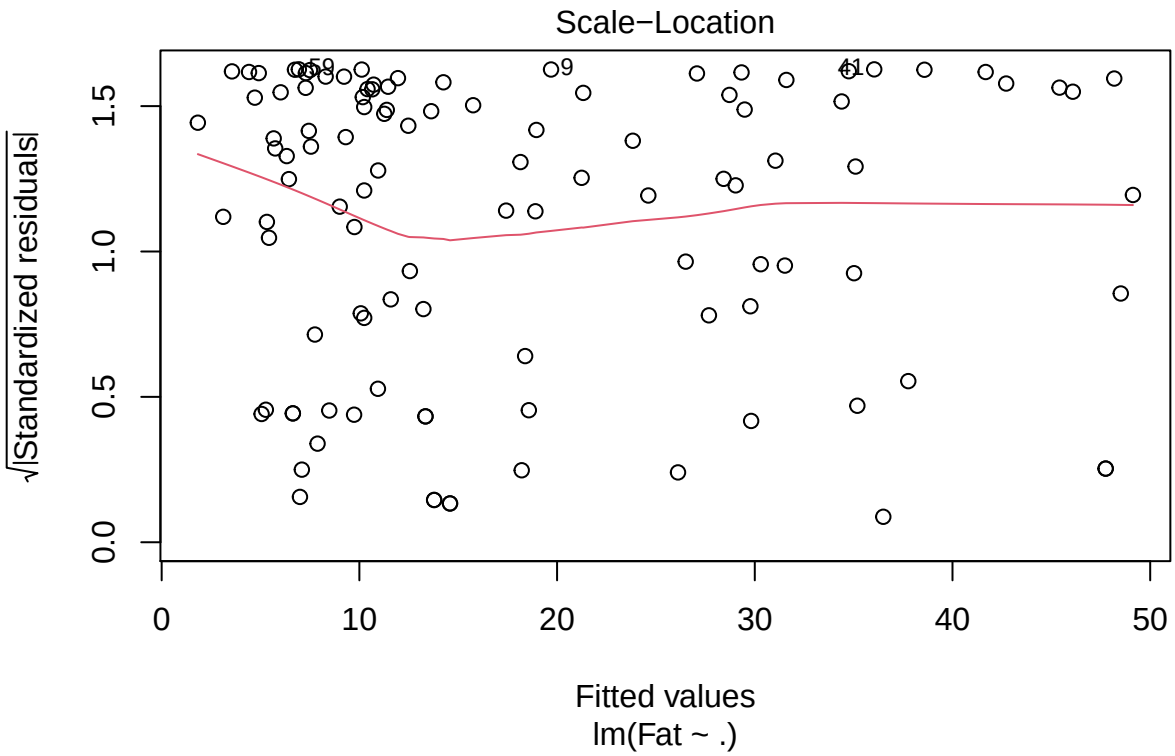
```
## Channel28    7.402e+04  5.113e+04   1.448    0.191
## Channel29   -1.917e+05  1.168e+05  -1.641    0.145
## Channel30    2.530e+05  1.377e+05   1.838    0.109
## Channel31   -2.305e+05  1.284e+05  -1.796    0.116
## Channel32    1.513e+05  1.267e+05   1.194    0.271
## Channel33   -6.027e+04  7.816e+04  -0.771    0.466
## Channel34    2.033e+04  8.490e+04   0.239    0.818
## Channel35   -2.198e+04  7.412e+04  -0.297    0.775
## Channel36   -8.665e+03  2.884e+04  -0.300    0.773
## Channel37   -8.937e+02  2.051e+04  -0.044    0.966
## Channel38    3.879e+04  3.033e+04   1.279    0.242
## Channel39   -6.190e+03  3.381e+04  -0.183    0.860
## Channel40   -3.198e+04  7.458e+04  -0.429    0.681
## Channel41    2.516e+04  1.365e+05   0.184    0.859
## Channel42    1.824e+04  1.221e+05   0.149    0.885
## Channel43   -7.572e+04  8.476e+04  -0.893    0.401
## Channel44    6.748e+04  6.432e+04   1.049    0.329
## Channel45    2.415e+03  4.373e+04   0.055    0.957
## Channel46   -2.947e+04  4.474e+04  -0.659    0.531
## Channel47    8.802e+03  1.886e+04   0.467    0.655
## Channel48    1.198e+02  2.398e+04   0.005    0.996
## Channel49   -9.484e+03  5.591e+04  -0.170    0.870
## Channel50    2.491e+04  5.874e+04   0.424    0.684
## Channel51    2.485e+04  5.611e+04   0.443    0.671
## Channel52   -1.212e+05  9.951e+04  -1.218    0.263
## Channel53    1.335e+05  1.249e+05   1.069    0.320
## Channel54   -6.570e+04  1.055e+05  -0.623    0.553
## Channel55    2.495e+04  6.596e+04   0.378    0.716
## Channel56   -6.734e+03  2.642e+04  -0.255    0.806
## Channel57   -2.395e+04  3.479e+04  -0.688    0.513
## Channel58    2.592e+04  3.207e+04   0.808    0.446
## Channel59   -2.369e+04  1.964e+04  -1.206    0.267
## Channel60    2.834e+04  3.312e+04   0.856    0.421
## Channel61   -6.166e+03  2.169e+04  -0.284    0.784
## Channel62    1.564e+04  2.053e+04   0.762    0.471
## Channel63   -1.866e+04  2.029e+04  -0.920    0.388
## Channel64   -6.514e+03  4.106e+04  -0.159    0.878
## Channel65    4.727e+04  8.471e+04   0.558    0.594
## Channel66   -9.974e+04  1.048e+05  -0.952    0.373
## Channel67    8.323e+04  9.810e+04   0.848    0.424
## Channel68   -3.676e+04  8.440e+04  -0.436    0.676
## Channel69    1.918e+04  6.159e+04   0.311    0.765
## Channel70   -2.469e+04  2.897e+04  -0.852    0.422
## Channel71    3.638e+04  3.484e+04   1.044    0.331
## Channel72   -2.748e+04  3.618e+04  -0.759    0.472
## Channel73   -6.648e+03  3.435e+04  -0.194    0.852
## Channel74    1.625e+04  3.303e+04   0.492    0.638
## Channel75   -6.345e+03  3.779e+04  -0.168    0.871
## Channel76    7.170e+03  3.213e+04   0.223    0.830
## Channel77   -1.833e+04  2.455e+04  -0.747    0.480
## Channel78    1.840e+04  2.705e+04   0.680    0.518
## Channel79   -1.287e+03  3.102e+04  -0.041    0.968
## Channel80    2.800e+04  2.579e+04   1.086    0.314
## Channel81   -3.227e+04  2.967e+04  -1.087    0.313
```

```
## Channel82   -6.592e+03  4.358e+04  -0.151    0.884
## Channel83   -1.828e+04  6.173e+04  -0.296    0.776
## Channel84    5.988e+04  6.608e+04   0.906    0.395
## Channel85   -4.833e+04  5.709e+04  -0.847    0.425
## Channel86    4.882e+04  6.810e+04   0.717    0.497
## Channel87   -6.241e+04  6.264e+04  -0.996    0.352
## Channel88    4.451e+04  6.118e+04   0.728    0.490
## Channel89    1.682e+04  5.564e+04   0.302    0.771
## Channel90   -6.311e+04  9.198e+04  -0.686    0.515
## Channel91    3.292e+04  1.004e+05   0.328    0.753
## Channel92   -2.224e+04  9.367e+04  -0.237    0.819
## Channel93    3.475e+04  7.074e+04   0.491    0.638
## Channel94   -3.323e+04  6.154e+04  -0.540    0.606
## Channel95    5.249e+04  4.529e+04   1.159    0.284
## Channel96   -6.676e+04  3.800e+04  -1.757    0.122
## Channel97    7.727e+04  5.196e+04   1.487    0.181
## Channel98   -5.564e+04  6.241e+04  -0.891    0.402
## Channel99   -6.791e+01  6.462e+04  -0.001    0.999
## Channel100   1.208e+04  2.640e+04   0.458    0.661
##
## Residual standard error: 1.159 on 7 degrees of freedom
## Multiple R-squared:  0.9995, Adjusted R-squared:  0.992
## F-statistic: 133.3 on 100 and 7 DF,  p-value: 2.687e-07
```

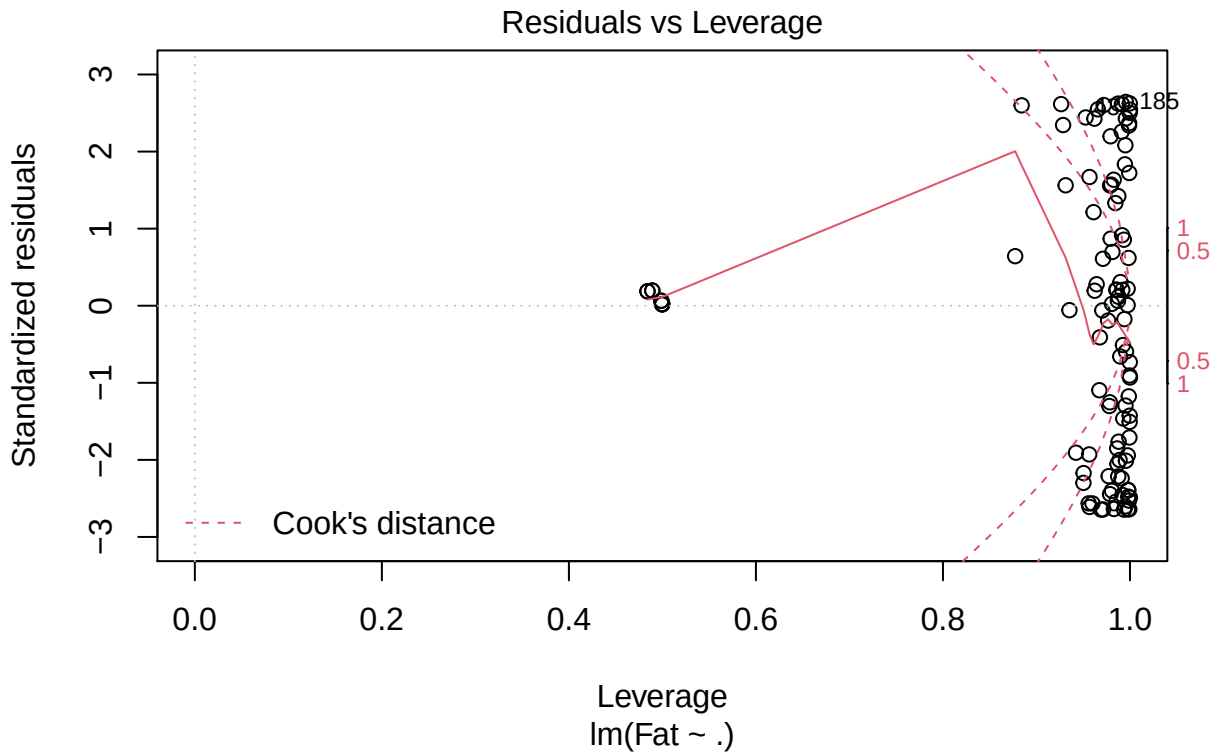
```
#plot model
plot(fat_model)
```




Normal Q−Q
Standardized residuals
3
2
1
0
−1
−2
59
41
9
−2
−1
0
1
2
Theoretical Quantiles
lm(Fat ~ .)




```
## Warning in sqrt(crit * p * (1 - hh)/hh): NaNs produced

## Warning in sqrt(crit * p * (1 - hh)/hh): NaNs produced
```




```
# We now want to test the model with the test and training data

fat_train = predict(fat_model, data = train[,2:101])
fat_test = predict(fat_model, data = test[,2:101])

# Calculating Mean squared error, MSE = sum(value - predicted_value)^2/N
# Take out the columns for fat data
train_column_fat = train$Fat
test_column_fat = test$Fat

# Create MSE function
MSE = function(real_fat, pred_fat)  {
  MSE = sum((real_fat - pred_fat)^2)/length(real_fat)
  return(MSE)
}
train_MSE = MSE(train_column_fat, fat_train)
test_MSE = MSE(test_column_fat, fat_test)
```

```
## Warning in real_fat - pred_fat: longer object length is not a multiple of
## shorter object length
```

```
train_MSE
```

```
## [1] 0.08701866
```

```
test_MSE
```

```
## [1] 306.2159
```

Report underlying prob. model, est. train&test Comment on quality of fit and prediction -> quality of model

The prob. model: Y = XB + error = B[0] + X[1]B[1] + X[2]B[2] + ... + X[p]B[p] + E. Where E~(N(0, sigma^2))

When looking at the mean squeared error (MSE) we find the the model fits the training set really well, the error is only 0.087. However, for the testing set the error is significantly larger, at 306.216. This indictes that the model is overfittning from the testdata and has too high veriance, although the good fit for the training data tells us it has low bias. This is due to the bias-veriance trade of, where overfitting occurs when the bias is low but the veriance is high and underfitting when the bias is too large and veriance is too low.

The overfitting in this case probably occurs because of the model has a large number of parameters, 100 variables and becomes to complex with high veriance and low bias.

Task 2

Fat modeled as LASSO Regression, channels used as features. Report objective function that should be optimized in this scenario.

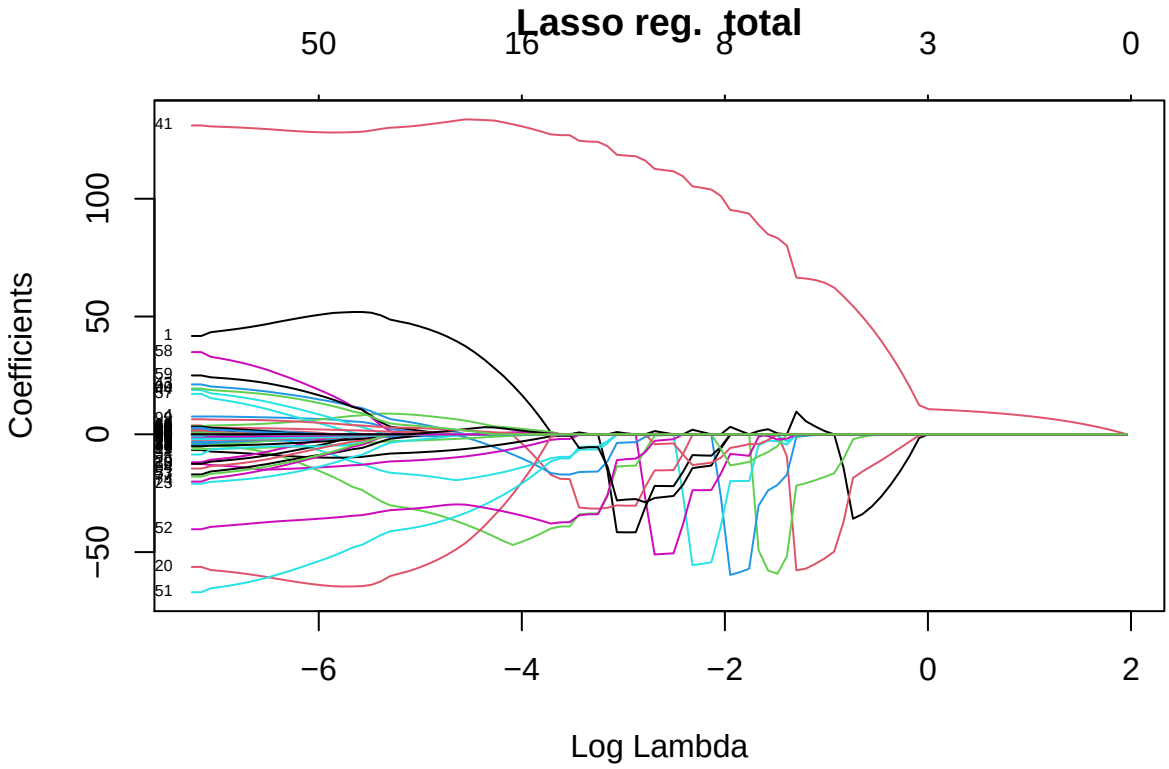
From slides in lecture 1d.we get that the Objective function is –>

w(hat(Lasso)) = argmin sumi = 1 .. N + lambda*sumj = 1 .. N Where lambda > 0 is the penelty factor. I differs from Ridge regression by having an absolute value in the penalty function, which can be 0. Ridge Regression can only be close but never actually 0 since it has a quaddratic penalty function

TASK 3

Do as shown in lecture 1d for Ridge regression

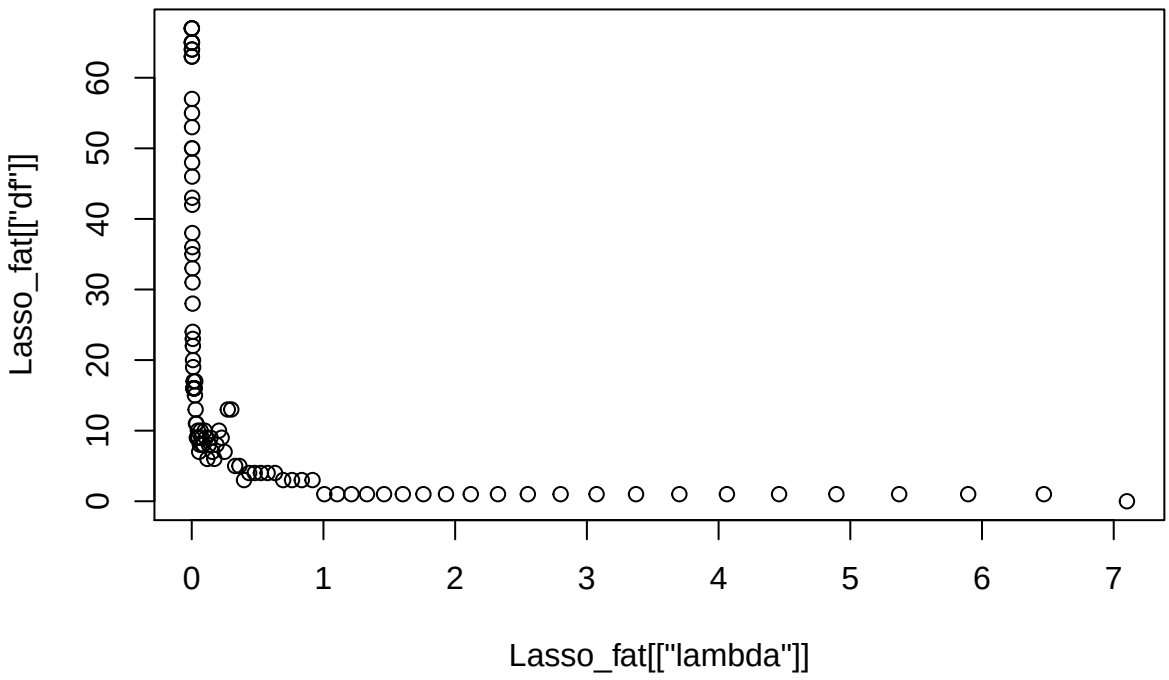
```
covariates = train[,2:101]

response = train[,102]

Lasso_fat = glmnet(as.matrix(covariates), response, alpha = 1, family = "gaussian")

plot(Lasso_fat, xvar = "lambda", label = TRUE, main = "Lasso reg. \ total")
```



In the figure shown above we see when which features are larger than zero for different log(Lambda) values in the Lasso Regression.Around the value log(lambda) = -0.5 up to around log(lambda) = 0 we find that there only three features that are separated from zero.

Task4

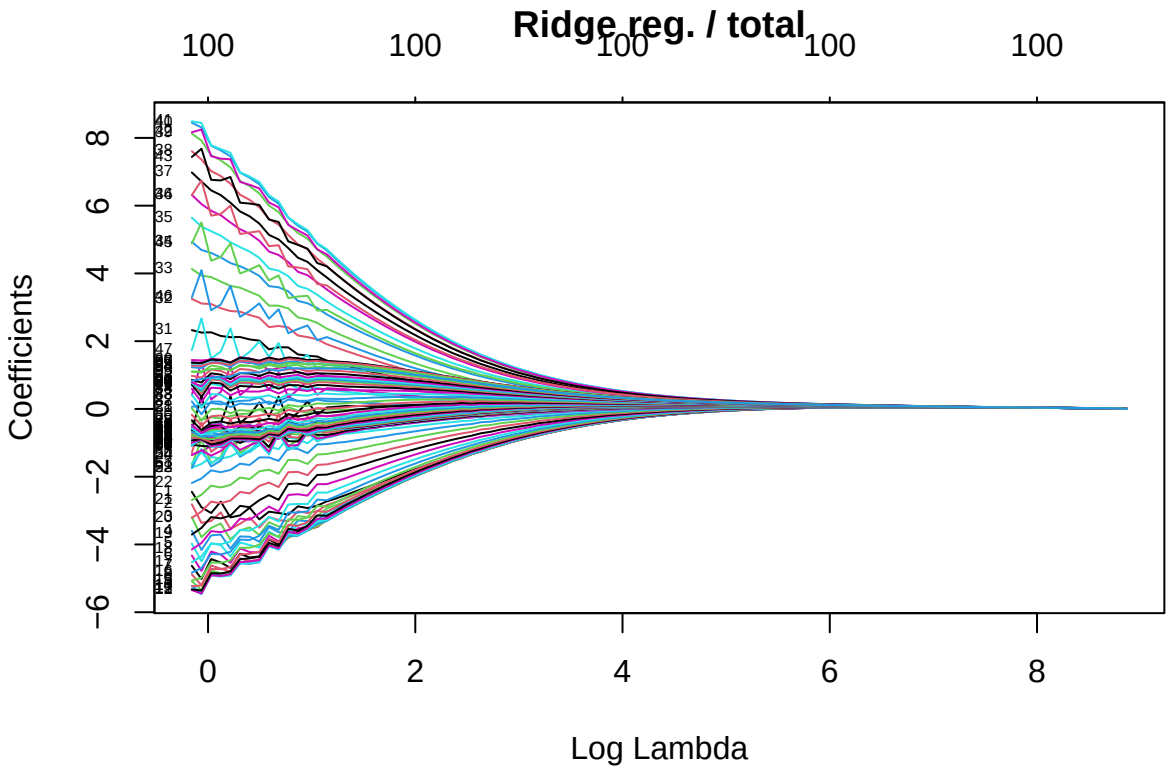
```
plot(Lasso_fat[['lambda']], Lasso_fat[['df']])
```



In the figure shown above we see a plot that describes how the degrees of freedom change depending on the value of lambda. More parameters will give higher degrees of freedom and as we can see the degrees of freedom are reduced with an increasing value of lambda. Since a higher values of lambda increases the amount of zero-valued parameters the degrees of freedom is also decreased.

Task 5

```
fat_Ridge_model = glmnet(as.matrix(covariates), response, alpha = 0, family = "gaussian")

plot(fat_Ridge_model, xvar = "lambda", label = TRUE, main = "Ridge reg. / total")
```



In this figure we see a plot of the ridge regression and if we compare it to the figure in 3.4 we see that in the ridge regression all parameters are separated from zero. This is because in ridge regression we use a squared penalty value, which can never be zero, meaning we always use all parameters. In the Lasso regression we instead have an absolute penalty value which can give parameters a value of zero, which means that we can ignore certain channels with larger lambda values.

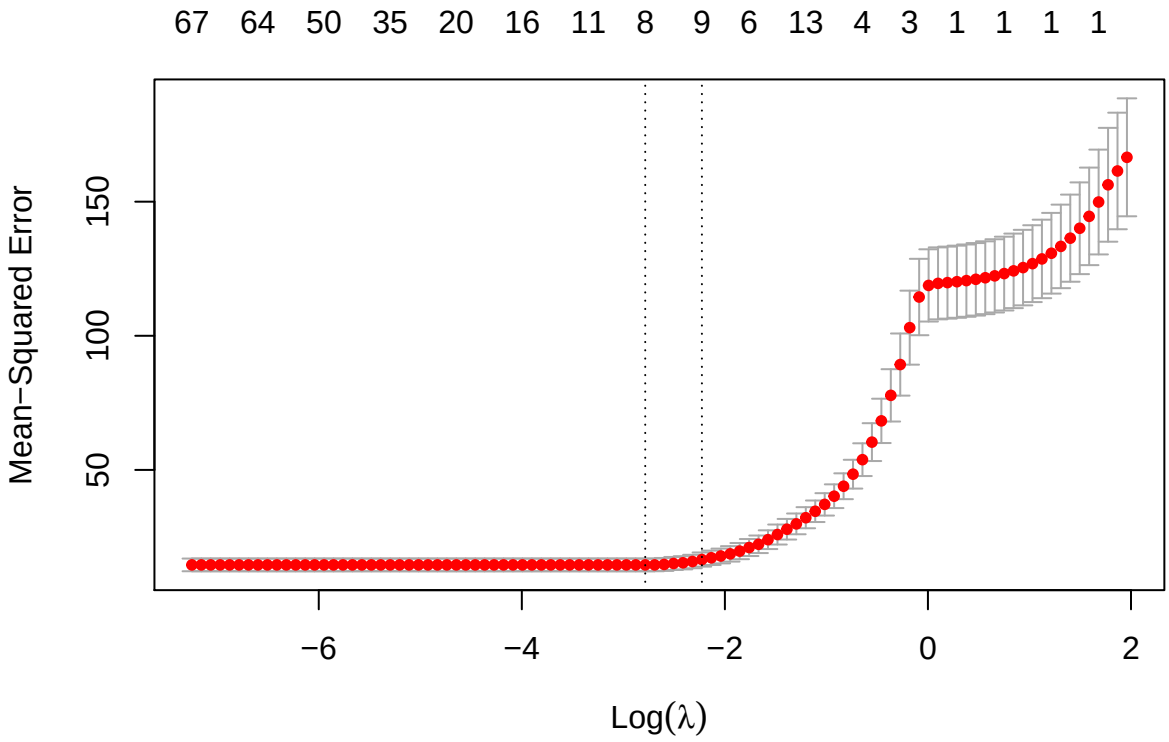
In ridge regression as the value of lambda increases the significance of more extreme channels can tend towards zero, which leads to lower veriance and lower bias. In Lasso regression we can set some of these more extreme channels to absolute zero, which reduces the number of features.

Task 6

```
CV_Lasso_fat = cv.glmnet(as.matrix(covariates), response, alpha = 1,family = "gaussian", type.measure =
```

```
## Warning: Only mse, deviance, mae available as type.measure for Gaussian models;
## mse used instead
```

```
plot(CV_Lasso_fat)
```




```
coef(CV_Lasso_fat, s = "lambda.min")
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
##                        1
## (Intercept)  30.075298
## Channel1      .
## Channel2      .
## Channel3      .
## Channel4      .
## Channel5      .
## Channel6      .
## Channel7      .
## Channel8      .
## Channel9      .
## Channel10     .
## Channel11     .
## Channel12   -27.085632
## Channel13   -30.980052
## Channel14   -22.359741
## Channel15    -6.757625
## Channel16    -1.131232
## Channel17     .
## Channel18     .
## Channel19     .
## Channel20     .
## Channel21     .
```

```
## Channel22     .
## Channel23     .
## Channel24     .
## Channel25     .
## Channel26     .
## Channel27     .
## Channel28     .
## Channel29     .
## Channel30     .
## Channel31     .
## Channel32     .
## Channel33     .
## Channel34     .
## Channel35     .
## Channel36     .
## Channel37     .
## Channel38     .
## Channel39     .
## Channel40     .
## Channel41   116.214495
## Channel42     .
## Channel43     .
## Channel44     .
## Channel45     .
## Channel46     .
## Channel47     .
## Channel48     .
## Channel49     .
## Channel50     .
## Channel51     .
## Channel52    -7.574339
## Channel53   -28.886522
## Channel54     .
## Channel55     .
## Channel56     .
## Channel57     .
## Channel58     .
## Channel59     .
## Channel60     .
## Channel61     .
## Channel62     .
## Channel63     .
## Channel64     .
## Channel65     .
## Channel66     .
## Channel67     .
## Channel68     .
## Channel69     .
## Channel70     .
## Channel71     .
## Channel72     .
## Channel73     .
## Channel74     .
## Channel75     .
```

```
## Channel76     .
## Channel77     .
## Channel78     .
## Channel79     .
## Channel80     .
## Channel81     .
## Channel82     .
## Channel83     .
## Channel84     .
## Channel85     .
## Channel86     .
## Channel87     .
## Channel88     .
## Channel89     .
## Channel90     .
## Channel91     .
## Channel92     .
## Channel93     .
## Channel94     .
## Channel95     .
## Channel96     .
## Channel97     .
## Channel98     .
## Channel99     .
## Channel100    .
```

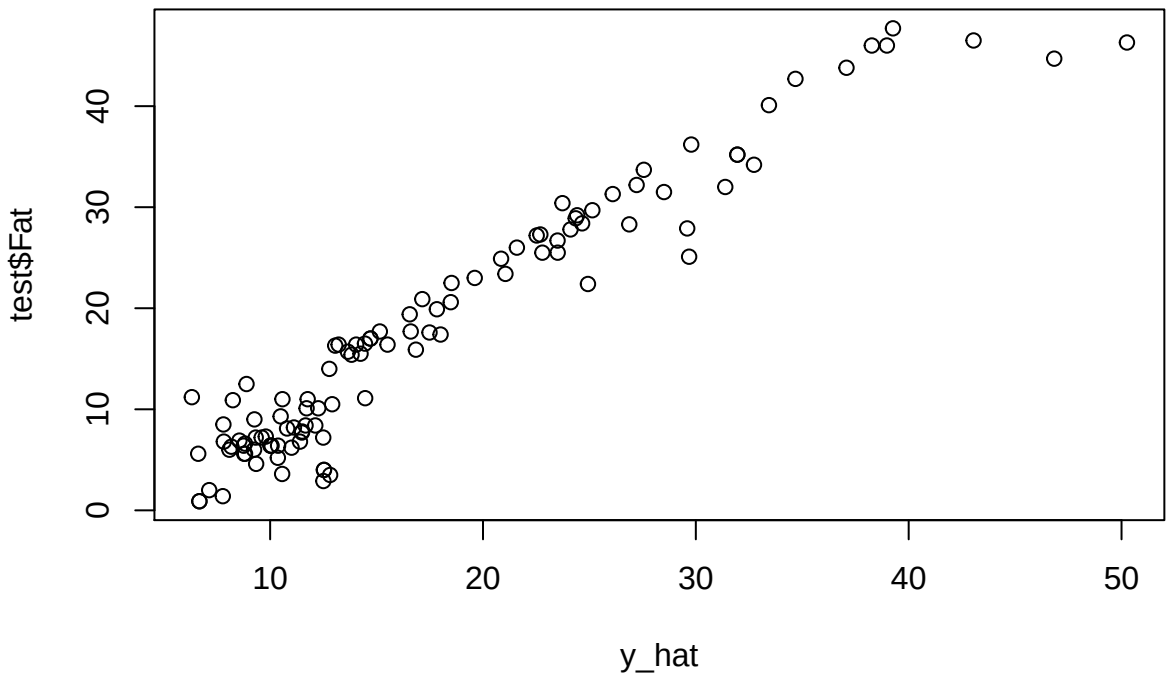
```
print(log(CV_Lasso_fat$lambda.min))
```

```
## [1] -2.784556
```

```
y_hat = predict(CV_Lasso_fat, newx=as.matrix(test[,2:101]), s = "lambda.min")
# yhat

CV_Lasso_fat[["cvsd"]]
```

```
##   [1] 21.986259 21.728694 21.220824 19.536249 18.181076 17.099253 16.242199
##   [8] 15.568170 15.041738 14.633256 14.318275 14.076900 13.893136 13.754249
##  [15] 13.650182 13.573038 13.516647 13.476201 13.447961 13.429029 13.417155
##  [22] 13.463307 14.241714 13.789893 11.583962  9.750675  8.263228  7.054201
##  [29]  6.102240  5.366363  4.818532  4.428288  4.175004  4.012031  3.933047
##  [36]  3.871176  3.842989  3.727776  3.494377  3.271864  3.165466  3.013863
##  [43]  2.876065  2.756805  2.678753  2.634259  2.506968  2.459292  2.431500
##  [50]  2.418790  2.363947  2.358603  2.361770  2.361770  2.361770  2.361770
##  [57]  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770
##  [64]  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770
##  [71]  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770
##  [78]  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770
##  [85]  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770
##  [92]  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770  2.361770
##  [99]  2.361770  2.361770
```

```
plot(y_hat, test$Fat)
```



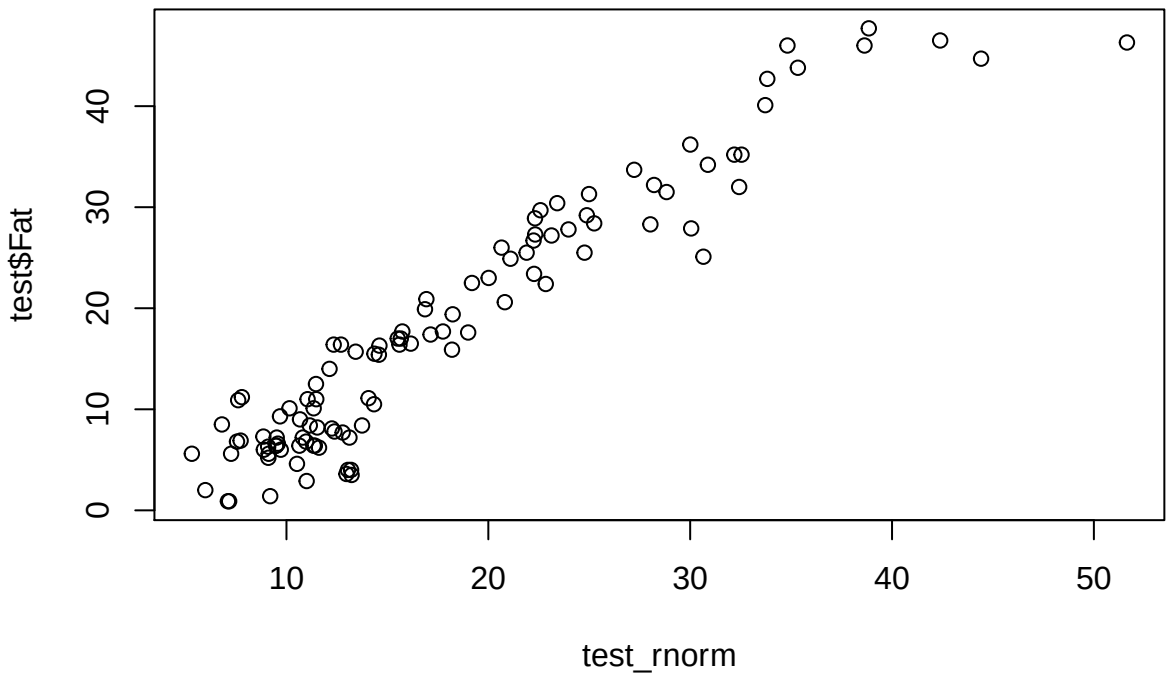
In the first figure we see a plot of the cv-score depending on different lambda values. We get the optimal lambda value at log(lambda) = -2.785 and as seen in the plot the model then uses 8 variables. The confidence interval for the optimal value goes from down to slightly above log(lambda) = -2, meaning the selected value is statistically better than log(lambda) = -2.

In the second figure we see a scatter plot of how our predicted test values compared to the original test values. The model predictions are quite good, however we can see that for the lower values (around 10) its more difficult for the model to predict, both under- and overshooting. Also, we see that for the highest values (slightly above 40) the model tends to overshoot the predictions.

Task 7

Use the optimal lambda, y_hat, from task 6, and get sd, standard deviation

```
sd = sqrt(sum(test$Fat - y_hat)^2/107)

# Rnorm - random varible normal distribution.
test_rnorm = rnorm(107, y_hat, sd)
plot(test_rnorm, test$Fat)
```



In the last figure we see the test_norm as a normal distribution of the predicted test values compared to the original test values. The figure is similar but not identical with the figure from 3.6. However the model is now using standard deviation and normal distribution meaning that the model gets a more even but higher spread of the scattered plots, as we can see when comparing the figures. Though, the model acts very similar to the model in 3.6 and tend to have similar faults.

R code

```
library(knitr)
knitr::opts_chunk$set(echo = TRUE)
# --------------------
# Assignment 1
# --------------------

# 1. Importing and spliting data.

# Read the data using the readr package.
library(readr)
optdigits <- read_csv("optdigits.csv",col_names = FALSE)
# The target have to be a nominal value in order for
# the kknn prediction to work. We tranform target X65 using
# as.factor function.
optdigits$X65 <- as.factor(optdigits$X65)
```

```r
# Split the data into training, validation and test set.
n <- dim(optdigits)[1]
# We use a random number generator, every time we use 12345 we will
# be able to get the exact same results.
set.seed(12345)

# Id for the training set, which is halv of the total data.
id <- sample(1:n, floor(n*0.5))
opt.training <- optdigits[id,] # The data for training.

# Next we want to divide the rest-data into validation- and test-set.
id1 <- setdiff(1:n, id)
set.seed(12345)
id2 <- sample(id1, floor(n*0.25))
opt.valid <- optdigits[id2,]

id3 <- setdiff(id1,id2)
opt.test <- optdigits[id3,]

# Remove the variables we no longer use.
rm(optdigits)
rm(n)
rm(id)
rm(id1)
rm(id2)
rm(id3)
# Import library kknn
# Needs to be installed on the computer.
library(kknn)

# This also works. However the predictions are not exactly the same.
# Probably becase of the "scale" and "Kmax" settings.
# opt.kknn2 <- train.kknn(X65~.,opt.training,KMAX=30)
# table(predict(opt.kknn2,opt.test),opt.test$X65)

# Fit data using the kknn package. 30 neighbors.
opt.kknn <- kknn(formula = X65 ~ .,
                 # ~. Means that we use all other variables to predict X65.
                 train = opt.training,
                 test = opt.test,
                 scale = TRUE,
                 k = 30,
                 kernel = "rectangular")

confusion_table <- table(opt.kknn$fit, opt.test$X65)

# Do the same thing but with the training set as the test set.
kknn.training <- kknn(formula = X65 ~ .,
                 # ~. Means that we use all other variables to predict X65.
                 train = opt.training,
                 test = opt.training,
                 scale = TRUE,
                 k = 30,
```

```
                kernel = "rectangular")

missclass=function(X,X1){
  n=length(X)
  return(1-sum(diag(table(X,X1)))/n)
}
mc_kknn <- missclass(opt.test$X65, opt.kknn$fit)
mc.kknn.training <- missclass(opt.training$X65,kknn.training$fit)
# First we add the probabilities for the test data to a table
# with all the test data. We do this so we can sort the data in regards
# to the probabilites.
test.prob <- opt.test
prob8 <- opt.kknn$prob[,9]

test.prob$prob <- prob8
rm(prob8)
# Filter out so we only have rows with X65 = 8.
# (Eight is the target value).
library(dplyr) # Package that has the filter function we use.
test.prob <- filter(test.prob, X65 ==8)

test.prob <- test.prob[order(test.prob$prob),]

worst3 <- head(test.prob, n =3)
best2 <- tail(test.prob, n=2)

# Jag tror vi kanske endast ska kolla på de värden som faktiskt har 8 som sant värde.
# så måste ta ut dessa rader ut opt.test :)
rowToMatrix=function(V){
  A = matrix(,8,8)
  x = 1
  for(row in 1:8) {
    for(col in 1:8) {
      A[row,col] <- (V[x])
      x <- x+1
    }
}
  rm(x)
  return(A)
}

bestRow1 <- rowToMatrix(as.numeric(best2[1,]))
bestRow2 <- rowToMatrix(as.numeric(best2[2,]))

worstRow1 <- rowToMatrix(as.numeric(worst3[1,]))
worstRow2 <- rowToMatrix(as.numeric(worst3[2,]))
worstRow3 <- rowToMatrix(as.numeric(worst3[3,]))
heatmap(bestRow1, Colv=NA, Rowv=NA, main="Best predicted eight.")
heatmap(bestRow2, Colv=NA, Rowv=NA, main="2nd best predicted eight.")
heatmap(worstRow1, Colv=NA, Rowv=NA, main="Worst Predicted 8.")
heatmap(worstRow2, Colv=NA, Rowv=NA, main="2nd worst predicted 8.")
heatmap(worstRow3, Colv=NA, Rowv=NA, main="3rd worst predicted 8.")
# Create dataframes to store misclassification rate in.
```

```
k <- c()
misclassRate <- c()
mc.training <- data.frame(k, misclassRate)
mc.valid <- data.frame(k, misclassRate)

# Do knn fitting with k-values ranging from 1 to 30.
for (i in 1:30) {
  k <- train.kknn(formula = X65 ~ .,
                  data = opt.training,
                  ks = i,
                  kernel = "rectangular")

   # Predictions for the training data set.
  pred.k <- predict(k, opt.training)
  # Calculate the misclassification error.
  mc <- missclass(opt.training$X65, pred.k)
  mc.training <- rbind(mc.training, c(i, mc))

  # Predictions for the validation data set.
  pred.k <- predict(k, opt.valid)
  # Calculate the misclassification error.
  mc <- missclass(opt.valid$X65, pred.k)
  mc.valid <- rbind(mc.valid, c(i, mc))
}
# Remove unnecessary variables.
rm(mc); rm(pred.k); rm(k)
plot(mc.training$X1,mc.training$X0, type="l", ylim = c(0,0.08), xlab = "k", ylab = "misclassification ra
points(mc.valid$X1, mc.valid$X0.0272251308900524, type="l", col="blue")
k <- train.kknn(formula = X65 ~ .,
                  data = opt.training,
                  ks = 4,
                  kernel = "rectangular")

# Predictions for the test data set.
pred.k <- predict(k, opt.test)

# Calculate the misclassification error.
mc4k <- missclass(opt.test$X65, pred.k)

matrix(c(mc.training[4,2],mc.valid[4,2],mc4k),nrow=1,ncol=3,dimnames = list("misclassification rate", c
k <- c()
crossEntropy <- c()
ce.valid <- data.frame(k, crossEntropy)

# Next we fitt the model for all different Ks.
for (i in 1:30) {
  k <- kknn(formula = X65 ~ .,
            train = opt.training,
            test = opt.valid,
            k = i,
            kernel = "rectangular")

  # Calculation of the cross-entropy
```

```
  x <- 0
  for (j in 1:nrow(opt.valid)) {
    # Get the probability of the correct class for each observation.
    phat = k$prob[j,as.numeric(opt.valid$X65[j])]
    x = x + log(phat+(1e-15))
  }

  ce.valid <- rbind(ce.valid, c(i, (-x)))
}

plot(ce.valid, ylab="cross entropy", xlab="k", main="Cross entropy for the validation set")

# --------------------
# Assignment 2
# --------------------

# Read the data
parkinsons_data <- read.csv("parkinsons.csv")

#Scale the data
parkinsons_data = scale(parkinsons_data)

#Divide it into training and test data
n = dim(parkinsons_data)[1]
set.seed(12345)
# Random number generator, every time we use 12345 we will
# be able to get the exact same results.
id = sample(1:n, floor(n*0.6))
train_data = parkinsons_data[id,]
test_data = parkinsons_data[-id,]

# a)
loglikelihood = function(w, sigma, data){
  #As input we have the w-vector, sigma and the relevant data
  n = dim(data)[1]
  Y = data[,5] #the fifth column in the data set is the real motor_UPDRS values
  X = data[,7:22] #the feature values in in the 7-22 column
  sigma_sq = sigma^2
  return(-n/2*log(2*pi)-n/2*log(sigma_sq)-t(Y-X%*%w)%*%(Y-X%*%w)/(2*sigma_sq))
}

#The ridge regression function is created by using the loglikelihood function above
#and by adding the penalty scalar lambda
#This function will be called from the ridgeOpt function below

# b)
ridge_function = function(w, lambda, data) {  #w is a vector with 17x1 rows
  sigma = w[1] #the first value in  w is the sigma
  w = w[-1] #the remaining 16 values is the w-vector
  return(-loglikelihood(w, sigma, data)+lambda*sum(w^2))
}

#The ridgeOpt function will now be created using
```

```r
#the optim function to find the optimal parameters for the current lambda

#c)
ridgeOpt = function(lambda, data) {
  w <- rep(1,17)
  opt <- optim(par = w,fn = ridge_function, lambda = lambda, data = data, method="BFGS")
  return(opt)
}

#The function for calculating the degrees of freedom for different lambdas is now created
#accordingly to the lecture slides. We also know that the degrees of freedom decrease when lambda incre

# d)
df_function = function(lambda, data) {
  x <- data[,7:22]
  P <- x%*%solve(t(x)%*%x+lambda*diag(dim(x)[2]))%*%t(x)
  return(sum(diag(P)))
}


sigma_1_train <- ridgeOpt(1, train_data)$par[1]
sigma_1_test <- ridgeOpt(1, test_data)$par[1]

sigma_2_train<- ridgeOpt(100, train_data)$par[1]
sigma_2_test<- ridgeOpt(100, test_data)$par[1]

sigma_3_train <- ridgeOpt(1000, train_data)$par[1]
sigma_3_test <- ridgeOpt(1000, test_data)$par[1]

w_1_train <- ridgeOpt(1,train_data)$par[2:17]
w_1_test <- ridgeOpt(1,test_data)$par[2:17]

w_2_train <- ridgeOpt(100, train_data)$par[2:17]
w_2_test <- ridgeOpt(100, test_data)$par[2:17]

w_3_train <- ridgeOpt(1000, train_data)$par[2:17]
w_3_test <- ridgeOpt(1000, test_data)$par[2:17]

Y_train = train_data[,5]
Y_test = test_data[,5]

X_train = train_data[,7:22]
X_test = test_data[,7:22]

predic_Y_1_train <- X_train%*%w_1_train
predic_Y_1_test <- X_test%*%w_1_test

predic_Y_2_train <- X_train%*%w_2_train
predic_Y_2_test <- X_test%*%w_2_train

predic_Y_3_train <- X_train%*%w_3_train
predic_Y_3_test <- X_test%*%w_3_test
```

```
MSE_1_train <- (1/n)*(sum(predic_Y_1_train - Y_train)^2)
MSE_1_test <- (1/n)*(sum(predic_Y_1_test - Y_test)^2)

MSE_2_train <- (1/n)*(sum(predic_Y_2_train - Y_train)^2)
MSE_2_test <- (1/n)*(sum(predic_Y_2_test - Y_test)^2)

MSE_3_train <- (1/n)*(sum(predic_Y_3_train -  Y_train)^2)
MSE_3_test <- (1/n)*(sum(predic_Y_3_test - Y_test)^2)

AIC_1_train <- 2*df_function(1, train_data) - 2*loglikelihood(w_1_train, sigma_1_train, train_data)
AIC_1_test <- 2*df_function(1, test_data) - 2*loglikelihood(w_1_test, sigma_1_test, test_data)

AIC_2_train <- 2*df_function(100, train_data) - 2*loglikelihood(w_2_train, sigma_2_train, train_data)
AIC_2_test <- 2*df_function(100, test_data) - 2*loglikelihood(w_2_test, sigma_2_test, test_data)

AIC_3_train <- 2*df_function(1000, train_data) - 2*loglikelihood(w_3_train, sigma_3_train, train_data)
AIC_3_test <- 2*df_function(1000, test_data) - 2*loglikelihood(w_3_test, sigma_3_test, test_data)
# --------------------
# Assignment 3
# --------------------

# import libraries
# library(readr)
library(ggplot2)
library(glmnet)

#Read the Data using the readr package.
tecator = read.csv("tecator.csv")
#Split the data into training and testing data
n = dim(tecator)[1]
#We use set.seed() and a starting number, 12345, to generate a random sequence of number that will give
set.seed(12345)
#Divide the data in half (0.5)
id = sample(1:n, floor(n*0.5))
#assign the training data
train = tecator[-id,]
#assign the test data
test = tecator[id,]


#Fitting linear model for fat and absorbency
fat_model = lm(Fat~., data = train[,2:102])
#Produce result summary
summary(fat_model)
#plot model
plot(fat_model)

# We now want to test the model with the test and training data

fat_train = predict(fat_model, data = train[,2:101])
fat_test = predict(fat_model, data = test[,2:101])

# Calculating Mean squared error, MSE = sum(value - predicted_value)^2/N
```

```r
# Take out the columns for fat data
train_column_fat = train$Fat
test_column_fat = test$Fat

# Create MSE function
MSE = function(real_fat, pred_fat)  {
  MSE = sum((real_fat - pred_fat)^2)/length(real_fat)
  return(MSE)
}
train_MSE = MSE(train_column_fat, fat_train)
test_MSE = MSE(test_column_fat, fat_test)
train_MSE
test_MSE

covariates = train[,2:101]

response = train[,102]

Lasso_fat = glmnet(as.matrix(covariates), response, alpha = 1, family = "gaussian")

plot(Lasso_fat, xvar = "lambda", label = TRUE, main = "Lasso reg. \ total")
plot(Lasso_fat[['lambda']], Lasso_fat[['df']])
fat_Ridge_model = glmnet(as.matrix(covariates), response, alpha = 0, family = "gaussian")

plot(fat_Ridge_model, xvar = "lambda", label = TRUE, main = "Ridge reg. / total")
CV_Lasso_fat = cv.glmnet(as.matrix(covariates), response, alpha = 1,family = "gaussian", type.measure =

plot(CV_Lasso_fat)

coef(CV_Lasso_fat, s = "lambda.min")

print(log(CV_Lasso_fat$lambda.min))

y_hat = predict(CV_Lasso_fat, newx=as.matrix(test[,2:101]), s = "lambda.min")
# yhat

CV_Lasso_fat[["cvsd"]]

plot(y_hat, test$Fat)
sd = sqrt(sum(test$Fat - y_hat)^2/107)

# Rnorm - random varible normal distribution.
test_rnorm = rnorm(107, y_hat, sd)
plot(test_rnorm, test$Fat)
```