Practical Session 3 - Modèles de Régression Linéaire

Richard CHEAM & Menghor THUO

16-11-2023

Contents

Function implementation

- The functions below are implemented in order to facilitate the work later.

```
logistic_performance <- function(Y_hat, Y){

  #initialize confusion matrix
  confusion_matrix = table(Predicted = Y_hat, Observed = Y)
  cat("Confusion matrix:\n\n")
  print(confusion_matrix)
  cat("\n")

  #initialize the components to measure the quality of the model from matrix
  TN <- confusion_matrix[1,1]
  TP <- confusion_matrix[2,2]
  FN <- confusion_matrix[1,2]
  FP <- confusion_matrix[2,1]
  TOTAL <- TN + TP + FN + FP
```

```
  #find accuracy, error_rate,...,FNR, of the model
  accuracy <- (TP + TN)/TOTAL
  error_rate <- (FP + FN)/TOTAL
  recall <- TP/(FN + TP)
  specificity <- TN/(TN + FP)
  precision <- TP/(FP + TP)
  FPR <- FP/(TN + FP)
  FNR <- FN/(FN + TP)

  #display those values
  cat("Accuracy:", accuracy*100, "%", "\n")
  cat("Error rate: ", error_rate*100, "%", "\n")
  cat("Recall: ", recall*100, "%", "\n")
  cat("Specificity: ", specificity*100, "%", "\n")
  cat("Precision: ", precision*100, "%", "\n")
  cat("False positive rate:", FPR*100, "%", "\n")
  cat("False negative rate:", FNR*100, "%", "\n")

  #return a list containing all the values for accessibility
  return(list(accuracy=accuracy, error_rate=error_rate, recall=recall,
              specificity=specificity, precision=precision, FPR=FPR, FNR=FNR))
}
```

```
MSE <- function(Y, Y_hat){
  obs = length(Y)
  return ((1/obs) * sum((Y-Y_hat)^2))
}
```

II. Cookies Study

- First, we read the cookies dataset as a dataframe and load into a variable called df. We then separate the Y dependent variable (fat) and the X explanatory variables (spectra information), where the number of observations, $n = 32$, and the predictors, $p = 700$.

```
#read the data from csv file
df = read.csv(file="cookies.csv", sep=',', dec='.', header = TRUE)

#initialize a column fat to be Y
Y <- df$fat

#every column except Y is X
X <- subset(df, select = -fat)
```

- Since the aim of this study is to explain if, for one cookie, the fat is greater or not than the median of the fat values for each the cookies, the variable $YBin$ was then taken into account as a binary target instead of fat.

```
#to get binary target whether the fat (Y) is greater or not than its median
YBin = as.numeric(Y > median(Y))
```

Logistic regression model using features

- Since the spectra of each cookie is sum-up in 5 characteristics (called features), which are the mean, the standard deviation, the slope, the minimum and the maximum of each spectra, hence, it is necessary to calculate them before study a logistic model.

Calculating features of each cookie

- Each calculation of the features are taken from the section *V.CookiesStudy* of the 1st practical session.

```
#calculate mean of each cookie by applying mean() function on each row of matrix X
mean <- apply(X, 1, mean)

#calculate standard deviation of each cookie
sd <- apply(X , 1, sd)

#calculate the minimum value of each cookie
minimum <- apply(X , 1, min)

#calculate the maximum value of each cookie
maximum <- apply(X , 1, max)

#calculate slope for each cookie
slope <- numeric(32)
x <- seq(1,700,1)
for (i in 1:32){
  mod <- lm(unlist(df[i, 2:701])~x)
slope[i] <- coef(mod)[2]
}
```

Full logistic regression

- Here we are preparing the features data by combining them with the binary target $YBin$.
- Since the observation is only 32, meaning we are facing an inadequate amount of data. Thus, splitting data into train and test dataset will not be a good approach. We will now train and test on the same data, and we will look into a k-fold procedure instead later on in order to make a more logical prediction where we make prediction many times on a new dataset.

```
#bind all the fretures with binary target YBin to get a dataframe
features_data <- data.frame(YBin, mean, sd, slope, minimum, maximum)

#every column except YBin is predictor, X
X_features <- subset(features_data, select = -YBin)

#assign YBin column
Y_features <- features_data$YBin

#visualize 6 first rows of the dataframe
head(features_data)
```

```
##   YBin      mean        sd       slope  minimum maximum
```

```
## 1    0 0.9851499 0.4111868 0.001914311 0.259270 1.73946
## 2    1 1.0355417 0.4123933 0.001898164 0.266864 1.66273
## 3    0 1.0010620 0.4025158 0.001860203 0.251654 1.60960
## 4    0 1.0280481 0.4040351 0.001861782 0.277777 1.63881
## 5    1 1.0655011 0.4158252 0.001910926 0.288328 1.70320
## 6    0 1.0840236 0.4262425 0.001967228 0.284625 1.74356
```

- By manipulate the multiple linear regression, we then have a logistic regression model with a logit function as follows:

$$\log(\frac{p(X)}{1-p(X)}) = \beta_0 + \beta_1 X_1 + ... + \beta_p X_p$$

$$\Rightarrow p(X) = \frac{e^{\beta_0+\beta_1 X_1+...+\beta_p X_p}}{1+e^{\beta_0+\beta_1 X_1+...+\beta_p X_p}}$$

- Where $X = (X_1, ..., X_p)$ are p predictors, and $p(X) = Pr(Y = 1/X = x) \in [0, 1]$

Estimation step

- Here, we fit the logistic model by using $glm()$ function where $family = binomial$ indicates the link function is the logit function.

```
#fit a full logistic model on features dataset
logit_mod <- glm(YBin~., family = binomial, data = features_data)
summary(logit_mod)
```

```
##
## Call:
## glm(formula = YBin ~ ., family = binomial, data = features_data)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.80802  -0.65396  -0.04465   0.67651   1.91688
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.565e+01  2.440e+01  -1.051   0.2931
## mean        -4.440e+01  6.208e+01  -0.715   0.4745
## sd           1.129e+03  6.271e+02   1.801   0.0717 .
## slope       -2.284e+05  1.186e+05  -1.925   0.0542 .
## minimum      1.371e+02  1.288e+02   1.065   0.2870
## maximum      1.894e+00  2.294e+01   0.083   0.9342
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 44.361  on 31  degrees of freedom
## Residual deviance: 26.927  on 26  degrees of freedom
## AIC: 38.927
##
## Number of Fisher Scoring iterations: 6
```

- Based on the summary, even though '0.1' at the end of *sd* and *slope* predictors have a small impact on response variable, they are the most significant ones in this case with the smallest p-value compared to others in which we reject H_0. We will see later if this is still the case for other approaches such as variable selection, etc.

Prediction step

- By using *predict.glm()* function with the argument $type = response$, the output will be the probabilities.

```
#estimation (computation) of the probability
probs <- predict.glm(logit_mod, type = "response")
```

Decision step

- In this step, by considering the MAP (Maximum A Posteriori) threshold choice S = 0.5, which means $\hat{Y} = 1$ if $p(X) > S$, and $\hat{Y} = 0$ otherwise.

```
#choose a threshold S, then make a binary decision based on the probab obtained
MAP_threshold <- 0.5
Y_hat <- ifelse(probs > MAP_threshold, 1, 0)
cat("Prediction: ", Y_hat)
```

```
## Prediction:  0 1 0 1 1 1 0 1 1 0 1 0 1 1 1 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0 1 0
```

- By using the function implemented above we have the overview of this model as below:

```
performance <- logistic_performance(Y_hat, features_data$YBin)
```

```
## Confusion matrix:
##
##           Observed
## Predicted  0  1
##         0 12  2
##         1  4 14
##
## Accuracy: 81.25 %
## Error rate:  18.75 %
## Recall:  87.5 %
## Specificity:  75 %
## Precision:  77.77778 %
## False positive rate: 25 %
## False negative rate: 12.5 %
```

Compute odd-ratio

- Odd-ratio is important as it is an indicator used to characterized the negative or positive influence of a co-variable on the binary target Y.

```r
OR <- exp(logit_mod$coefficients)
OR
```

```
##  (Intercept)         mean           sd        slope      minimum      maximum
## 7.256443e-12 5.214511e-20          Inf 0.000000e+00 3.578738e+59 6.644141e+00
```

- $OR < 1$ means a negative influence of X on Y, and $OR > 1$ means a positive influence.
- We can see only *mean* and *slope* are smaller than 1, whereas the others are greater than 1. Visibly, the OR of infinity (*sd*) can occur when there is perfect prediction or separation in the data. We will then see if it is the case in the variable selection section.

K-fold cross validation for logistic regression

- Since we have an insufficient amount of data, then splitting data will not be a good approach to evaluate this model, hence, we consider instead the k-fold procedure.

```r
#algo from prof

#set index to number of observations we have then sample it to have random order
ind = sample(nrow(features_data))

#assign a new tab equal to random order of features_data
tab = features_data[ind,]

#initialize number of folds
k = 5

#num of obs in order to find each block
n = nrow(features_data)

#divide into block based on obs-n and number of folds k
l_bloc = trunc(n/k)

# initialize the total performance metrics of all folds
tot_accuracy = 0;
tot_error = 0;
tot_recall = 0;
tot_specificity = 0;
tot_precision = 0;
tot_fpr = 0;
tot_fnr = 0;

#k-fold procedure
for (i in 1:k){

  #print order of fold
  cat("======= Fold", i, "========", "\n\n")

  #initialize index for i-fold, for example: i=1,n=32,k=5 will give index of 1-6
  ind_i = ((i-1)*l_bloc + 1) : (i*l_bloc)
```

```
  #initialize data for test and train
  tabTrain = tab[-ind_i,]
  tabTest = tab[ind_i,]

  #train generalized linear model
  modTrain = glm(YBin~., family = "binomial", data = tabTrain)

  #prediction and decision step for logistic regression
  Y_hat = predict.glm(modTrain, type = "response", newdata = tabTest)
  Y_hat = ifelse(Y_hat > MAP_threshold, 1, 0)

  #print performance for i-fold
  performance = logistic_performance(Y_hat, tabTest$YBin)
  cat("\n")

  #calculate total of the performance metrics for each fold
  tot_accuracy = tot_accuracy + performance$accuracy;
  tot_error = tot_error + performance$error_rate;
  tot_recall = tot_recall + performance$recall;
  tot_specificity = tot_specificity + performance$specificity;
  tot_precision = tot_precision + performance$precision;
  tot_fpr = tot_fpr + performance$FPR;
  tot_fnr = tot_fnr + performance$FNR;
}
```

```
## ======= Fold 1 ========
##
## Confusion matrix:
##
##           Observed
## Predicted 0 1
##         0 3 0
##         1 1 2
##
## Accuracy: 83.33333 %
## Error rate:  16.66667 %
## Recall:  100 %
## Specificity:  75 %
## Precision:  66.66667 %
## False positive rate: 25 %
## False negative rate: 0 %
##
## ======= Fold 2 ========
##
## Confusion matrix:
##
##           Observed
## Predicted 0 1
##         0 3 1
##         1 0 2
##
## Accuracy: 83.33333 %
## Error rate:  16.66667 %
```

```
## Recall:  66.66667 %
## Specificity:  100 %
## Precision:  100 %
## False positive rate: 0 %
## False negative rate: 33.33333 %
##
## ======= Fold 3 ========

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 1 3
##         1 0 2
##
## Accuracy: 50 %
## Error rate:  50 %
## Recall:  40 %
## Specificity:  100 %
## Precision:  100 %
## False positive rate: 0 %
## False negative rate: 60 %
##
## ======= Fold 4 ========
##
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 3 0
##         1 1 2
##
## Accuracy: 83.33333 %
## Error rate:  16.66667 %
## Recall:  100 %
## Specificity:  75 %
## Precision:  66.66667 %
## False positive rate: 25 %
## False negative rate: 0 %
##
## ======= Fold 5 ========
##
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 1 1
##         1 1 3
##
## Accuracy: 66.66667 %
## Error rate:  33.33333 %
## Recall:  75 %
```

```
## Specificity:  50 %
## Precision:  75 %
## False positive rate: 50 %
## False negative rate: 25 %
```

```
#calculate average of the performance
avg_accuracy = (tot_accuracy / k)
avg_error = (tot_error / k)
avg_recall = (tot_recall / k)
avg_specificity = (tot_specificity / k)
avg_precision = (tot_precision / k)
avg_fpr = (tot_fpr / k)
avg_fnr = (tot_fnr / k)

#print end fold
cat("======= End Fold =======\n\n")
```

```
## ======= End Fold =======
```

```
#print all metrics, average performance of the k-fold cross validation
cat("Average accuracy:", avg_accuracy*100, "%", "\n")
```

```
## Average accuracy: 73.33333 %
```

```
cat("Average error rate: ", avg_error*100, "%", "\n")
```

```
## Average error rate:  26.66667 %
```

```
cat("Average recall: ", avg_recall*100, "%", "\n")
```

```
## Average recall:  76.33333 %
```

```
cat("Average specificity: ", avg_specificity*100, "%", "\n")
```

```
## Average specificity:  80 %
```

```
cat("Average precision: ", avg_precision*100, "%", "\n")
```

```
## Average precision:  81.66667 %
```

```
cat("Average false positive rate:", avg_fpr*100, "%", "\n")
```

```
## Average false positive rate: 20 %
```

```
cat("Average false negative rate:", avg_fnr*100, "%", "\n")
```

```
## Average false negative rate: 23.66667 %
```

```
#save average performance values to a list for comparison
logit_avg_perf <- list(average_accuracy = avg_accuracy,
                       average_error = avg_error,
                       average_recall = avg_recall,
                       average_specificity = avg_specificity,
                       average_precision = avg_precision,
                       average_fpr = avg_fpr,
                       average_fnr = avg_fnr)
```

- From the output above, we can see that by initialize the number of fold to 5, and use the k-1 fold trained model on the unseen test data, the accuracy dropped in which it is more reasonable and at the same time reliable since we tested many times on the new data. We will then consider this performance rather than the previous one.

Logistic regression with variable selection

Forward selection

- Since the null (M_0) will give all 0 as the output, it will not be taken into account. We will now look into the variable selection method. By using *step*() function without defining k in its arguments, this model will follow the AIC criterion.

```
res0 <- glm(YBin~1, data = features_data, family = binomial);
resFor <- step(res0, list(upper=logit_mod), direction='forward')
```

```
## Start:  AIC=46.36
## YBin ~ 1
##
##           Df Deviance    AIC
## + minimum  1   32.105 36.105
## + mean     1   33.321 37.321
## + sd       1   38.981 42.981
## + slope    1   39.838 43.838
## + maximum  1   41.004 45.004
## <none>         44.361 46.361
##
## Step:  AIC=36.1
## YBin ~ minimum
##
##           Df Deviance    AIC
## <none>         32.105 36.105
## + mean     1   32.067 38.067
## + maximum  1   32.076 38.076
## + sd       1   32.078 38.078
## + slope    1   32.103 38.103
```

```
formula(resFor)
```

```
## YBin ~ minimum
```

- From the formula, we can see that this model eliminated all the variables except for the *minimum*. We now will see its performance below.

```
probsFor <- predict.glm(resFor, type = "response")
Y_hat_for <- ifelse(probsFor > MAP_threshold, 1, 0)
perf_for <- logistic_performance(Y_hat_for, features_data$YBin)
```

```
## Confusion matrix:
##
##           Observed
## Predicted  0  1
##         0 11  5
##         1  5 11
##
## Accuracy: 68.75 %
## Error rate:  31.25 %
## Recall:  68.75 %
## Specificity:  68.75 %
## Precision:  68.75 %
## False positive rate: 31.25 %
## False negative rate: 31.25 %
```

- The accuracy obtained is lower than what we had before by doing multiple logistic regression.

K-fold cross validation for forward selection

- We will now perform k-fold cross validation by using forward selection approach to see whether it is better than the model we previously obtained.

```
#set index to number of observations we have then sample it to have random order
ind = sample(nrow(features_data))

#assign a new tab equal to random order of features_data
tab = features_data[ind,]

#initialize number of fold
k = 5

#initialize number of obs
n = nrow(features_data)

#divide into block based on obs-n and number of folds k
l_bloc = trunc(n/k)

# initialize the total performance metrics of all folds
tot_accuracy = 0;
tot_error = 0;
tot_recall = 0;
tot_specificity = 0;
tot_precision = 0;
tot_fpr = 0;
tot_fnr = 0;

#k-fold process
for (i in 1:k){
```

```
  #print order of fold
  cat("======= Fold", i, "========", "\n\n")

  #initialize index for i-fold, for example: i=1,n=32,k=5 will give index of 1-6
  ind_i = ((i-1)*l_bloc + 1) : (i*l_bloc)

  #initialize data for test and train
  tabTrain = tab[-ind_i,]
  tabTest = tab[ind_i,]

  #train forward selection model
  modTrain = step(res0, list(upper=logit_mod), direction='forward')

  #prediction and decision step
  Y_hat = predict.glm(modTrain, type = "response", newdata = tabTest)
  Y_hat = ifelse(Y_hat > MAP_threshold, 1, 0)

  #find performance of this model
  performance = logistic_performance(Y_hat, tabTest$YBin)
  cat("\n")

  #calculate total accuracy for each fold
  tot_accuracy = tot_accuracy + performance$accuracy;
  tot_error = tot_error + performance$error_rate;
  tot_recall = tot_recall + performance$recall;
  tot_specificity = tot_specificity + performance$specificity;
  tot_precision = tot_precision + performance$precision;
  tot_fpr = tot_fpr + performance$FPR;
  tot_fnr = tot_fnr + performance$FNR;
}
```

```
## ======= Fold 1 ========
##
## Start:  AIC=46.36
## YBin ~ 1
##
##           Df Deviance    AIC
## + minimum  1   32.105 36.105
## + mean     1   33.321 37.321
## + sd       1   38.981 42.981
## + slope    1   39.838 43.838
## + maximum  1   41.004 45.004
## <none>         44.361 46.361
##
## Step:  AIC=36.1
## YBin ~ minimum
##
##           Df Deviance    AIC
## <none>         32.105 36.105
## + mean     1   32.067 38.067
## + maximum  1   32.076 38.076
## + sd       1   32.078 38.078
## + slope    1   32.103 38.103
```

```
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 4 0
##         1 0 2
##
## Accuracy: 100 %
## Error rate:  0 %
## Recall:  100 %
## Specificity:  100 %
## Precision:  100 %
## False positive rate: 0 %
## False negative rate: 0 %
##
## ======= Fold 2 ========
##
## Start:  AIC=46.36
## YBin ~ 1
##
##           Df Deviance    AIC
## + minimum  1   32.105 36.105
## + mean     1   33.321 37.321
## + sd       1   38.981 42.981
## + slope    1   39.838 43.838
## + maximum  1   41.004 45.004
## <none>         44.361 46.361
##
## Step:  AIC=36.1
## YBin ~ minimum
##
##           Df Deviance    AIC
## <none>         32.105 36.105
## + mean     1   32.067 38.067
## + maximum  1   32.076 38.076
## + sd       1   32.078 38.078
## + slope    1   32.103 38.103
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 1 1
##         1 1 3
##
## Accuracy: 66.66667 %
## Error rate:  33.33333 %
## Recall:  75 %
## Specificity:  50 %
## Precision:  75 %
## False positive rate: 50 %
## False negative rate: 25 %
##
## ======= Fold 3 ========
##
```

```
## Start:  AIC=46.36
## YBin ~ 1
##
##           Df Deviance    AIC
## + minimum  1   32.105 36.105
## + mean     1   33.321 37.321
## + sd       1   38.981 42.981
## + slope    1   39.838 43.838
## + maximum  1   41.004 45.004
## <none>         44.361 46.361
##
## Step:  AIC=36.1
## YBin ~ minimum
##
##           Df Deviance    AIC
## <none>         32.105 36.105
## + mean     1   32.067 38.067
## + maximum  1   32.076 38.076
## + sd       1   32.078 38.078
## + slope    1   32.103 38.103
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 3 0
##         1 1 2
##
## Accuracy: 83.33333 %
## Error rate:  16.66667 %
## Recall:  100 %
## Specificity:  75 %
## Precision:  66.66667 %
## False positive rate: 25 %
## False negative rate: 0 %
##
## ======= Fold 4 ========
##
## Start:  AIC=46.36
## YBin ~ 1
##
##           Df Deviance    AIC
## + minimum  1   32.105 36.105
## + mean     1   33.321 37.321
## + sd       1   38.981 42.981
## + slope    1   39.838 43.838
## + maximum  1   41.004 45.004
## <none>         44.361 46.361
##
## Step:  AIC=36.1
## YBin ~ minimum
##
##           Df Deviance    AIC
## <none>         32.105 36.105
## + mean     1   32.067 38.067
```

```
## + maximum  1   32.076 38.076
## + sd       1   32.078 38.078
## + slope    1   32.103 38.103
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 0 2
##         1 2 2
##
## Accuracy: 33.33333 %
## Error rate:  66.66667 %
## Recall:  50 %
## Specificity:  0 %
## Precision:  50 %
## False positive rate: 100 %
## False negative rate: 50 %
##
## ======= Fold 5 ========
##
## Start:  AIC=46.36
## YBin ~ 1
##
##           Df Deviance    AIC
## + minimum  1   32.105 36.105
## + mean     1   33.321 37.321
## + sd       1   38.981 42.981
## + slope    1   39.838 43.838
## + maximum  1   41.004 45.004
## <none>         44.361 46.361
##
## Step:  AIC=36.1
## YBin ~ minimum
##
##           Df Deviance    AIC
## <none>         32.105 36.105
## + mean     1   32.067 38.067
## + maximum  1   32.076 38.076
## + sd       1   32.078 38.078
## + slope    1   32.103 38.103
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 3 2
##         1 0 1
##
## Accuracy: 66.66667 %
## Error rate:  33.33333 %
## Recall:  33.33333 %
## Specificity:  100 %
## Precision:  100 %
## False positive rate: 0 %
## False negative rate: 66.66667 %
```

```r
#find average performance of this model
avg_accuracy = (tot_accuracy / k)
avg_error = (tot_error / k)
avg_recall = (tot_recall / k)
avg_specificity = (tot_specificity / k)
avg_precision = (tot_precision / k)
avg_fpr = (tot_fpr / k)
avg_fnr = (tot_fnr / k)

#print end fold
cat("======= End Fold =======\n\n")
```

```
## ======= End Fold =======
```

```r
#display the average performance of k-fold
cat("Average accuracy:", avg_accuracy*100, "%", "\n")
```

```
## Average accuracy: 70 %
```

```r
cat("Average error rate: ", avg_error*100, "%", "\n")
```

```
## Average error rate:  30 %
```

```r
cat("Average recall: ", avg_recall*100, "%", "\n")
```

```
## Average recall:  71.66667 %
```

```r
cat("Average specificity: ", avg_specificity*100, "%", "\n")
```

```
## Average specificity:  65 %
```

```r
cat("Average precision: ", avg_precision*100, "%", "\n")
```

```
## Average precision:  78.33333 %
```

```r
cat("Average false positive rate:", avg_fpr*100, "%", "\n")
```

```
## Average false positive rate: 35 %
```

```r
cat("Average false negative rate:", avg_fnr*100, "%", "\n")
```

```
## Average false negative rate: 28.33333 %
```

```r
#save average performance values to a list for comparison
logit_avg_for <- list(average_accuracy = avg_accuracy,
                      average_error = avg_error,
                      average_recall = avg_recall,
                      average_specificity = avg_specificity,
                      average_precision = avg_precision,
                      average_fpr = avg_fpr,
                      average_fnr = avg_fnr)
```

- The accuracy is surprisingly higher than the one without using k-fold process. We will now see the backward selection.

Backward selection

- Now we perform backward selection to see if it is a better model than forward, and to see as well the variables remaining.

```
resBack <- step(logit_mod, direction='backward')
```

```
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##         Df Deviance    AIC
## <none>       28.422 34.422
## - slope  1   38.981 42.981
## - sd     1   39.838 43.838
```

```
formula(resBack)
```

```
## YBin ~ sd + slope
```

- This model neglects the others except *sd* and *slope*.

```r
probsBack <- predict.glm(resBack, type = "response")
Y_hat_back <- ifelse(probsBack > MAP_threshold, 1, 0)
perf_back <- logistic_performance(Y_hat_back, features_data$YBin)
```

```
## Confusion matrix:
##
##           Observed
## Predicted  0  1
##         0 12  3
##         1  4 13
##
## Accuracy: 78.125 %
## Error rate:  21.875 %
## Recall:  81.25 %
## Specificity:  75 %
## Precision:  76.47059 %
## False positive rate: 25 %
## False negative rate: 18.75 %
```

K-fold cross validtion for backward

- We then perform k-fold cross-validation in order to get unbiased accuracy (average).

```r
#set index to number of observations we have then sample it to have random order
ind = sample(nrow(features_data))

#assign a new tab equal to random order of features_data
tab = features_data[ind,]

#initialize number of fold
k = 5

#num of obs in order to find each block
n = nrow(features_data)

#divide into block based on obs-n and number of folds k
l_bloc = trunc(n/k)

# initialize the total performance metrics of all folds
tot_accuracy = 0;
tot_error = 0;
tot_recall = 0;
tot_specificity = 0;
tot_precision = 0;
tot_fpr = 0;
tot_fnr = 0;

#k-fold procedure
for (i in 1:k){
  #print order of fold
  cat("======= Fold", i, "========", "\n\n")
```

```
  #initialize index for i-fold, for example: i=1,n=32,k=5 will give index of 1-6
  ind_i = ((i-1)*l_bloc + 1) : (i*l_bloc)

  #initialize data for test and train
  tabTrain = tab[-ind_i,]
  tabTest = tab[ind_i,]

  #train stepwise selection
  modTrain = step(logit_mod, direction='backward')

  #prediction and decision step for logistic regression
  Y_hat = predict.glm(modTrain, type = "response", newdata = tabTest)
  Y_hat = ifelse(Y_hat > MAP_threshold, 1, 0)

  #find performance for this model
  performance = logistic_performance(Y_hat, tabTest$YBin)
  cat("\n")

  #sum up all metrics for each fold to find total
  tot_accuracy = tot_accuracy + performance$accuracy;
  tot_error = tot_error + performance$error_rate;
  tot_recall = tot_recall + performance$recall;
  tot_specificity = tot_specificity + performance$specificity;
  tot_precision = tot_precision + performance$precision;
  tot_fpr = tot_fpr + performance$FPR;
  tot_fnr = tot_fnr + performance$FNR;
}
```

```
## ======= Fold 1 ========
##
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
```

```
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##         Df Deviance    AIC
## <none>       28.422 34.422
## - slope  1   38.981 42.981
## - sd     1   39.838 43.838
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 1 1
##         1 1 3
##
## Accuracy: 66.66667 %
## Error rate:  33.33333 %
## Recall:  75 %
## Specificity:  50 %
## Precision:  75 %
## False positive rate: 50 %
## False negative rate: 25 %
##
## ======= Fold 2 ========
##
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
```

```
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##         Df Deviance    AIC
## <none>       28.422 34.422
## - slope  1   38.981 42.981
## - sd     1   39.838 43.838
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 3 1
##         1 0 2
##
## Accuracy: 83.33333 %
## Error rate:  16.66667 %
## Recall:  66.66667 %
## Specificity:  100 %
## Precision:  100 %
## False positive rate: 0 %
## False negative rate: 33.33333 %
##
## ======= Fold 3 ========
##
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
```

```
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##         Df Deviance    AIC
## <none>       28.422 34.422
## - slope  1   38.981 42.981
## - sd     1   39.838 43.838
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 3 0
##         1 2 1
##
## Accuracy: 66.66667 %
## Error rate:  33.33333 %
## Recall:  100 %
## Specificity:  60 %
## Precision:  33.33333 %
## False positive rate: 40 %
## False negative rate: 0 %
##
## ======= Fold 4 ========
##
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
```

```
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##         Df Deviance    AIC
## <none>       28.422 34.422
## - slope  1   38.981 42.981
## - sd     1   39.838 43.838
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 2 1
##         1 1 2
##
## Accuracy: 66.66667 %
## Error rate:  33.33333 %
## Recall:  66.66667 %
## Specificity:  66.66667 %
## Precision:  66.66667 %
## False positive rate: 33.33333 %
## False negative rate: 33.33333 %
##
## ======= Fold 5 ========
##
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
```

```
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##         Df Deviance    AIC
## <none>       28.422 34.422
## - slope  1   38.981 42.981
## - sd     1   39.838 43.838
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 2 0
##         1 0 4
##
## Accuracy: 100 %
## Error rate:  0 %
## Recall:  100 %
## Specificity:  100 %
## Precision:  100 %
## False positive rate: 0 %
## False negative rate: 0 %
```

```
#find average performance
avg_accuracy = (tot_accuracy / k)
avg_error = (tot_error / k)
avg_recall = (tot_recall / k)
avg_specificity = (tot_specificity / k)
avg_precision = (tot_precision / k)
avg_fpr = (tot_fpr / k)
avg_fnr = (tot_fnr / k)


#print end fold
cat("======= End Fold =======\n\n")
```

```
## ======= End Fold =======
```

```
cat("Average accuracy:", avg_accuracy*100, "%", "\n")
```

```
## Average accuracy: 76.66667 %
```

```
cat("Average error rate: ", avg_error*100, "%", "\n")
```

```
## Average error rate:  23.33333 %
```

```
cat("Average recall: ", avg_recall*100, "%", "\n")
```

```
## Average recall:  81.66667 %
```

```
cat("Average specificity: ", avg_specificity*100, "%", "\n")
```

```
## Average specificity:  75.33333 %
```

```
cat("Average precision: ", avg_precision*100, "%", "\n")
```

```
## Average precision:  75 %
```

```
cat("Average false positive rate:", avg_fpr*100, "%", "\n")
```

```
## Average false positive rate: 24.66667 %
```

```
cat("Average false negative rate:", avg_fnr*100, "%", "\n")
```

```
## Average false negative rate: 18.33333 %
```

```
#save average performance values to a list for comparison
logit_avg_back <- list(average_accuracy = avg_accuracy,
                                  average_error = avg_error,
                                  average_recall = avg_recall,
                                  average_specificity = avg_specificity,
                                  average_precision = avg_precision,
                                  average_fpr = avg_fpr,
                                  average_fnr = avg_fnr)
```

Stepwise selection

- Now we perform stepwise selection to see if it is a better model than forward and backward, and also to see the variables remaining.

```
resStep <- step(logit_mod, direction='both')
```

```
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
```

```
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## + maximum  1   26.927 38.927
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## + mean     1   26.934 36.934
## + maximum  1   27.452 37.452
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##           Df Deviance    AIC
## <none>         28.422 34.422
## + minimum  1   27.804 35.804
## + mean     1   28.321 36.321
## + maximum  1   28.421 36.421
## - slope    1   38.981 42.981
## - sd       1   39.838 43.838
```

```
formula(resStep)
```

```
## YBin ~ sd + slope
```

- It remains the same variables *sd*, and *slope* as the backward.

```
probsStep <- predict.glm(resStep, type = "response")
Y_hat_step <- ifelse(probsStep > MAP_threshold, 1, 0)
perf_step <- logistic_performance(Y_hat_step, features_data$YBin)
```

```
## Confusion matrix:
##
##           Observed
## Predicted  0  1
##         0 12  3
##         1  4 13
##
## Accuracy: 78.125 %
## Error rate:  21.875 %
## Recall:  81.25 %
```

```
## Specificity:  75 %
## Precision:  76.47059 %
## False positive rate: 25 %
## False negative rate: 18.75 %
```

- Here we can see the performance of this model that it is the same as backward since they prioritize the same predictors.

K-fold cross validtion for stepwise

- We then perform k-fold and since the selection of variable is the same between backward and stepwise selection, the accuracy of both are bound to be the same.

```
#set index to number of observations we have then sample it to have random order
ind = sample(nrow(features_data))

#assign a new tab equal to random order of features_data
tab = features_data[ind,]

#initialize number of fold
k = 5

#num of obs in order to find each block
n = nrow(features_data)

#divide into block based on obs-n and number of folds k
l_bloc = trunc(n/k)

# initialize the total performance metrics of all folds
tot_accuracy = 0;
tot_error = 0;
tot_recall = 0;
tot_specificity = 0;
tot_precision = 0;
tot_fpr = 0;
tot_fnr = 0;

#k-fold procedure
for (i in 1:k){
  #print order of fold
  cat("======= Fold", i, "========", "\n\n")

  #initialize index for i-fold, for example: i=1,n=32,k=5 will give index of 1-6
  ind_i = ((i-1)*l_bloc + 1) : (i*l_bloc)

  #initialize data for test and train
  tabTrain = tab[-ind_i,]
  tabTest = tab[ind_i,]

  #train stepwise selection
  modTrain = step(logit_mod, direction='both')

  #prediction and decision step for logistic regression
```

```
  Y_hat = predict.glm(modTrain, type = "response", newdata = tabTest)
  Y_hat = ifelse(Y_hat > MAP_threshold, 1, 0)

  #find performance for this model
  performance = logistic_performance(Y_hat, tabTest$YBin)
  cat("\n")

  #sum up all metrics for each fold to find total
  tot_accuracy = tot_accuracy + performance$accuracy;
  tot_error = tot_error + performance$error_rate;
  tot_recall = tot_recall + performance$recall;
  tot_specificity = tot_specificity + performance$specificity;
  tot_precision = tot_precision + performance$precision;
  tot_fpr = tot_fpr + performance$FPR;
  tot_fnr = tot_fnr + performance$FNR;
}
```

```
## ======= Fold 1 ========
##
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## + maximum  1   26.927 38.927
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## + mean     1   26.934 36.934
## + maximum  1   27.452 37.452
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
```

```
## YBin ~ sd + slope
##
##           Df Deviance    AIC
## <none>         28.422 34.422
## + minimum  1   27.804 35.804
## + mean     1   28.321 36.321
## + maximum  1   28.421 36.421
## - slope    1   38.981 42.981
## - sd       1   39.838 43.838
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 3 0
##         1 1 2
##
## Accuracy: 83.33333 %
## Error rate:  16.66667 %
## Recall:  100 %
## Specificity:  75 %
## Precision:  66.66667 %
## False positive rate: 25 %
## False negative rate: 0 %
##
## ======= Fold 2 ========
##
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## + maximum  1   26.927 38.927
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
```

```
## + mean     1   26.934 36.934
## + maximum  1   27.452 37.452
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##           Df Deviance    AIC
## <none>         28.422 34.422
## + minimum  1   27.804 35.804
## + mean     1   28.321 36.321
## + maximum  1   28.421 36.421
## - slope    1   38.981 42.981
## - sd       1   39.838 43.838
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 3 1
##         1 0 2
##
## Accuracy: 83.33333 %
## Error rate:  16.66667 %
## Recall:  66.66667 %
## Specificity:  100 %
## Precision:  100 %
## False positive rate: 0 %
## False negative rate: 33.33333 %
##
## ======= Fold 3 ========
##
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## + maximum  1   26.927 38.927
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
```

```
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## + mean     1   26.934 36.934
## + maximum  1   27.452 37.452
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##           Df Deviance    AIC
## <none>         28.422 34.422
## + minimum  1   27.804 35.804
## + mean     1   28.321 36.321
## + maximum  1   28.421 36.421
## - slope    1   38.981 42.981
## - sd       1   39.838 43.838
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 4 0
##         1 0 2
##
## Accuracy: 100 %
## Error rate:  0 %
## Recall:  100 %
## Specificity:  100 %
## Precision:  100 %
## False positive rate: 0 %
## False negative rate: 0 %
##
## ======= Fold 4 ========
##
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
```

```
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## + maximum  1   26.927 38.927
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## + mean     1   26.934 36.934
## + maximum  1   27.452 37.452
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##           Df Deviance    AIC
## <none>         28.422 34.422
## + minimum  1   27.804 35.804
## + mean     1   28.321 36.321
## + maximum  1   28.421 36.421
## - slope    1   38.981 42.981
## - sd       1   39.838 43.838
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 0 2
##         1 2 2
##
## Accuracy: 33.33333 %
## Error rate:  66.66667 %
## Recall:  50 %
## Specificity:  0 %
## Precision:  50 %
## False positive rate: 100 %
## False negative rate: 50 %
##
## ======= Fold 5 ========
##
## Start:  AIC=38.93
## YBin ~ mean + sd + slope + minimum + maximum
##
##           Df Deviance    AIC
## - maximum  1   26.934 36.934
## - mean     1   27.452 37.452
## - minimum  1   28.213 38.213
## <none>         26.927 38.927
## - sd       1   31.095 41.095
## - slope    1   32.044 42.044
```

```
##
## Step:  AIC=36.93
## YBin ~ mean + sd + slope + minimum
##
##           Df Deviance    AIC
## - mean     1   27.804 35.804
## - minimum  1   28.321 36.321
## <none>         26.934 36.934
## + maximum  1   26.927 38.927
## - sd       1   32.019 40.019
## - slope    1   32.067 40.067
##
## Step:  AIC=35.8
## YBin ~ sd + slope + minimum
##
##           Df Deviance    AIC
## - minimum  1   28.422 34.422
## <none>         27.804 35.804
## + mean     1   26.934 36.934
## + maximum  1   27.452 37.452
## - slope    1   32.078 38.078
## - sd       1   32.103 38.103
##
## Step:  AIC=34.42
## YBin ~ sd + slope
##
##           Df Deviance    AIC
## <none>         28.422 34.422
## + minimum  1   27.804 35.804
## + mean     1   28.321 36.321
## + maximum  1   28.421 36.421
## - slope    1   38.981 42.981
## - sd       1   39.838 43.838
## Confusion matrix:
##
##          Observed
## Predicted 0 1
##         0 1 0
##         1 1 4
##
## Accuracy: 83.33333 %
## Error rate:  16.66667 %
## Recall:  100 %
## Specificity:  50 %
## Precision:  80 %
## False positive rate: 50 %
## False negative rate: 0 %
```

```
#find average performance
avg_accuracy = (tot_accuracy / k)
avg_error = (tot_error / k)
avg_recall = (tot_recall / k)
avg_specificity = (tot_specificity / k)
avg_precision = (tot_precision / k)
```

```
avg_fpr = (tot_fpr / k)
avg_fnr = (tot_fnr / k)


#print end fold
cat("======= End Fold =======\n\n")
```

```
## ======= End Fold =======
```

```
cat("Average accuracy:", avg_accuracy*100, "%", "\n")
```

```
## Average accuracy: 76.66667 %
```

```
cat("Average error rate: ", avg_error*100, "%", "\n")
```

```
## Average error rate:  23.33333 %
```

```
cat("Average recall: ", avg_recall*100, "%", "\n")
```

```
## Average recall:  83.33333 %
```

```
cat("Average specificity: ", avg_specificity*100, "%", "\n")
```

```
## Average specificity:  65 %
```

```
cat("Average precision: ", avg_precision*100, "%", "\n")
```

```
## Average precision:  79.33333 %
```

```
cat("Average false positive rate:", avg_fpr*100, "%", "\n")
```

```
## Average false positive rate: 35 %
```

```
cat("Average false negative rate:", avg_fnr*100, "%", "\n")
```

```
## Average false negative rate: 16.66667 %
```

```
#save average performance values to a list for comparison
logit_avg_step <- list(average_accuracy = avg_accuracy,
                                average_error = avg_error,
                                average_recall = avg_recall,
                                average_specificity = avg_specificity,
                                average_precision = avg_precision,
                                average_fpr = avg_fpr,
                                average_fnr = avg_fnr)
```

Conclusion

```
data.frame(
  Model=c("Full Logistic", "Forward", "Backward", "Stepwise"),
  Accuracy=c(logit_avg_perf$average_accuracy,
             logit_avg_for$average_accuracy,
             logit_avg_back$average_accuracy,
             logit_avg_step$average_accuracy),
  Error_rate=c(logit_avg_perf$average_error,
               logit_avg_for$average_error,
               logit_avg_back$average_error,
               logit_avg_step$average_error),
  Recall=c(logit_avg_perf$average_recall,
           logit_avg_for$average_recall,
           logit_avg_back$average_recall,
           logit_avg_step$average_recall),
  Specificity=c(logit_avg_perf$average_specificity,
                logit_avg_for$average_specificity,
                logit_avg_back$average_specificity,
                logit_avg_step$average_specificity),
  Precision=c(logit_avg_perf$average_precision,
              logit_avg_for$average_precision,
              logit_avg_back$average_precision,
              logit_avg_step$average_precision),
  False_Positive_Rate=c(logit_avg_perf$average_fpr,
                        logit_avg_for$average_fpr,
                        logit_avg_back$average_fpr,
                        logit_avg_step$average_fpr),
  False_Negative_Rate=c(logit_avg_perf$average_fnr,
                        logit_avg_for$average_fnr,
                        logit_avg_back$average_fnr,
                        logit_avg_step$average_fnr)
)
```

```
##           Model  Accuracy Error_rate    Recall Specificity Precision
## 1 Full Logistic 0.7333333  0.2666667 0.7633333   0.8000000 0.8166667
## 2       Forward 0.7000000  0.3000000 0.7166667   0.6500000 0.7833333
## 3      Backward 0.7666667  0.2333333 0.8166667   0.7533333 0.7500000
## 4      Stepwise 0.7666667  0.2333333 0.8333333   0.6500000 0.7933333
##   False_Positive_Rate False_Negative_Rate
## 1           0.2000000           0.2366667
## 2           0.3500000           0.2833333
## 3           0.2466667           0.1833333
## 4           0.3500000           0.1666667
```

- Based on the table above, we can see that backward and stepwise logistic regression are interchangeable in terms of the performance metrics that provide insights into the models' behavior. Hence, in the final analysis, backward and stepwise selection are a better model than full logistic and forward selection in order to describe cookies dataset in which we are desire to explain if the fat is higher than the median value of all the fat indicators given the previous computed features.

Logistic regression model using the spectra

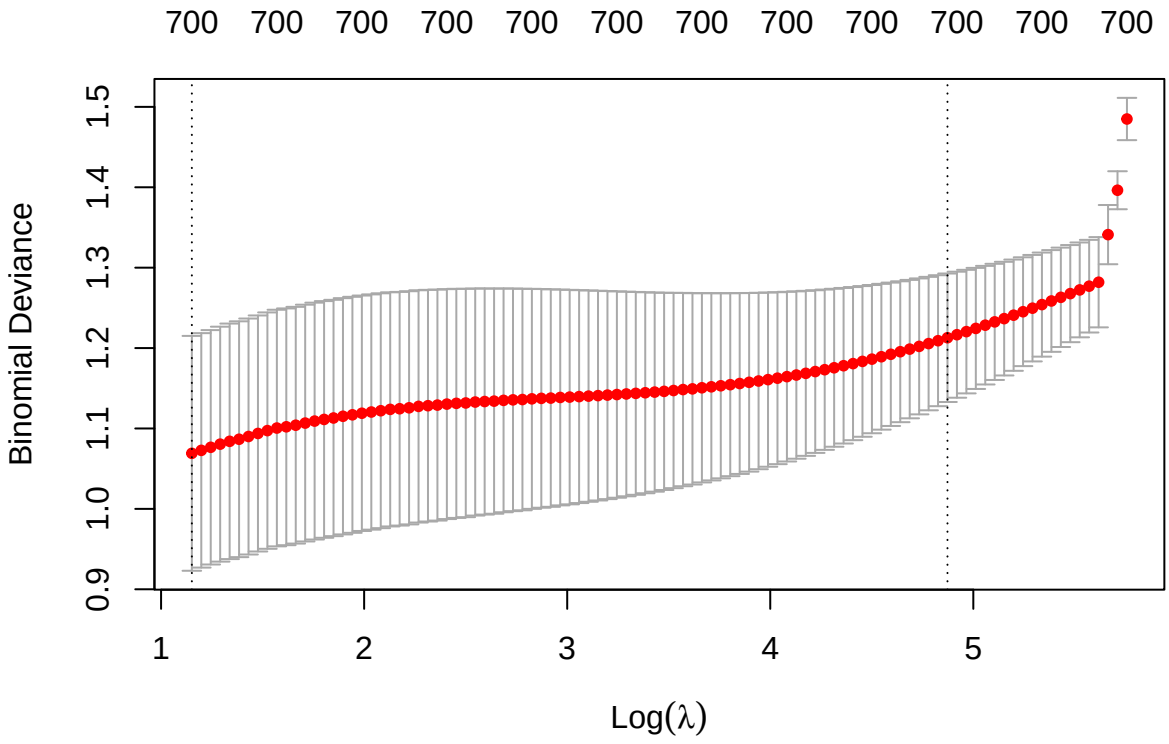
- In this section we will study the l1 and l2 penalized logistic regression in which it involves adding a penalty term to the logistic regression objective function to prevent overfitting in order to explain the low or high values of the fat given the spectra since a full logistic will have a 100% accuracy.
- First we bind the $YBin$ obtained from above, then remove the response variable fat from the data frame since we want to predict the binary target variable.

```
spectra_data <- cbind(YBin, df)
spectra_data <- subset(spectra_data, select = -fat)
X_spec <- subset(spectra_data, select = -YBin)
Y_spec <- spectra_data$YBin
```

Penalized Logistic Regression: Ridge

- First, we perform a cross-validation using $cv.glmnet()$ to find the best tuning parameter λ.

```
cv_ridge <- cv.glmnet(as.matrix(X_spec), Y_spec, alpha = 0, family = "binomial")
plot(cv_ridge)
```



- The plot displays the cross-validation error according to the log of lambda. The steady vertical line indicates that the log of the optimal value of lambda is approximately 3, which is the one that minimizes the prediction error. This lambda value will give the most accurate model. The exact value of lambda can be viewed as follow:

```
cv_ridge$lambda.min
```

```
## [1] 3.162763
```

- The function *cv.glmnet*() finds also the value of lambda that gives the simplest model but also lies within one standard error of the optimal value of lambda. It is is called lambda.1se.

```
cv_ridge$lambda.1se
```

```
## [1] 130.6857
```

- We now fit the final ridge model by using 2 different lambda values from above, min and 1se.

Estimation step

- Here, family is the response type, so "binomial" for a binary outcome variable, and alpha is the elastic net mixing parameter where "1": for lasso regression, and "0": for ridge regression.

```
ridgeMin <- glmnet(as.matrix(X_spec), Y_spec, alpha = 0,
                   family = "binomial", lambda = cv_ridge$lambda.min)
ridge1se <- glmnet(as.matrix(X_spec), Y_spec, alpha = 0,
                   family = "binomial", lambda = cv_ridge$lambda.1se)
```

Prediction step

- Find probabilities for both model which using λ_{min}, and λ_{1se}.

```
probs_ridgeMin <- predict(ridgeMin, s = cv_ridge$lambda.min,
                          newx = as.matrix(X_spec), type = "response")
probs_ridge1se <- predict(ridge1se, s = cv_ridge$lambda.1se,
                          newx = as.matrix(X_spec), type = "response")
```

Decision step

- Make decision for both model based on probabilities obtained.

```
Y_hat_ridgeMin <- ifelse(probs_ridgeMin > MAP_threshold, 1, 0)
Y_hat_ridge1se <- ifelse(probs_ridge1se > MAP_threshold, 1, 0)
```

Model performance

- Here we find which one is better by finding their error.

```
MSE(spectra_data$YBin, Y_hat_ridgeMin)
```

```
## [1] 0.21875
```

```
MSE(spectra_data$YBin, Y_hat_ridge1se)
```

```
## [1] 0.25
```

- Setting $\lambda = \lambda_{1se}$ produces a simpler model compared to λ_{min}, nevertheless, the model might be a little bit less accurate than the one obtained with λ_{min} since its error is smaller.

K-fold cross validation for l2 penalized logistic regression: Ridge

- Now we perform k-fold cross validation for l2 penalized logistic regression by choosing λ_{min} as the best lambda.

```
#set index to number of observations we have then sample it to have random order
ind = sample(nrow(spectra_data))

#assign a new tab equal to random order of spectra
tab = spectra_data[ind,]

#initialize number of fold
k = 5

#num of obs in order to find each block
n = nrow(spectra_data)

#divide into block based on obs-n and number of folds k
l_bloc = trunc(n/k)

#initlialize the total of model performance
total_MSE = 0
total_accu = 0

#k-fold procedure
for (i in 1:k){
  #print order of fold
  cat("======= Fold", i, "========", "\n\n")

  #initialize index for i-fold, for example: i=1,n=32,k=5 will give index of 1-6
  ind_i = ((i-1)*l_bloc + 1) : (i*l_bloc)

  #initialize data for test and train
  tabTrain = tab[-ind_i,]
  tabTest = tab[ind_i,]

  #initialize predictor and response variable for train, test
  X_train = subset(tabTrain, select = -YBin)
  Y_train = tabTrain$YBin
  X_test = subset(tabTest, select = -YBin)
  Y_test = tabTest$YBin

  #perform cross-validation to find best lambda
  cv_ridge <- cv.glmnet(as.matrix(X_train), Y_train, alpha = 0, family = "binomial", grouped=FALSE)
```

```
  #train penalized ridge model using best lambda obtained from cv
  modTrain = glmnet(as.matrix(X_train), Y_train, alpha = 0,
                    family = "binomial", lambda = cv_ridge$lambda.min)

  #find probability and make decision for this model (prediction and decision step)
  probs = predict(modTrain, s = cv_ridge$lambda.min, newx = as.matrix(X_test),
                  type = "response")
  Y_hat = ifelse(probs > MAP_threshold, 1, 0)

  #find error and accuracy
  mse = MSE(Y_test, Y_hat)
  accu = mean(Y_test == Y_hat)

  #sum up each fold to find total error and accuracy
  total_MSE = total_MSE + mse
  total_accu = total_accu + accu

  #display performance
  cat("Lambda selected for Fold", i, ":", cv_ridge$lambda.min, "\n")
  cat("MSE for Fold", i, ":", mse, "\n")
  cat("Accuracy for Fold", i, ":", accu, "\n\n")
}
```

```
## ======= Fold 1 ========
##
## Lambda selected for Fold 1 : 2.954799
## MSE for Fold 1 : 0
## Accuracy for Fold 1 : 1
##
## ======= Fold 2 ========
##
## Lambda selected for Fold 2 : 3.140997
## MSE for Fold 2 : 0.3333333
## Accuracy for Fold 2 : 0.6666667
##
## ======= Fold 3 ========
##
## Lambda selected for Fold 3 : 3.229501
## MSE for Fold 3 : 0.3333333
## Accuracy for Fold 3 : 0.6666667
##
## ======= Fold 4 ========
##
## Lambda selected for Fold 4 : 3.036183
## MSE for Fold 4 : 0.1666667
## Accuracy for Fold 4 : 0.8333333
##
## ======= Fold 5 ========
##
## Lambda selected for Fold 5 : 3.342359
## MSE for Fold 5 : 0.3333333
## Accuracy for Fold 5 : 0.6666667
```

```
#find average performance for this model
avg_mse = total_MSE/k
avg_accu = total_accu/k

#print end fold
cat("======= End Fold =======\n\n")
```

```
## ======= End Fold =======
```

```
#display the average performance
cat("Average MSE: ", avg_mse, "\n")
```

```
## Average MSE:  0.2333333
```

```
cat("Average accuracy: ", avg_accu, "\n")
```

```
## Average accuracy:  0.7666667
```

```
ridge_min_perf <- list(MSE = avg_mse, accuracy = avg_accu)
```

- By performing the k-fold cross validation the error is slightly went up, and it is logical since each time it tested on the unseen data.
- Now we perform k-fold cross validation for l2 penalized logistic regression by choosing λ_{1se} as the best lambda.

```
#set index to number of observations we have then sample it to have random order
ind = sample(nrow(spectra_data))

#assign a new tab equal to random order of spectra
tab = spectra_data[ind,]

#initialize number of fold
k = 5

#num of obs in order to find each block
n = nrow(spectra_data)

#divide into block based on obs-n and number of folds k
l_bloc = trunc(n/k)

#initlialize the total of model performance
total_MSE = 0
total_accu = 0

#k-fold procedure
for (i in 1:k){
  #print order of fold
  cat("======= Fold", i, "========", "\n\n")

  #initialize index for i-fold, for example: i=1,n=32,k=5 will give index of 1-6
```

```
  ind_i = ((i-1)*l_bloc + 1) : (i*l_bloc)

  #initialize data for test and train
  tabTrain = tab[-ind_i,]
  tabTest = tab[ind_i,]

  #initialize predictor and response variable for train, test
  X_train = subset(tabTrain, select = -YBin)
  Y_train = tabTrain$YBin
  X_test = subset(tabTest, select = -YBin)
  Y_test = tabTest$YBin

  #perform cross-validation to find best lambda
  cv_ridge <- cv.glmnet(as.matrix(X_train), Y_train, alpha = 0, family = "binomial", grouped=FALSE)

  #train penalized ridge model using best lambda obtained from cv
  modTrain = glmnet(as.matrix(X_train), Y_train, alpha = 0,
                    family = "binomial", lambda = cv_ridge$lambda.1se)

  #find probability and make decision for this model (prediction and decision step)
  probs = predict(modTrain, s = cv_ridge$lambda.1se, newx = as.matrix(X_test),
                  type = "response")
  Y_hat = ifelse(probs > MAP_threshold, 1, 0)

  #find error and accuracy
  mse = MSE(Y_test, Y_hat)
  accu = mean(Y_test == Y_hat)

  #sum up each fold to find total error and accuracy
  total_MSE = total_MSE + mse
  total_accu = total_accu + accu

  #display performance
  cat("Lambda selected for Fold", i, ":", cv_ridge$lambda.1se, "\n")
  cat("MSE for Fold", i, ":", mse, "\n")
  cat("Accuracy for Fold", i, ":", accu, "\n\n")
}
```

```
## ======= Fold 1 ========
##
## Lambda selected for Fold 1 : 305.3628
## MSE for Fold 1 : 0.5
## Accuracy for Fold 1 : 0.5
##
## ======= Fold 2 ========
##
## Lambda selected for Fold 2 : 288.2237
## MSE for Fold 2 : 0.5
## Accuracy for Fold 2 : 0.5
##
## ======= Fold 3 ========
##
## Lambda selected for Fold 3 : 223.3328
```

```
## MSE for Fold 3 : 0.3333333
## Accuracy for Fold 3 : 0.6666667
##
## ======= Fold 4 ========
##
## Lambda selected for Fold 4 : 286.6561
## MSE for Fold 4 : 0.3333333
## Accuracy for Fold 4 : 0.6666667
##
## ======= Fold 5 ========
##
## Lambda selected for Fold 5 : 156.1311
## MSE for Fold 5 : 0
## Accuracy for Fold 5 : 1
```

```
#find average performance for this model
avg_mse = total_MSE/k
avg_accu = total_accu/k

#print end fold
cat("======= End Fold =======\n\n")
```

```
## ======= End Fold =======
```

```
#display the average performance
cat("Average MSE: ", avg_mse, "\n")
```

```
## Average MSE:  0.3333333
```

```
cat("Average accuracy: ", avg_accu, "\n")
```

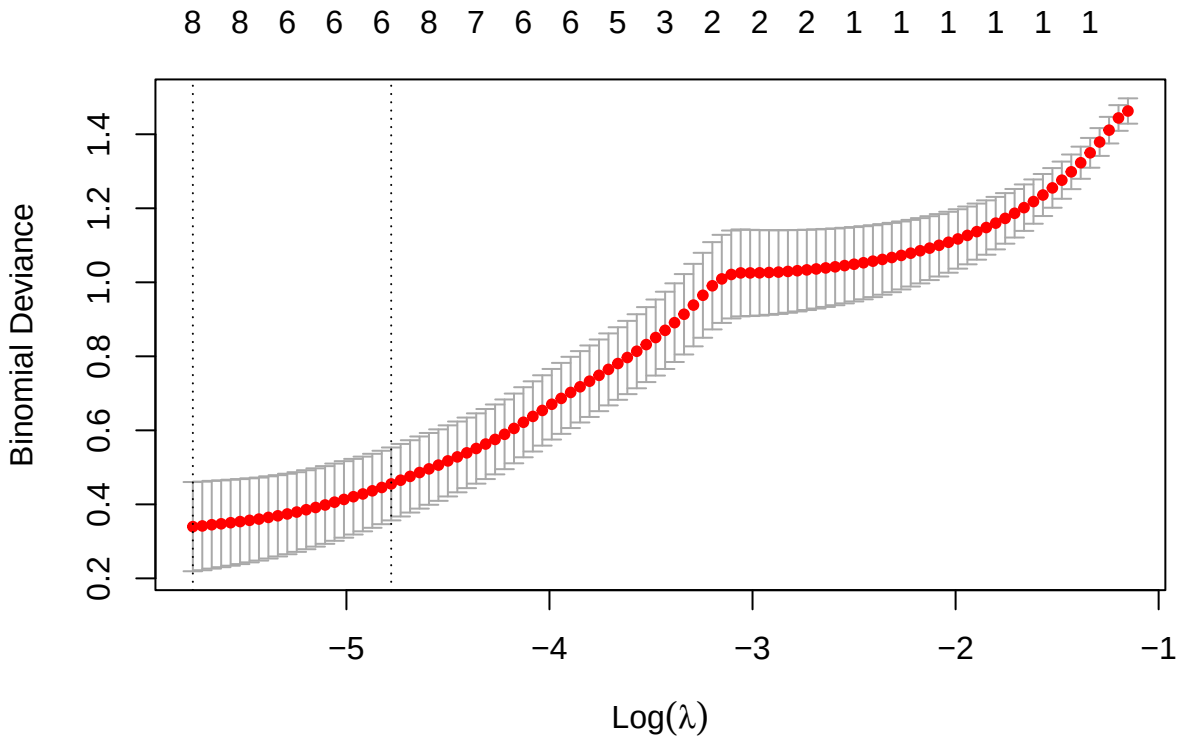
```
## Average accuracy:  0.6666667
```

```
ridge_1se_perf <- list(MSE = avg_mse, accuracy = avg_accu)
```

Penalized Logistic Regression: Lasso

- We follow the same procedure as before (Ridge), by just changing $\alpha = 0$ to $\alpha = 1$.

```
cv_lasso <- cv.glmnet(as.matrix(X_spec), Y_spec, alpha = 1, family = "binomial")
plot(cv_lasso)
```



- In the same way, the plot displays the cross-validation error according to the log of lambda. We will see its exact value below.

```
cv_lasso$lambda.min
```

```
## [1] 0.003162763
```

```
cv_lasso$lambda.1se
```

```
## [1] 0.008400575
```

Estimation step

- Lasso regression is effective for variable selection by setting some coefficients to exactly zero, as for Ridge, it only tries to keep all coefficients relatively small. We now see the remaining variables.

```
lassoMin <- glmnet(as.matrix(X_spec), Y_spec, alpha = 1,
                   family = "binomial", lambda = cv_lasso$lambda.min)

lasso1se <- glmnet(as.matrix(X_spec), Y_spec, alpha = 1,
                   family = "binomial", lambda = cv_lasso$lambda.1se)
```

```
lassoMin_remain <- rownames(coef(lassoMin))[which(coef(lassoMin)[, "s0"] != 0)]
lassoMin_remain
```

```
## [1] "(Intercept)" "X1208"       "X1724"       "X1902"       "X1912"
## [6] "X2072"       "X2074"       "X2300"
```

- These are the values correspond to the coefficients labeled above.

```
lassoMin_coef <- as.numeric(coef(lassoMin)) #set as numeric values
lassoMin_coef <- lassoMin_coef[lassoMin_coef != 0] #see how many variables were selected
lassoMin_coef
```

```
## [1]   1.963472 -17.398336 -37.742525 175.186760  26.272032 -24.068952 -87.211198
## [8] -42.066689
```

Prediction step

- Now we find the probability for λ_{min}, and λ_{1se} model

```
probs_lassoMin <- predict(lassoMin, s = cv_lasso$lambda.min,
                          type = "response", newx = as.matrix(X_spec))
probs_lasso1se <- predict(lasso1se, s = cv_lasso$lambda.1se,
                          type = "response", newx = as.matrix(X_spec))
```

Decision step

- Then, we make a decision based on the probability obtained

```
Y_hat_lassoMin <- ifelse(probs_lassoMin > MAP_threshold, 1, 0)
Y_hat_lasso1se <- ifelse(probs_lasso1se > MAP_threshold, 1, 0)
```

Model performance

- Lastly, we find the error for each model to see which one is better.

```
MSE(Y_spec, Y_hat_lassoMin)
```

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

```
MSE(Y_spec, Y_hat_lasso1se)
```

```
## [1] 0.03125
```

Here, again, the λ_{min} is more accurate then the λ_{1se} one, hence, we will perform k-fold only with the λ_{min}.

K-fold cross validation for l1 penalized logistic regression: Lasso

- Now we perform k-fold cross validation for l1 penalized logistic regression using λ_{min} as the best lambda.

```
#set index to number of observations we have then sample it to have random order
ind = sample(nrow(spectra_data))

#assign a new tab equal to random order of spectra
tab = spectra_data[ind,]

#initialize number of fold
k = 5

#num of obs in order to find each block
n = nrow(spectra_data)

#divide into block based on obs-n and number of folds k
l_bloc = trunc(n/k)

#initlialize the total of model performance
total_MSE = 0
total_accu = 0

#k-fold procedure
for (i in 1:k){
  #print order of fold
  cat("======= Fold", i, "========", "\n\n")

  #initialize index for i-fold, for example: i=1,n=32,k=5 will give index of 1-6
  ind_i = ((i-1)*l_bloc + 1) : (i*l_bloc)

  #initialize data for test and train
  tabTrain = tab[-ind_i,]
  tabTest = tab[ind_i,]

  #initialize predictor and response variable for train, test
  X_train = subset(tabTrain, select = -YBin)
  Y_train = tabTrain$YBin
  X_test = subset(tabTest, select = -YBin)
  Y_test = tabTest$YBin

  #perform cross-validation to find best lambda
  cv_lasso <- cv.glmnet(as.matrix(X_train), Y_train, alpha = 1, family = "binomial", grouped = FALSE)

  #train penalized ridge model using best lambda obtained from cv
  modTrain = glmnet(as.matrix(X_train), Y_train, alpha = 1,
                    family = "binomial", lambda = cv_lasso$lambda.min)

  #find error and accuracy
  probs = predict(modTrain, s = cv_lasso$lambda.min, newx = as.matrix(X_test),
                  type = "response")
  Y_hat = ifelse(probs > MAP_threshold, 1, 0)

  #calculate error and accuracy for each fold
  mse = MSE(Y_test, Y_hat)
  accu = mean(Y_test == Y_hat)
```

```
  #sum up each fold to find total error and accuracy
  total_MSE = total_MSE + mse
  total_accu = total_accu + accu

  #print lambda selected, mse, accuracy for each fold
  cat("Lambda selected for Fold", i, ":", cv_lasso$lambda.min, "\n")
  cat("MSE for Fold", i, ":", mse, "\n")
  cat("Accuracy for Fold", i, ":", accu, "\n\n")
}
```

```
## ======= Fold 1 ========
##
## Lambda selected for Fold 1 : 0.002925508
## MSE for Fold 1 : 0
## Accuracy for Fold 1 : 1
##
## ======= Fold 2 ========
##
## Lambda selected for Fold 2 : 0.005892191
## MSE for Fold 2 : 0
## Accuracy for Fold 2 : 1
##
## ======= Fold 3 ========
##
## Lambda selected for Fold 3 : 0.002809667
## MSE for Fold 3 : 0.3333333
## Accuracy for Fold 3 : 0.6666667
##
## ======= Fold 4 ========
##
## Lambda selected for Fold 4 : 0.00284364
## MSE for Fold 4 : 0
## Accuracy for Fold 4 : 1
##
## ======= Fold 5 ========
##
## Lambda selected for Fold 5 : 0.003449371
## MSE for Fold 5 : 0.1666667
## Accuracy for Fold 5 : 0.8333333
```

```
#calculate the average performance
avg_mse = total_MSE/k
avg_accu = total_accu/k

#print performance
cat("======= End Fold =======\n\n")
```

```
## ======= End Fold =======
```

```
cat("Average MSE: ", avg_mse, "\n")
```

```
## Average MSE:  0.1
```

```
cat("Average accuracy: ", avg_accu, "\n")
```

```
## Average accuracy:  0.9
```

```
lasso_min_perf <- list(MSE = avg_mse, accuracy = avg_accu)
```

- Now we perform k-fold cross validation for l1 penalized logistic regression using λ_{1se} as the best lambda.

```
#set index to number of observations we have then sample it to have random order
ind = sample(nrow(spectra_data))

#assign a new tab equal to random order of spectra
tab = spectra_data[ind,]

#initialize number of fold
k = 5

#num of obs in order to find each block
n = nrow(spectra_data)

#divide into block based on obs-n and number of folds k
l_bloc = trunc(n/k)

#initlialize the total of model performance
total_MSE = 0
total_accu = 0

#k-fold procedure
for (i in 1:k){
  #print order of fold
  cat("======= Fold", i, "========", "\n\n")

  #initialize index for i-fold, for example: i=1,n=32,k=5 will give index of 1-6
  ind_i = ((i-1)*l_bloc + 1) : (i*l_bloc)

  #initialize data for test and train
  tabTrain = tab[-ind_i,]
  tabTest = tab[ind_i,]

  #initialize predictor and response variable for train, test
  X_train = subset(tabTrain, select = -YBin)
  Y_train = tabTrain$YBin
  X_test = subset(tabTest, select = -YBin)
  Y_test = tabTest$YBin

  #perform cross-validation to find best lambda
  cv_lasso <- cv.glmnet(as.matrix(X_train), Y_train, alpha = 1, family = "binomial", grouped = FALSE)

  #train penalized ridge model using best lambda obtained from cv
  modTrain = glmnet(as.matrix(X_train), Y_train, alpha = 1,
                    family = "binomial", lambda = cv_lasso$lambda.1se)
```

```
  #find error and accuracy
  probs = predict(modTrain, s = cv_lasso$lambda.1se, newx = as.matrix(X_test),
                  type = "response")
  Y_hat = ifelse(probs > MAP_threshold, 1, 0)

  #calculate error and accuracy for each fold
  mse = MSE(Y_test, Y_hat)
  accu = mean(Y_test == Y_hat)

  #sum up each fold to find total error and accuracy
  total_MSE = total_MSE + mse
  total_accu = total_accu + accu

  #print lambda selected, mse, accuracy for each fold
  cat("Lambda selected for Fold", i, ":", cv_lasso$lambda.1se, "\n")
  cat("MSE for Fold", i, ":", mse, "\n")
  cat("Accuracy for Fold", i, ":", accu, "\n\n")
}
```

```
## ======= Fold 1 ========
##
## Lambda selected for Fold 1 : 0.02289081
## MSE for Fold 1 : 0
## Accuracy for Fold 1 : 1
##
## ======= Fold 2 ========
##
## Lambda selected for Fold 2 : 0.02926221
## MSE for Fold 2 : 0
## Accuracy for Fold 2 : 1
##
## ======= Fold 3 ========
##
## Lambda selected for Fold 3 : 0.01684343
## MSE for Fold 3 : 0.3333333
## Accuracy for Fold 3 : 0.6666667
##
## ======= Fold 4 ========
##
## Lambda selected for Fold 4 : 0.02103788
## MSE for Fold 4 : 0.6666667
## Accuracy for Fold 4 : 0.3333333
##
## ======= Fold 5 ========
##
## Lambda selected for Fold 5 : 0.005590667
## MSE for Fold 5 : 0.1666667
## Accuracy for Fold 5 : 0.8333333
```

```
#calculate the average performance
avg_mse = total_MSE/k
avg_accu = total_accu/k
```

```
#print performance
cat("======= End Fold =======\n\n")
```

```
## ======= End Fold =======
```

```
cat("Average MSE: ", avg_mse, "\n")
```

```
## Average MSE:  0.2333333
```

```
cat("Average accuracy: ", avg_accu, "\n")
```

```
## Average accuracy:  0.7666667
```

```
lasso_1se_perf <- list(MSE = avg_mse, accuracy = avg_accu)
```

Conclusion

```
data.frame(
  Model= c("Ridge Min", "Ridge 1se", "Lasso Min", "Lasso 1se"),
  MSE = c(ridge_min_perf$MSE, ridge_1se_perf$MSE,
          lasso_min_perf$MSE, lasso_1se_perf$MSE),
  Accuracy = c(ridge_min_perf$accuracy, ridge_1se_perf$accuracy,
               lasso_min_perf$accuracy, lasso_1se_perf$accuracy)
  )
```

```
##       Model       MSE  Accuracy
## 1 Ridge Min 0.2333333 0.7666667
## 2 Ridge 1se 0.3333333 0.6666667
## 3 Lasso Min 0.1000000 0.9000000
## 4 Lasso 1se 0.2333333 0.7666667
```

- In the final analysis, not only Lasso provides a simpler model than Ridge since it eliminates certain variables from the model, but visibly, the table above also suggests that, the l_1 penalized regression with the chosen λmin has a better prediction than Ridge; therefore, it is the best model to explain the low or high values of the fat given the spectra.