Lab 1 Block 1 Group K1

Nicolas Taba , Siddharth Saminathan , Salvador Marti Roman

11/17/2020

Statement of contribution

Nicolas Taba contributed to assignment 1. Salvador Marti Roman contributed to assignment 2. Siddharth Saminathan contributed to assignment 3.

All members of the group discussed the difficulties encountered.

Assignment 1. Handwritten digit recognition with K-means

Import data dividing it into sets.

```
# Set working directory

library(ggplot2)
library(kknn)
library("ggpubr")

# read the data file and process names and target.
opt =
  "https://raw.githubusercontent.com/TheClassyPenguin/MachineLearningLab1/main/optdigits.csv"
data <- read.csv(url(opt))
data[,ncol(data)] <- data.frame(sapply(data[,ncol(data)], as.character),
                                stringsAsFactors = TRUE)
names(data) <- c(seq(1:(ncol(data)-1)),"target")

# Split the data into training/validation/test (50/25/25), need to separate our target.65th column

n <- dim(data)[1]
set.seed(12345)
id <- sample(1:n, floor(n*0.5))
train_set <- data[id,]
id1 <- setdiff(1:n, id)
set.seed(12345)
id2 <- sample(id1, floor(n*0.25))
valid_set <- data[id2,]
id3 <- setdiff(id1,id2)
test_set <- data[id3,]
```

Use the training data to fit a 30-nearest neighbor classifier.

```
## Confusion matrix for the test data

##    model_fit
```

```
##      0  1  2  3  4  5  6  7  8  9
##   0 98  0  0  0  0  0  0  0  0  0
##   1  0 92  3  0  0  0  0  0  0  2
##   2  0  0 93  1  0  0  0  0  1  0
##   3  0  0  0 95  0  0  0  2  1  0
##   4  1  0  0  0 89  0  1  5  1  3
##   5  0  1  0  0  0 81  0  0  0  5
##   6  0  0  0  0  0  0 94  0  0  0
##   7  0  2  0  0  0  0  0 92  1  0
##   8  0  3  0  1  0  0  1  0 86  0
##   9  0  0  0  3  0  0  0  2  0 96
```

```
##  The missclassification rate for the test data is:  0.041841
```

```
## Confusion matrix for the training data.
```

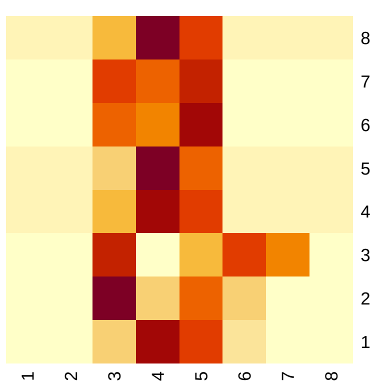
```
##    model_fit_train
##       0   1   2   3   4   5   6   7   8   9
##   0 177   0   0   0   1   0   0   0   0   0
##   1   0 174   9   0   0   0   1   0   1   3
##   2   0   0 171   0   0   0   0   1   1   0
##   3   0   0   0 198   0   1   0   1   0   0
##   4   0   1   0   0 168   0   1   6   1   2
##   5   0   0   0   0   0 186   0   2   0   9
##   6   0   0   0   0   0   0 200   0   0   0
##   7   0   1   0   1   0   0   0 193   0   0
##   8   0   7   0   1   0   0   1   0 196   0
##   9   0   3   0   2   2   0   0   2   3 184
```

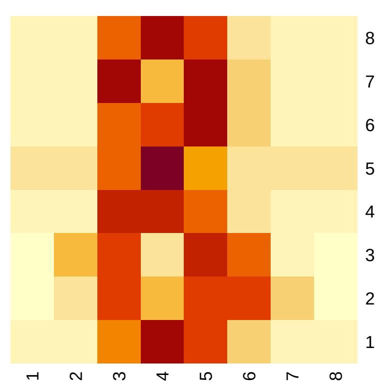
```
##  The missclassification rate for the training data is:  0.03349032
```

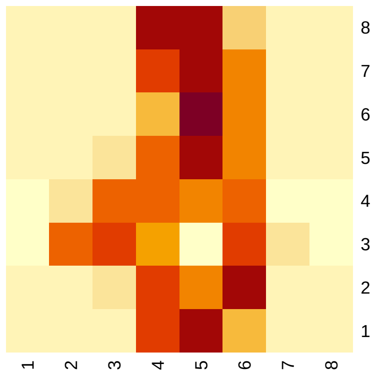
The missclassification rate increases from 3% to 4% from the training set to the data set. Most of the missclassifications occur with digits 4 and 8 in both training and test sets. The increase in missclassification between the training and data set is expected as the variance between true target class and model is increased and we can expect more overfitting when testing on the training set itself (less missclassification).

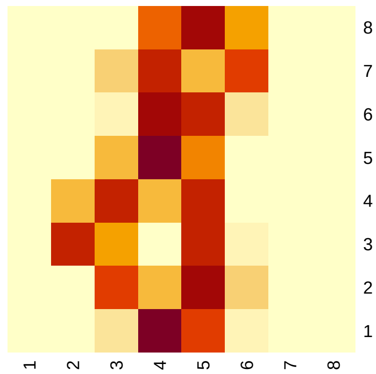
Find the best and worst cases of digit "8"

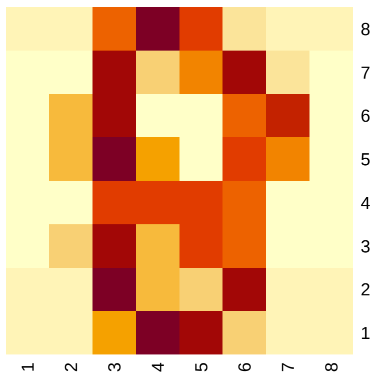
We plot the 2 best cases of 8s and the 3 worst cases of 8s from the training data. s







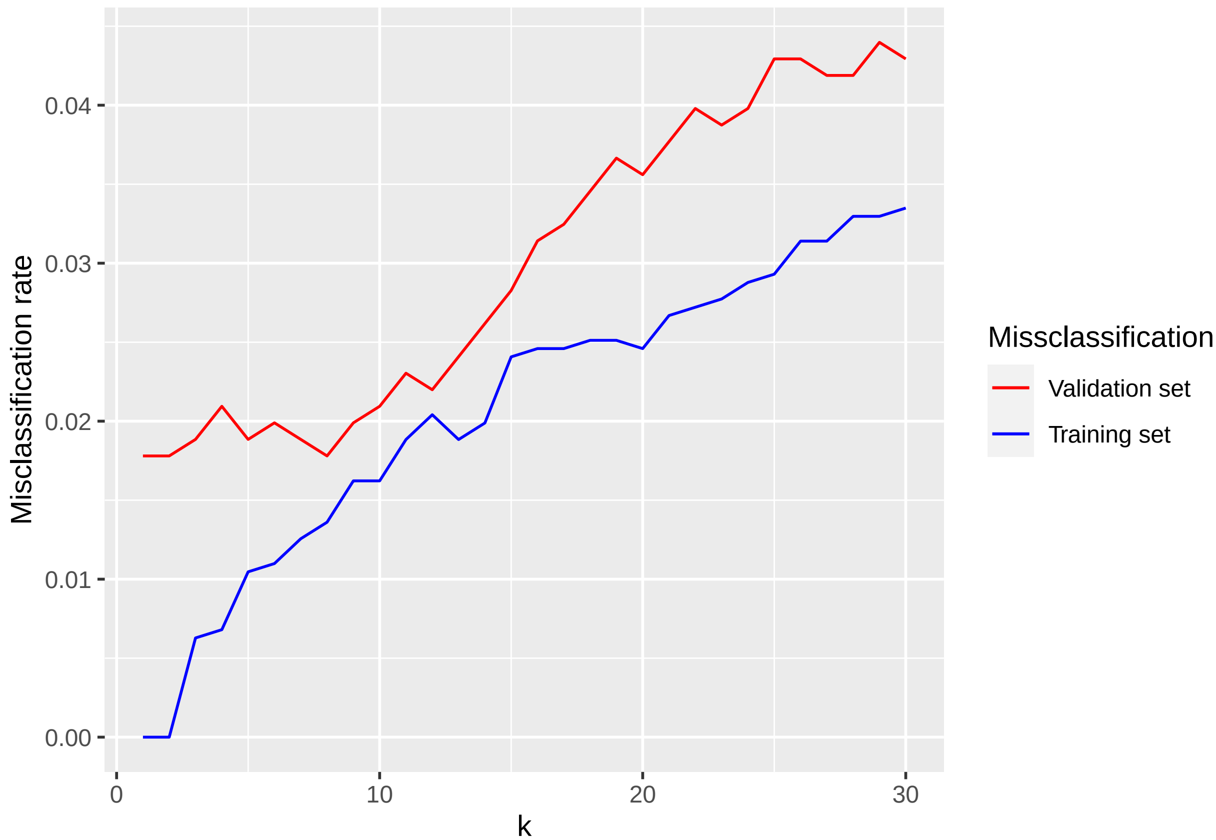




The best result is somewhat difficult to assess if it is an 8 visually. However, the last two of the worst results are not and can quite easily be conceived to being 8s.

Fit k-nearest neighbor classifiers for different values of K.

We let K = 1,2,. . . ,30 and plot the dependence of the trianing data and validation missclassification errors as a function of K.

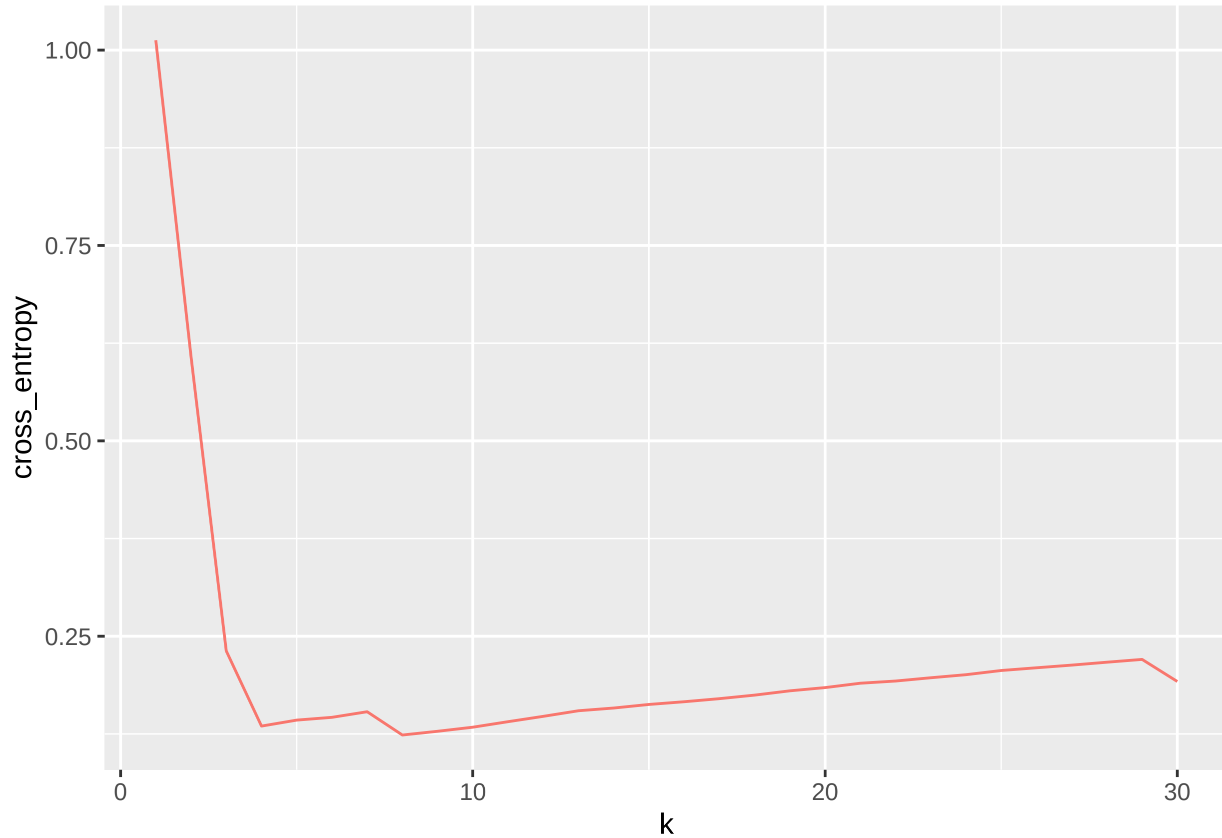



```
## Best K is:  1 2 8
```

```
## [1] 0.03138075
```

For smaller values of k, our model becomes more complex. The training and validation errors increase for increasing values of K. There is low bias and high variance for small values of K. For larger values of K, the bias is high and our predictions report greater errors. If we look for the model with the smallest value of validation errors, we find the best value of K to be 1,2 and 8. Here, we chose to compute the missclassification rate for K = 8 as at that value, the model is the most general and reduces the variance while increasing the bias. We find test error rate equal to 3% which is much higher than our training and validation set errors (around 1,7% for each). We expect the test errors to be greater, but we might want to use a different metric to measure the fit of our model in order to make sure that we have a model with preditions that are general enough to ensure that we have chosen the right bias/variance tradeoff in our choice of K.

Fit k-nearest neighbour classifier and use cross-entropy to choose the best classifier.




```
## Best K is:  8
```

The optimal value for k in this instance is 8. This is a better choice than missclassification error because we are accounting for the uncertainty of missclasifying to a different class than the target whereas missclassification error imposes a strict boundary between classes.

Assignment 2. Ridge regression and model selection

The data file parkinson.csv is composed of a range of biomedical voice measurements from 42 people with early-stage Parkinson's disease recruited to a six-month trial of a telemonitoring device for remote symptom progression monitoring. The purpose is to predict Parkinson's disease symptom score (motor UPDRS) from the following voice characteristics:

- Jitter(%),Jitter(Abs),Jitter:RAP,Jitter:PPQ5,Jitter:DDP - Several measures of variation in fundamental frequency
- Shimmer,Shimmer(dB),Shimmer:APQ3,Shimmer:APQ5,Shimmer:APQ11,Shi mmer:DDA - Several measures of variation in amplitude
- NHR,HNR - Two measures of ratio of noise to tonal components in the voice
- RPDE - A nonlinear dynamical complexity measure
- DFA - Signal fractal scaling exponent
- PPE - A nonlinear measure of fundamental frequency variation

```
csv_url =
  "https://raw.githubusercontent.com/TheClassyPenguin/MachineLearningLab1/main/parkinsons.csv"
raw_data = read.csv(url(csv_url), header=TRUE)
```

1. Assuming that motor_UPDRS is normally distributed and can be modeled by Ridge regression of the voice characteristics, write down the probabilistic model as a Bayesian model.

$$p(w, \sigma|D) \propto p(D|w, \sigma) * p(w, \sigma)$$

$$p(w \mid D) \propto \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - w^T X_i)^2}{2\sigma^2}} * \prod_{i=1}^{n} \frac{\sqrt{\lambda}}{2\sigma^2} e^{-\lambda \frac{w_i^2}{2\sigma^2}}$$

2. Scale the data and divide it into training and test data (60/40). Due to this, compute all models without intercept in the following steps

```
train_test_split = function(data, proportion){
  proportion_size = floor(proportion*nrow(data))

  train = data[sample(seq_len(nrow(data)), size = proportion_size),]
  test = data[-sample(seq_len(nrow(data)), size = proportion_size),]

  return(list(train=train,test=test))
}

exclude_columns = c("subject.","sex","age","test_time","total_UPDRS")#,"motor_UPDRS")

# Excludes from scaling
raw_data[,!names(raw_data) %in% exclude_columns] = apply(
  raw_data[,!names(raw_data) %in% exclude_columns],
  2,
  scale)

dataset = train_test_split(raw_data, 0.6)

exclude_x_columns = c("subject.","sex","age","test_time","motor_UPDRS","total_UPDRS")

x_train = dataset$train[,!names(dataset$train) %in% exclude_x_columns]
y_train = dataset$train[,names(dataset$train) == "motor_UPDRS"]
```

```
x_test = dataset$test[,!names(dataset$test) %in% exclude_x_columns]
y_test = dataset$test[,names(dataset$test) == "motor_UPDRS"]
```

3. Implement 4 following functions by using basic R commands only (no external packages):

a. Loglikelihood function that for a given parameter vector w and dispersion σ computes the log-likelihood function $\log P(D \mid w, \sigma)$ for the model from step 1 for the training data

$$p(D|w,\sigma) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - w^T X_i)^2}{2\sigma^2}} = (\frac{1}{\sigma\sqrt{2\pi}})^n e^{-\sum_{i=1}^{n} \frac{(y_i - w^T X_i)^2}{2\sigma^2}}$$

$$\log p(D|w,\sigma) = -\frac{\sum_{i=1}^{n}(y_i - w^T X_i)^2}{2\sigma^2} + n\log\frac{1}{\sigma\sqrt{2\pi}}$$

```
loglikelihood = function(x, y, w, dispersion){
  n = dim(x)[1] #for every datapoint
  base = n*log(1/(abs(dispersion)*sqrt(2*pi)))
  exponent = -sum((y - t(t(w)%*%t(x)))**2) / (2*(dispersion**2))
  return(base + exponent)
}
```

b. Ridge function that for given vector w, scalar σ and scalar λ uses function from 2a and adds up a Ridge penalty to the minus loglikelihood.

$$p(w,\sigma) = \prod_{i=1}^{m} \frac{\sqrt{\lambda}}{2\sigma^2} e^{-\lambda\frac{w_i^2}{2\sigma^2}} = (\frac{\sqrt{\lambda}}{2\sigma^2})^m e^{-\lambda\sum_{i=1}^{m}\frac{w_i^2}{2\sigma^2}}$$

$$\log p(w,\sigma) = -\lambda\frac{\sum_{i=1}^{m} w_i^2}{2\sigma^2} + m\log(\frac{\sqrt{\lambda}}{2\sigma^2})$$

$$\log p(w,\sigma|D) = -\frac{\sum_{i=1}^{n}(y_i - w^T X_i)^2}{2\sigma^2} + n\log\frac{1}{\sigma\sqrt{2\pi}} + -\lambda\frac{\sum_{i=1}^{m} w_i^2}{2\sigma^2} + m\log(\frac{\sqrt{\lambda}}{2\sigma^2})$$

```
ridge = function(par, x, y, lambda){
  m = dim(x)[2] #for every parameter
  w = as.matrix(par[1:(length(par)-1)])
  dispersion = par[length(par)]

  #Avoids 0 value without need for optimizer restriction
  if(isTRUE(all.equal(dispersion,0))){dispersion = dispersion + 0.0000001}

  exponent = - lambda * sum( w**2)  / (2 * (dispersion**2))
  base = m*log(sqrt(lambda) / (2 * (dispersion**2)))
  reg = base + exponent
  return(loglikelihood(x, y, w, dispersion)  + reg)
}
```

c. RidgeOpt function that depends on scalar λ , uses function from 2b and function optim() with method="BFGS" to find the optimal w and$\sigma$for the given λ.

```
ridgeOpt = function(lambda, x, y){
  x=as.matrix(x)
  y=as.matrix(y)
```

```r
  #Random Parameter initilization
  w = as.matrix(rnorm(dim(x)[2], mean = 0, sd = 0.01))
  sigma = runif(1, 0.001, 1)

  result = optim(par = c(w,sigma),
   ridge,
   x=x,
   y=y,
   lambda=lambda,
   method = "BFGS",
   control=list(fnscale=-1) #Maximizing instead of minimizing
   )
  return(result)
}

ridge_fit = ridgeOpt(1000, x_train, y_train)
ridge_fit
```

```
## $par
##  [1]  0.013868535 -0.049674413  0.001068370 -0.009435984  0.001146493
##  [6]  0.001935832  0.013373252 -0.027849332 -0.015873881  0.090313544
## [11] -0.027780413 -0.062727554 -0.100259104  0.026450260 -0.161376079
## [16]  0.145739424  0.970414364
##
## $value
## [1] -4866.758
##
## $counts
## function gradient
##      122       36
##
## $convergence
## [1] 0
##
## $message
## NULL
```

d. D function that for a given scalar λ computes the degrees of freedom of the regression model from step 1 based on the training data.

```r
D = function(lambda, params, x, y){
  w = as.matrix(params)
  x = as.matrix(x)
  y = as.matrix(y)
  df = x %*% solve(t(x) %*% x + (lambda * diag(length(w)))) %*% t(x)
  return(sum(diag(df)))
}

D(lambda = 0.2,
 params = ridge_fit$par[1:(length(ridge_fit$par)-1)],
 x = x_train,
 y = y_train)
```

```
## [1] 13.97798
```

4. By using function RidgeOpt, compute optimal w parameters for $\lambda = 1$, $\lambda = 100$ and $\lambda = 1000$. Use the estimated parameters to predict the motor_UPDRS values for training and test data and report the training and test MSE values. Which penalty parameter is most appropriate among the selected ones? Why is MSE a more appropriate measure here than other empirical risk functions?

```r
get_mean_square_error = function(params, x, y){
  w = as.matrix(params)
  x = as.matrix(x)
  y = as.matrix(y)
  mean_err = mean((y - t(t(w)%*%t(x)))**2)
  return(mean_err)
}

evaluate_performance = function(lambdas, x_train, y_train, x_test, y_test){
  train_results = c()
  test_results = c()
  for(i in 1:length(lambdas)){
    ridge_fit = ridgeOpt(lambdas[i], x_train, y_train)

    #Evaluating on train
    train_results = c(train_results,get_mean_square_error(
      params = ridge_fit$par[1:(length(ridge_fit$par)-1)],
      x_train,
      y_train))

    #Evaluating on test
    test_results = c(test_results, get_mean_square_error(
      params = ridge_fit$par[1:(length(ridge_fit$par)-1)],
      x_test,
      y_test))

  }
  result_frame = data.frame(train_results=train_results, test_results=test_results)
  rownames(result_frame) = lambdas
  return(result_frame)
}


lambdas = c(1, 100, 1000)
evaluate_performance(lambdas, x_train, y_train, x_test, y_test)
```

```
##      train_results test_results
## 1        0.9045600    0.9061764
## 100      0.9071588    0.9084115
## 1000     0.9290223    0.9305354
```

Of the 3 values tested of λ tested, $\lambda = 1$ performs better in both the train and test sets.

MSE is a function that heavily penalizes big errors due to how the difference of squares work. For this particular problem, diagnosing someone's motor skill precisely might be much less important than being in roughly correct.

MSE is also a smooth and continuous function known to perform well in gradient based optimization problems. BFGS as part of the hill-climbing family of optimization techniques benefits from the gradients created by this risk function.

5. Use functions from step 3 to compute AIC (Akaike Information Criterion) scores for the Ridge models with values $\lambda = 1$, $\lambda = 100$ and $\lambda = 1000$ and their corresponding optimal parameters w andσcomputed in step 4. What is the optimal model according to AIC criterion? What is the theoretical advantage of this kind of model selection compared to the holdout model selection done in step 4?

```
AIC = function(lambda, par, x, y){
  # AIC = -2(log-likelihood) + 2K
  loglik = ridge(par, x, y, lambda)
  params = par[1:(length(par)-1)]

  freedom = D(lambda, params, x, y)

  return(-2*loglik + 2*freedom)
}

evaluate_AIC = function(lambdas, x, y){
  AIC_results=c()

  for(i in 1:length(lambdas)){
    ridge_fit = ridgeOpt(lambdas[i], x, y)
    AIC_results=c(AIC_results,AIC(lambda=lambdas[i], par=ridge_fit$par, x=x, y=y))
  }

  result_frame = data.frame(AIC_SCORE=AIC_results)
  rownames(result_frame) = lambdas

  return(result_frame)
}

complete_x = rbind(x_train, x_test)
complete_y = c(y_train, y_test)

evaluate_AIC(lambdas = lambdas, x = complete_x, y = complete_y)
```

```
##      AIC_SCORE
## 1     16132.55
## 100   16098.36
## 1000  16233.25
```

Of the 3 values tested of λ tested, $\lambda = 100$ performs better.

The theoretical advantage of using AIC scores to determine model performance is that all of the data can be used to train the model while still being able to measure how well the model generalises. This can be useful when the amount of data available is very low.

Assignment 3.Linear regression and LASSO

Splitting the data into Train and Test sets.

```
#Splitting Data
tec =
  "https://raw.githubusercontent.com/TheClassyPenguin/MachineLearningLab1/main/tecator.csv"

data<-data.frame(read.csv(url(tec)))

n<-dim(data)[1]
set.seed(12345)
id<-sample(1:n ,floor(n*0.5))
train<-data[id,]
test<-data[-id,]
```

1

A linear Regression of Fat vs Channels was fit.

```
##
## Call:
## lm(formula = Fat ~ . - ï..Sample - Protein - Moisture, data = train)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.201500 -0.041315 -0.001041  0.037636  0.187860
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.815e+01  5.488e+00  -3.306  0.01628 *
## Channel1     2.653e+04  1.126e+04   2.357  0.05649 .
## Channel2    -5.871e+04  3.493e+04  -1.681  0.14385
## Channel3     1.154e+05  7.373e+04   1.565  0.16852
## Channel4    -2.432e+05  1.175e+05  -2.070  0.08387 .
## Channel5     3.026e+05  1.193e+05   2.536  0.04430 *
## Channel6    -2.365e+05  8.160e+04  -2.898  0.02741 *
## Channel7     1.090e+05  3.169e+04   3.440  0.01380 *
## Channel8    -6.054e+04  1.508e+04  -4.015  0.00700 **
## Channel9     7.871e+04  2.160e+04   3.643  0.01079 *
## Channel10   -1.730e+04  1.640e+04  -1.055  0.33215
## Channel11    9.562e+04  3.529e+04   2.710  0.03512 *
## Channel12   -2.114e+05  6.198e+04  -3.410  0.01431 *
## Channel13    9.725e+04  4.424e+04   2.198  0.07026 .
## Channel14    5.296e+04  4.666e+04   1.135  0.29968
## Channel15   -7.855e+04  5.245e+04  -1.498  0.18491
## Channel16   -8.209e+03  1.893e+04  -0.434  0.67969
## Channel17    3.769e+04  1.987e+04   1.897  0.10666
## Channel18    3.306e+04  7.934e+03   4.167  0.00590 **
## Channel19   -8.405e+04  1.929e+04  -4.358  0.00478 **
## Channel20    1.510e+05  3.361e+04   4.492  0.00414 **
## Channel21   -2.069e+05  4.256e+04  -4.862  0.00282 **
## Channel22    1.348e+05  3.824e+04   3.526  0.01243 *
## Channel23   -4.094e+04  3.546e+04  -1.154  0.29222
## Channel24    2.023e+04  2.761e+04   0.733  0.49134
```

```
## Channel25    3.269e+03  1.071e+04   0.305  0.77045
## Channel26   -1.297e+04  7.636e+03  -1.699  0.14028
## Channel27    4.131e+03  1.422e+04   0.291  0.78120
## Channel28   -4.548e+03  2.988e+04  -0.152  0.88402
## Channel29    1.089e+04  1.768e+04   0.616  0.56072
## Channel30   -7.985e+04  2.653e+04  -3.010  0.02371 *
## Channel31    1.756e+05  5.279e+04   3.326  0.01589 *
## Channel32   -1.107e+05  2.904e+04  -3.813  0.00883 **
## Channel33   -6.525e+04  5.407e+04  -1.207  0.27294
## Channel34    1.007e+05  6.589e+04   1.528  0.17738
## Channel35   -2.841e+03  1.214e+04  -0.234  0.82266
## Channel36   -2.268e+04  2.295e+04  -0.988  0.36127
## Channel37   -4.479e+04  1.292e+04  -3.468  0.01334 *
## Channel38    3.209e+04  1.843e+04   1.742  0.13221
## Channel39    1.992e+04  2.067e+04   0.964  0.37246
## Channel40   -9.833e+03  2.431e+04  -0.404  0.69988
## Channel41    1.659e+04  3.648e+04   0.455  0.66531
## Channel42   -1.829e+04  3.528e+04  -0.519  0.62260
## Channel43   -2.423e+04  2.427e+04  -0.998  0.35669
## Channel44    3.246e+04  2.013e+04   1.613  0.15793
## Channel45   -8.089e+03  4.023e+04  -0.201  0.84728
## Channel46    7.065e+03  2.810e+04   0.251  0.80990
## Channel47   -4.062e+04  1.007e+04  -4.034  0.00685 **
## Channel48    9.080e+04  2.618e+04   3.469  0.01332 *
## Channel49   -6.647e+04  2.372e+04  -2.803  0.03105 *
## Channel50   -4.196e+04  2.856e+04  -1.469  0.19213
## Channel51    1.097e+05  5.572e+04   1.968  0.09661 .
## Channel52   -1.148e+05  6.376e+04  -1.800  0.12196
## Channel53    9.525e+04  7.450e+04   1.278  0.24830
## Channel54   -4.534e+04  7.363e+04  -0.616  0.56067
## Channel55   -1.535e+03  4.933e+04  -0.031  0.97618
## Channel56   -2.377e+03  2.109e+04  -0.113  0.91394
## Channel57    3.174e+04  1.005e+04   3.158  0.01961 *
## Channel58    2.221e+03  1.048e+04   0.212  0.83915
## Channel59   -8.504e+04  2.574e+04  -3.304  0.01634 *
## Channel60    6.382e+04  1.607e+04   3.972  0.00735 **
## Channel61    2.151e+04  1.234e+04   1.742  0.13211
## Channel62   -2.859e+04  1.065e+04  -2.685  0.03631 *
## Channel63    1.796e+04  9.187e+03   1.955  0.09838 .
## Channel64    5.759e+04  3.526e+04   1.633  0.15354
## Channel65   -1.470e+05  6.911e+04  -2.127  0.07752 .
## Channel66    9.121e+04  4.461e+04   2.045  0.08688 .
## Channel67   -5.733e+03  2.197e+04  -0.261  0.80288
## Channel68   -6.290e+04  2.192e+04  -2.870  0.02843 *
## Channel69    6.421e+04  2.074e+04   3.096  0.02121 *
## Channel70   -1.749e+04  1.581e+04  -1.106  0.31111
## Channel71   -7.248e+03  1.934e+04  -0.375  0.72075
## Channel72    3.406e+04  1.185e+04   2.873  0.02830 *
## Channel73   -2.100e+04  1.132e+04  -1.855  0.11308
## Channel74   -3.314e+04  1.220e+04  -2.717  0.03480 *
## Channel75    7.039e+04  2.054e+04   3.427  0.01402 *
## Channel76   -3.187e+04  1.736e+04  -1.836  0.11597
## Channel77    2.061e+04  1.810e+04   1.138  0.29832
## Channel78   -1.180e+04  2.273e+04  -0.519  0.62225
```

```
## Channel79    2.669e+04  2.997e+04   0.890  0.40750
## Channel80   -6.051e+04  1.483e+04  -4.080  0.00650 **
## Channel81    1.386e+03  2.628e+04   0.053  0.95966
## Channel82    1.020e+05  4.694e+04   2.173  0.07275 .
## Channel83   -1.706e+05  4.688e+04  -3.640  0.01083 *
## Channel84    1.097e+05  2.892e+04   3.792  0.00905 **
## Channel85   -1.294e+05  3.600e+04  -3.594  0.01145 *
## Channel86    2.130e+05  4.345e+04   4.903  0.00270 **
## Channel87   -1.198e+05  3.818e+04  -3.139  0.02011 *
## Channel88   -2.199e+04  6.085e+04  -0.361  0.73021
## Channel89    7.974e+04  5.077e+04   1.571  0.16733
## Channel90   -1.711e+05  5.499e+04  -3.112  0.02079 *
## Channel91    2.107e+05  6.406e+04   3.289  0.01663 *
## Channel92   -1.959e+05  7.171e+04  -2.733  0.03407 *
## Channel93    2.874e+05  9.937e+04   2.892  0.02762 *
## Channel94   -3.064e+05  9.601e+04  -3.191  0.01881 *
## Channel95    2.048e+05  6.220e+04   3.292  0.01656 *
## Channel96   -5.600e+04  2.929e+04  -1.912  0.10441
## Channel97   -1.318e+04  3.050e+04  -0.432  0.68065
## Channel98   -2.724e+04  2.107e+04  -1.292  0.24375
## Channel99    3.556e+04  1.382e+04   2.573  0.04218 *
## Channel100  -1.206e+04  4.264e+03  -2.828  0.03006 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3191 on 6 degrees of freedom
## Multiple R-squared:      1,  Adjusted R-squared:  0.9994
## F-statistic:  1651 on 100 and 6 DF,  p-value: 1.058e-09
```

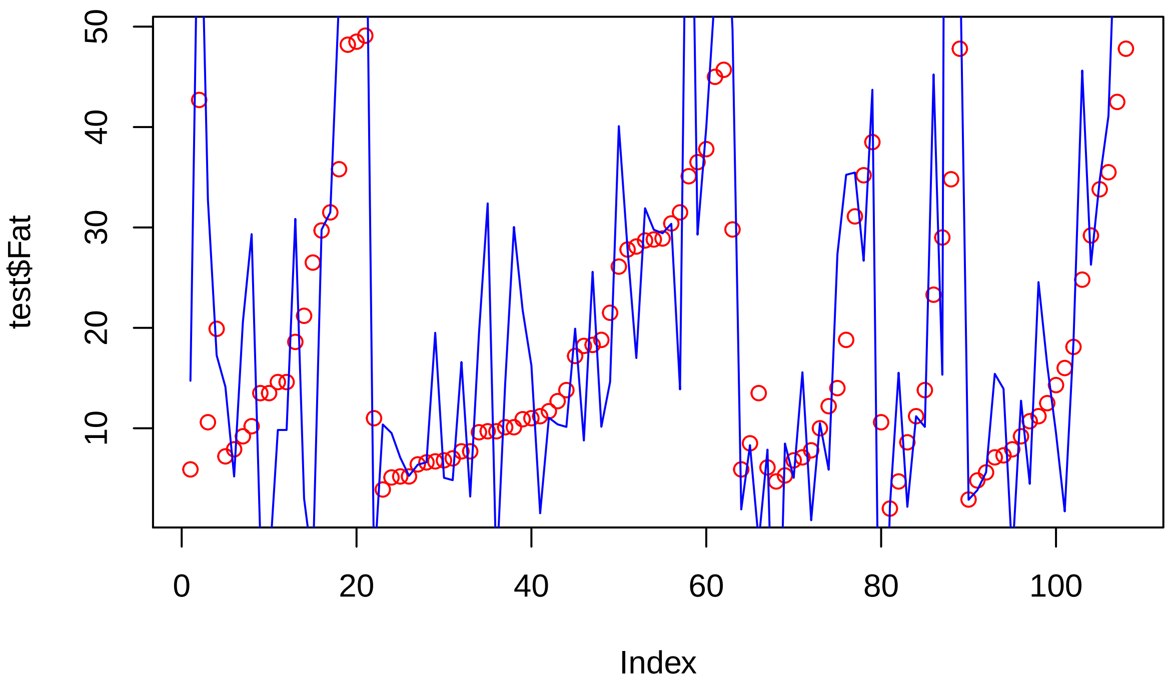
The Probabilistic Model was found to be a normal Distribution with

$$y = \beta_0 + \sum_{i=1}^{100} \beta_i * x_i + \epsilon \sim \mathsf{N}(\mu, \sigma^2)$$

Prediction and errors

```
##      train_rmse test_rmse
## [1,]  0.0755587  26.87805
```

Quality and Fit of the Model:



From the plot We can see that there is a higher variance in predicted values when compared to the original values. This shows us that the model didn't give us promising results.

The RMSE value is high and it clearly explains the reasons for higher variances from the original values.

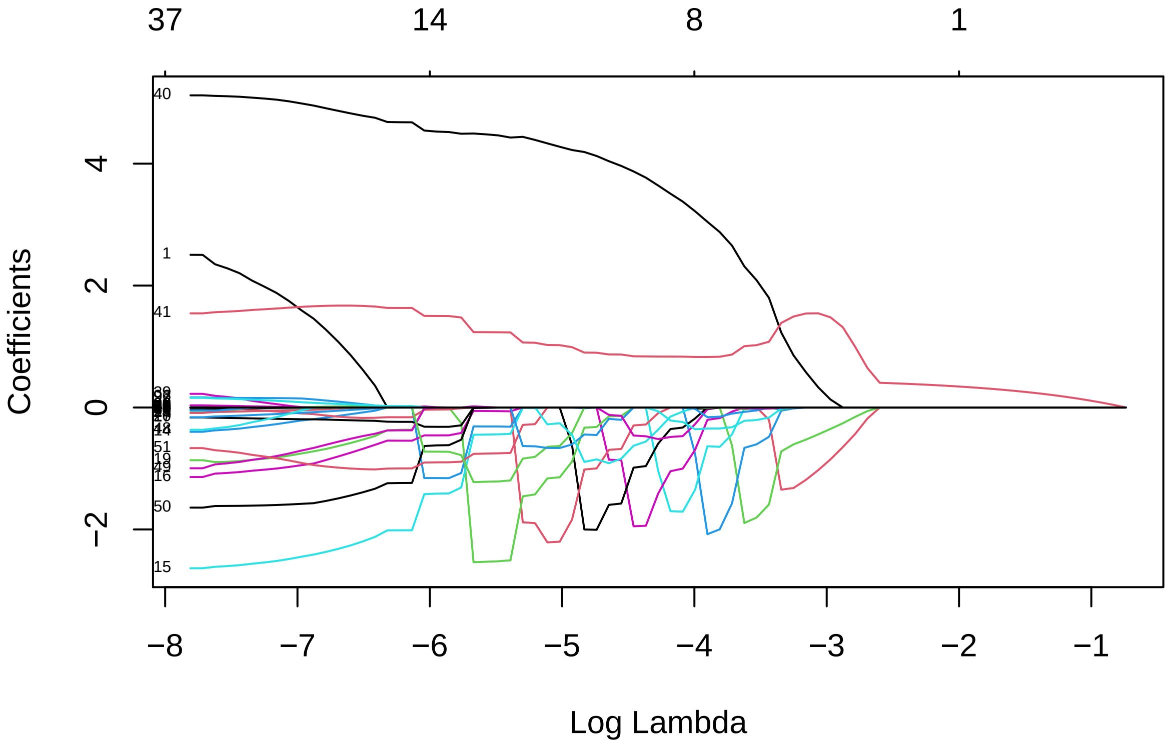
2. Function to be Optimized

$$\underset{beta}{\operatorname{argmin}} \left[\sum_{i=1}^{n} \left(Y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ji} \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right]$$

3. LASSO Regression

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

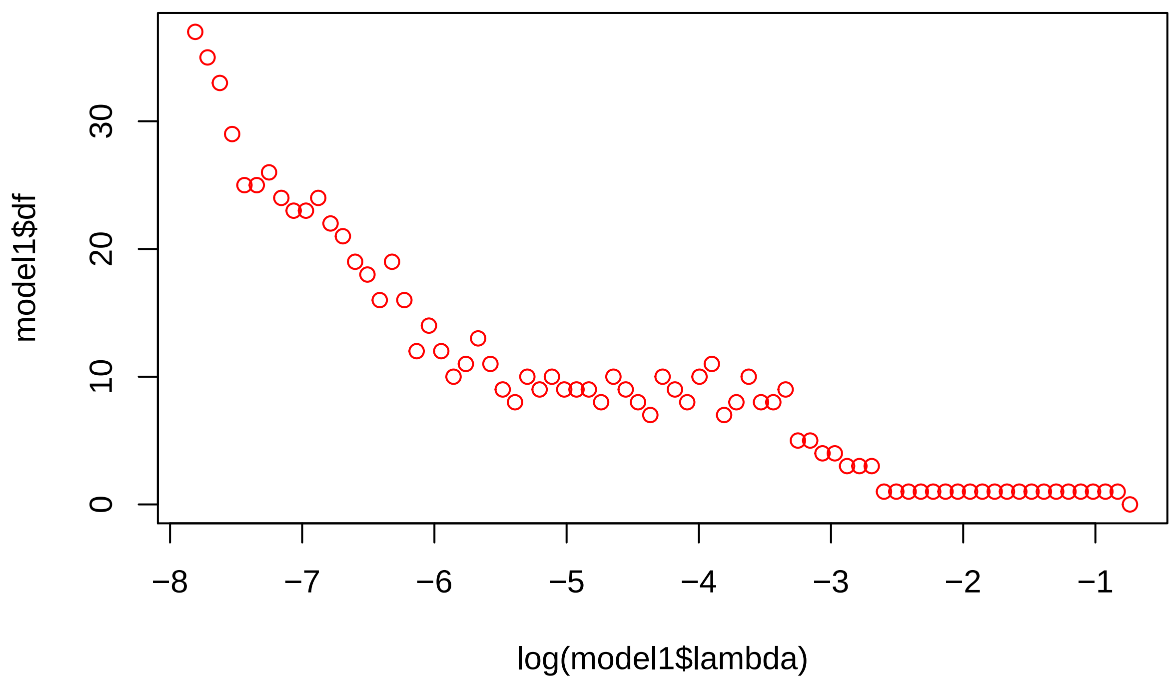
```
## Loaded glmnet 4.0-2
```



From the above plot, we can see that as lambda increases and the coefficients shirk thus reducing the variance.

For a model with only 3 features the value of λ=3. approx. log(λ)=

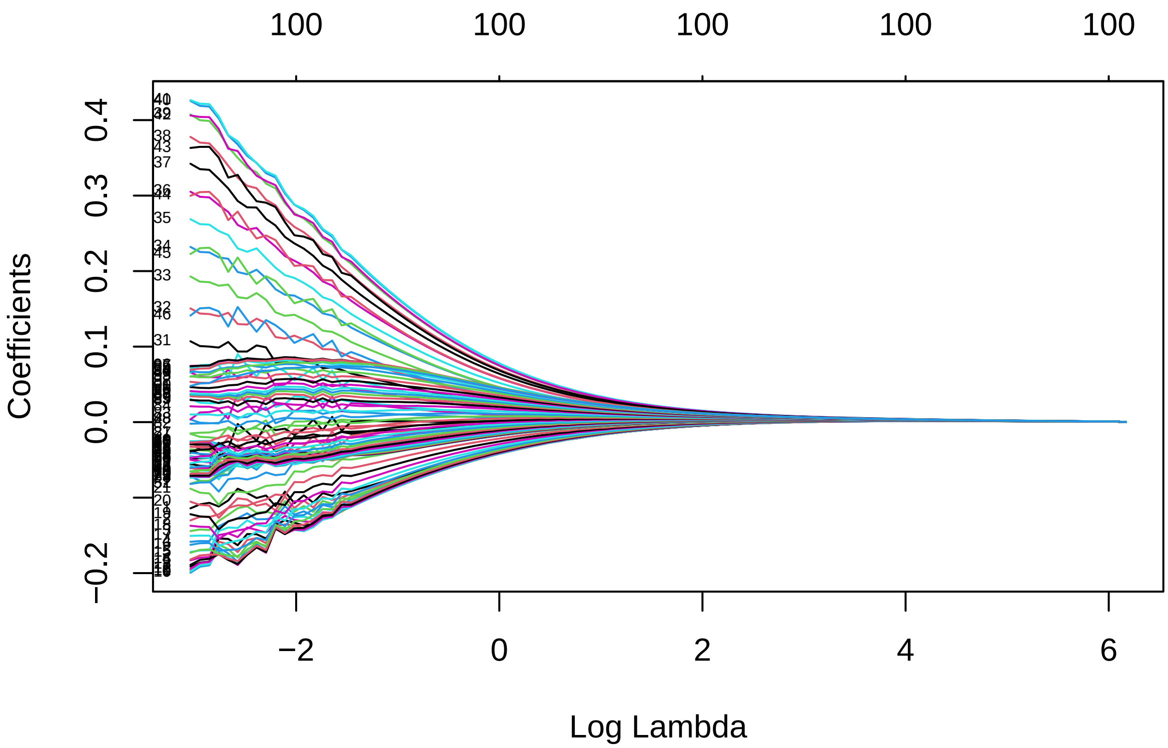
4. Degrees of Freedom Vs Lambda



As the value of degrees of freedom increases lambda value decreases thus making the model more complex as expected We can observe that features contribute more to the model with lower values of lambda.The Degrees of freedom represent the number of non-zero coefficients (features) of the model.

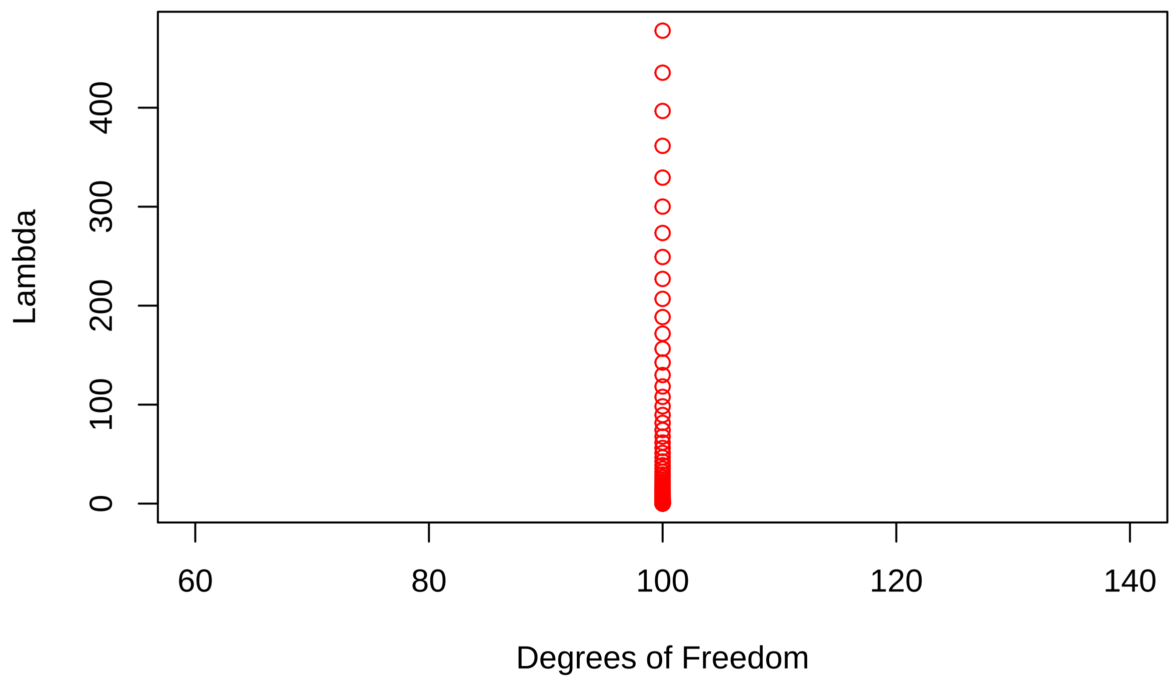
5. Ridge vs Lasso

Coeff VS log(Lambda)



The Ridge Regression model does not help in feature selection or elimination. However, the coefficients shrink as the value of lambda increases with an unchanged bais and the variance drops.

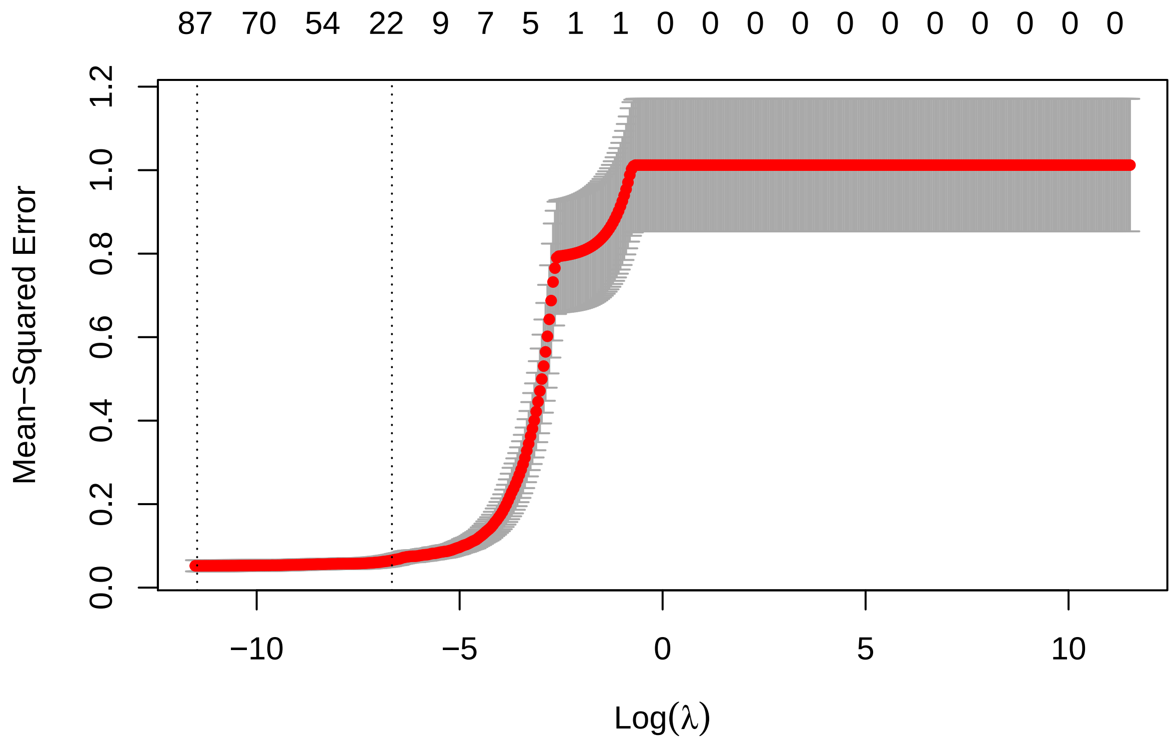
Degrees of Freedom VS Penalty Factor



Here all the 100 features are taken into account and there is no change in degrees of freedom with increasing value of lambda. Which is expected since ridge regression does not eliminate any features but shrinks the coefficients to a smaller value and not to zero.

Thus, the Degrees of Freedom represent all the coefficients for smaller values of lambda.There is a noticeable drop in more features corresponding to higher values of lambda which indicates irrelevance of features to the prediction as lambda value increases.

6. Cv and Optimal Model



From the above plot we can see that the mean squared error decreases as the value of lambda decreases.

```
## [1] 0.001271184
```

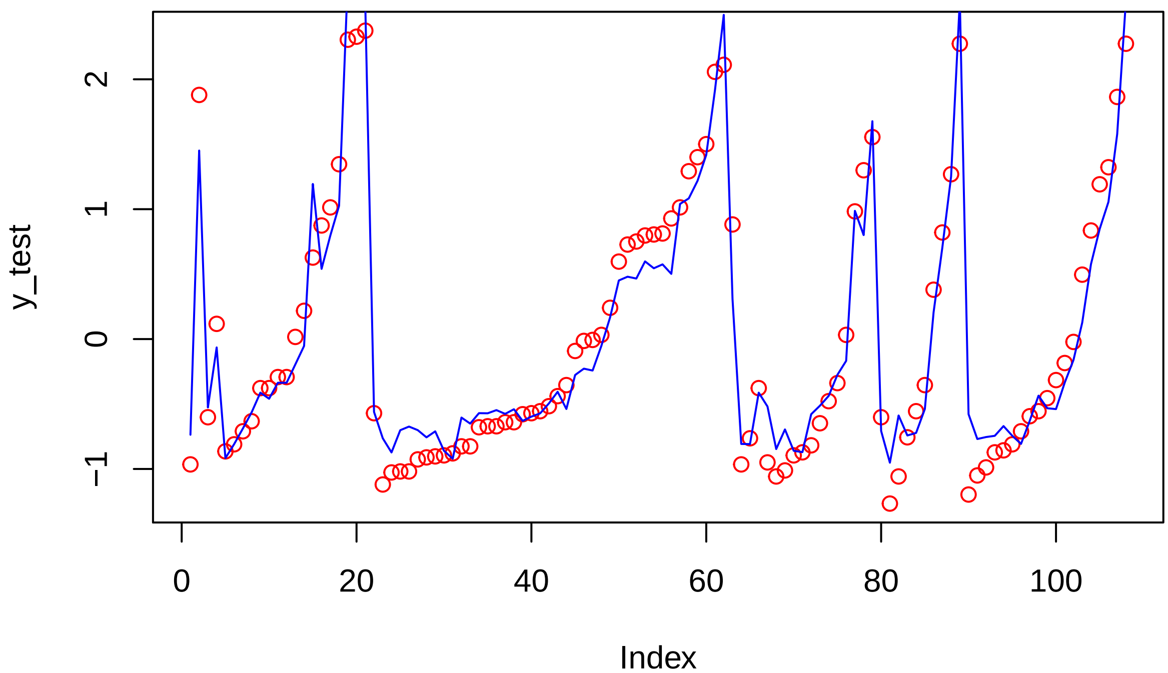
```
## [1] 19
```

19/100 variables were chosen from the Lasso Regression Model

The optimal value of lambda is not statistically significant that log(λ)=-2. This is because as the value of λ approaches -2 we see an increase in the value of RMSE which shows that the error increases as λ increases.

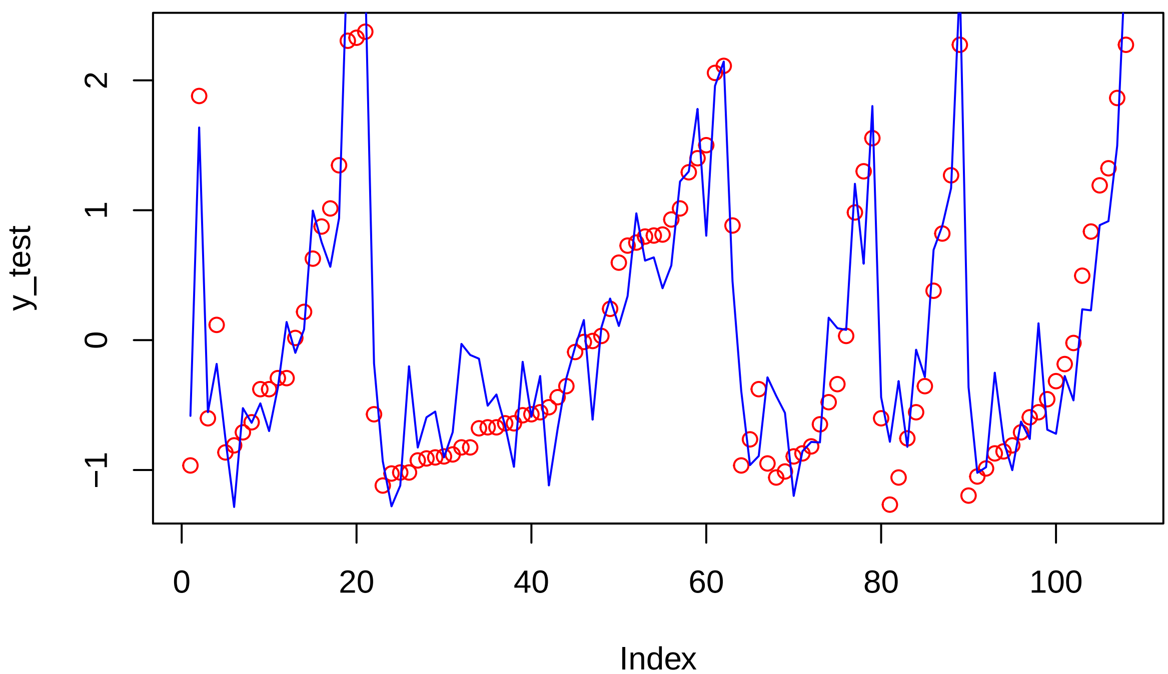
```
## The coefficient of Determination:  0.3218506
##                              MSE:  4.765318e-34
```

Observed vs Predicted for optimal λ values



The model seems to gives us promising results. The predictions are good and the variances has been minimized.Compared to the Linear Regression model in 2 we see a better fit and a significant reduction in the MSE values.

7. Data Generation



The Generated data does not fit well when compared to our optimal lasso model from 6. Hence I would say that Lasso Regression isn't suitable for data generation

Appendix: All code for this report

```r
knitr::opts_chunk$set(echo = TRUE)

# Set working directory

library(ggplot2)
library(kknn)
library("ggpubr")

# read the data file and process names and target.
opt =
  "https://raw.githubusercontent.com/TheClassyPenguin/MachineLearningLab1/main/optdigits.csv"
data <- read.csv(url(opt))
data[,ncol(data)] <- data.frame(sapply(data[,ncol(data)], as.character),
                                stringsAsFactors = TRUE)
names(data) <- c(seq(1:(ncol(data)-1)),"target")

# Split the data into training/validation/test (50/25/25), need to separate our target.65th column

n <- dim(data)[1]
set.seed(12345)
id <- sample(1:n, floor(n*0.5))
train_set <- data[id,]
id1 <- setdiff(1:n, id)
set.seed(12345)
id2 <- sample(id1, floor(n*0.25))
valid_set <- data[id2,]
id3 <- setdiff(id1,id2)
test_set <- data[id3,]
# Functions needed to solve the exercise
# Plotting function
heatmapping <- function(M, row){
  mapper <- heatmap(matrix(unlist(M[row,-65]), nrow=8), Colv = NA, Rowv = NA)
  return(mapper)
}
# Missclassification rate
missclass <- function(x, x1){
  return(1-(sum(diag(table(x, x1)))/sum(table(x,x1))))
}
# To binary function
to_binary <- function(i){
  b <- rep(0,10)
  b[i+1] <- 1
  return(I(b))
}

# k nearest 30 - Test
optdigits_kknn <- kknn(formula = train_set[,ncol(train_set)] ~ .,
              train = train_set, test = test_set, k = 30, kernel = "rectangular" )
model_fit <- fitted.values(optdigits_kknn)

# Confusion matrix
```

```r
conf_matrix <- table(test_set[,65], model_fit)
cat("Confusion matrix for the test data")
print(conf_matrix)

missclass_rate_test <- missclass(test_set[,65], model_fit)
cat('\ The missclassification rate for the test data is: ', missclass_rate_test)

# K nearest 30 - Train
optdigits_kknn_train <- kknn(formula = train_set[,ncol(train_set)] ~ .,
            train = train_set, test = train_set, k = 30, kernel = "rectangular" )
model_fit_train <- fitted.values(optdigits_kknn_train)

# Confusion matrix (training)
conf_matrix_train <- table(train_set[,65], model_fit_train)
cat("Confusion matrix for the training data.")
print(conf_matrix_train)

# Missclassification rate training
missclass_rate_train <- missclass(train_set[,65], model_fit_train)
cat("\ The missclassification rate for the training data is: ", missclass_rate_train)
#Probabilities

train_prob <- data.frame(optdigits_kknn_train$prob)
train_prob$target <- train_set$target
train_prob$fit <- optdigits_kknn_train$fitted.values

# Separating 8s

target_8 <- train_prob[train_prob$target == 8,]
fitted_8 <- target_8[target_8$fit == 8,]

# Best
indices_best <- as.numeric(row.names(fitted_8[order(-fitted_8[,9]),][1:2,]))

# Worse
indices_worst <- as.numeric(row.names(fitted_8[order(fitted_8[,9]),][1:3,]))

# Plot results
# 2 best
heatmap(t(matrix(unlist(train_set[indices_best[1],-65]), nrow=8)), Colv = NA, Rowv = NA)
heatmap(t(matrix(unlist(train_set[indices_best[2],-65]), nrow=8)), Colv = NA, Rowv = NA)

# 3 worst
heatmap(t(matrix(unlist(train_set[indices_worst[1],-65]), nrow=8)), Colv = NA, Rowv = NA)
heatmap(t(matrix(unlist(train_set[indices_worst[2],-65]), nrow=8)), Colv = NA, Rowv = NA)
heatmap(t(matrix(unlist(train_set[indices_worst[3],-65]), nrow=8)), Colv = NA, Rowv = NA)
# K-nearest neighbor classifier for K = 1,2,...,30.

k_classifier_valid <- c()
k_classifier_train <- c()


#This may take some time to compute
```

```r
for (i in 1:30){
optdigits_kknn_valid <- kknn(formula = train_set[,65] ~ ., train = train_set, test = valid_set, k = i, k
optdigits_kknn_train2 <- kknn(formula = train_set[,65] ~ ., train = train_set, test = train_set, k = i,

k_classifier_valid[i] <- missclass(valid_set[,65], fitted.values(optdigits_kknn_valid))
k_classifier_train[i] <- missclass(train_set[,65], fitted.values(optdigits_kknn_train2))
}

# plotting

Df <- data.frame("validation" = k_classifier_valid, "training" = k_classifier_train, "k" = 1:30)
plot1 <- ggplot()+
  geom_line(aes(x = Df$k, y = Df$validation, colour = "blue"))+
  geom_line(aes(x = Df$k, y = Df$training, colour = "red"))+
  ylab("Misclassification rate")+ xlab("k")+
  scale_color_manual(name = "Missclassification", labels = c("Validation set", "Training set"), values =

print(plot1)

best_k <- which(k_classifier_valid == min(k_classifier_valid))

cat("Best K is: ", best_k)

#Optimal K in Testing set

optdigits_kknn_test <- kknn(formula = train_set[,65] ~ ., train = train_set, test = test_set, k = best_k

test_fit <- fitted.values(optdigits_kknn_test)
conf_matrix_test <- table(test_set$target, test_fit)

missclass_rate_test2 <- missclass(x = test_set[,65], test_fit)
missclass_rate_test2
empirical_risk <- c()

# This is a long computation
for (i in 1:30){
  kknn_validation <- kknn(formula = train_set$target ~ .,train = train_set, test = valid_set, k = i, ker

  probs <- data.frame(kknn_validation$prob)
  probs$target <- valid_set$target
  probs$fit <- kknn_validation$fitted.values
  probs$binary <- (lapply(as.numeric(probs$target)-1, to_binary))

  #cross entropy loss
  for (j in 1:nrow(probs)){
    probs[j, "cross_entropy"] <- -sum(log(probs[j,1:10]+1e-15)* probs[[j, "binary"]])
  }

  empirical_risk[i] <- mean(probs$cross_entropy)
}


# Empirical risk for different k
```

```
Df2 <- data.frame("cross_entropy" = empirical_risk, "k" = 1:30)

plot2 <- ggplot(Df2, aes(x = k, y = cross_entropy, col = "red"))+
  geom_line()+
  theme(legend.position = "none")

print(plot2)

best_k2 <- which.min(empirical_risk)

cat("Best K is: ", best_k2)

csv_url =
  "https://raw.githubusercontent.com/TheClassyPenguin/MachineLearningLab1/main/parkinsons.csv"
raw_data = read.csv(url(csv_url), header=TRUE)

train_test_split = function(data, proportion){
  proportion_size = floor(proportion*nrow(data))

  train = data[sample(seq_len(nrow(data)), size = proportion_size),]
  test = data[-sample(seq_len(nrow(data)), size = proportion_size),]

  return(list(train=train,test=test))
}

exclude_columns = c("subject.","sex","age","test_time","total_UPDRS")#,"motor_UPDRS")

# Excludes from scaling
raw_data[,!names(raw_data) %in% exclude_columns] = apply(
  raw_data[,!names(raw_data) %in% exclude_columns],
  2,
  scale)

dataset = train_test_split(raw_data, 0.6)

exclude_x_columns = c("subject.","sex","age","test_time","motor_UPDRS","total_UPDRS")

x_train = dataset$train[,!names(dataset$train) %in% exclude_x_columns]
y_train = dataset$train[,names(dataset$train) == "motor_UPDRS"]

x_test = dataset$test[,!names(dataset$test) %in% exclude_x_columns]
y_test = dataset$test[,names(dataset$test) == "motor_UPDRS"]


loglikelihood = function(x, y, w, dispersion){
  n = dim(x)[1] #for every datapoint
  base = n*log(1/(abs(dispersion)*sqrt(2*pi)))
  exponent = -sum((y - t(t(w)%*%t(x)))**2) / (2*(dispersion**2))
  return(base + exponent)
}


ridge = function(par, x, y, lambda){
```

```
  m = dim(x)[2] #for every parameter
  w = as.matrix(par[1:(length(par)-1)])
  dispersion = par[length(par)]

  #Avoids 0 value without need for optimizer restriction
  if(isTRUE(all.equal(dispersion,0))){dispersion = dispersion + 0.0000001}

  exponent = - lambda * sum( w**2)  / (2 * (dispersion**2))
  base = m*log(sqrt(lambda) / (2 * (dispersion**2)))
  reg = base + exponent
  return(loglikelihood(x, y, w, dispersion)  + reg)
}

ridgeOpt = function(lambda, x, y){
  x=as.matrix(x)
  y=as.matrix(y)

  #Random Parameter initilization
  w = as.matrix(rnorm(dim(x)[2], mean = 0, sd = 0.01))
  sigma = runif(1, 0.001, 1)

  result = optim(par = c(w,sigma),
   ridge,
   x=x,
   y=y,
   lambda=lambda,
   method = "BFGS",
   control=list(fnscale=-1) #Maximizing instead of minimizing
   )
  return(result)
}

ridge_fit = ridgeOpt(1000, x_train, y_train)
ridge_fit


D = function(lambda, params, x, y){
  w = as.matrix(params)
  x = as.matrix(x)
  y = as.matrix(y)
  df = x %*% solve(t(x) %*% x + (lambda * diag(length(w)))) %*% t(x)
  return(sum(diag(df)))
}

D(lambda = 0.2,
 params = ridge_fit$par[1:(length(ridge_fit$par)-1)],
 x = x_train,
 y = y_train)


get_mean_square_error = function(params, x, y){
  w = as.matrix(params)
  x = as.matrix(x)
```

```
  y = as.matrix(y)
  mean_err = mean((y - t(t(w)%*%t(x)))**2)
  return(mean_err)
}

evaluate_performance = function(lambdas, x_train, y_train, x_test, y_test){
  train_results = c()
  test_results = c()
  for(i in 1:length(lambdas)){
    ridge_fit = ridgeOpt(lambdas[i], x_train, y_train)

    #Evaluating on train
    train_results = c(train_results,get_mean_square_error(
      params = ridge_fit$par[1:(length(ridge_fit$par)-1)],
      x_train,
      y_train))

    #Evaluating on test
    test_results = c(test_results, get_mean_square_error(
      params = ridge_fit$par[1:(length(ridge_fit$par)-1)],
      x_test,
      y_test))

  }
  result_frame = data.frame(train_results=train_results, test_results=test_results)
  rownames(result_frame) = lambdas
  return(result_frame)
}


lambdas = c(1, 100, 1000)
evaluate_performance(lambdas, x_train, y_train, x_test, y_test)


AIC = function(lambda, par, x, y){
  # AIC = -2(log-likelihood) + 2K
  loglik = ridge(par, x, y, lambda)
  params = par[1:(length(par)-1)]

  freedom = D(lambda, params, x, y)

  return(-2*loglik + 2*freedom)
}

evaluate_AIC = function(lambdas, x, y){
  AIC_results=c()

  for(i in 1:length(lambdas)){
    ridge_fit = ridgeOpt(lambdas[i], x, y)
    AIC_results=c(AIC_results,AIC(lambda=lambdas[i], par=ridge_fit$par, x=x, y=y))
  }

  result_frame = data.frame(AIC_SCORE=AIC_results)
```

```r
  rownames(result_frame) = lambdas

  return(result_frame)
}

complete_x = rbind(x_train, x_test)
complete_y = c(y_train, y_test)

evaluate_AIC(lambdas = lambdas, x = complete_x, y = complete_y)

#Splitting Data
tec =
  "https://raw.githubusercontent.com/TheClassyPenguin/MachineLearningLab1/main/tecator.csv"

data<-data.frame(read.csv(url(tec)))

n<-dim(data)[1]
set.seed(12345)
id<-sample(1:n ,floor(n*0.5))
train<-data[id,]
test<-data[-id,]

#Training a Linear Regression

model.train<-lm(Fat~ .-ï..Sample-Protein-Moisture,data=train)
summary(model.train)


#Train and Test Errors

train_pred<-predict(model.train ,  train)
train_mse<-mean((train$Fat-train_pred)^2)
train_rmse<-sqrt(train_mse)

test_pred<-predict(model.train , test)
test_mse<-mean((test$Fat-test_pred)^2)
test_rmse<-sqrt(test_mse)

errors<-cbind(train_rmse,test_rmse)
errors
plot(test$Fat ,lty=1.8,col="red")
lines(test_pred,type="l",col="blue")



#lasso

library(glmnet)

 x_train<-scale(train[,2:101])
 y_train<-train<-scale(train$Fat)

model1<-glmnet(as.matrix(x_train),y_train,family="gaussian" ,alpha=1)
```

```r
plot(model1,xvar="lambda",label=TRUE)


plot(log(model1$lambda),model1$df,col="red")

# Ridge Regression

set.seed(12345)
model2<-glmnet(as.matrix(x_train),y_train,family="gaussian" ,alpha=0)

plot(model2,xvar="lambda",label=TRUE)
plot(model2$df,model2$lambda,col="red",
     xlab="Degrees of Freedom",
     ylab="Lambda")

#optimal Lasso

set.seed(12345)
cv_model1<-cv.glmnet(as.matrix(x_train),y_train,family="gaussian" , lambda=10^seq(-5,5,length=500), alpl



plot(cv_model1)


lambda_optimal<-cv_model1$lambda.1se
lambda_optimal

#finding chosen features

features<-as.matrix(coef(cv_model1,cv_model1$lambda.1se))
optimal_features<-features[features!=0,]
length(optimal_features)




#Quality of Fit

x_test<-scale(test[,2:101])
y_test<-scale(test$Fat)

optimal_cv_lasso<-glmnet(as.matrix(x_test),y_test,alpha=1,lambda=cv_model1$lambda.1se)
pred_test_lasso<-predict(optimal_cv_lasso, newx = x_test,type="response")



test_mse<-(mean(pred_test_lasso-y_test)^2)
test_coefdet<-sum((pred_test_lasso-mean(y_test)^2))/sum((y_test-mean(y_test)^2))

cat("The coefficient of Determination: ",test_coefdet,"\n                                   MSE: ",test_mse)

pred_true<-data.frame(cbind(test$Fat,pred_test_lasso))
```

```r
colnames(pred_true)[1]="Fat"
colnames(pred_true)[2]="Pred"

plot(y_test ,lty=1.8,col="red")
lines(pred_test_lasso,type="l",col="blue")


#Data Generation

betas<-as.vector((coef(optimal_cv_lasso))[-1,])
res<-y_train-(x_train%*%betas)
stdev<-sd(res)

set.seed(12345)

target_values<-rnorm(length(y_test),pred_test_lasso,stdev)




plot(y_test ,lty=1.8,col="red")
lines(target_values,type="l",col="blue")
```