To elaborate, if we consider y as the target variable, and X as the set of predictors, then the coefficients of the regression model $\hat{y} = \beta_0 + \beta' X$ which estimates the target is selected such that the RSS, $\sum(\hat{y} - y)^2$ is minimum.

Now, in order to control the value of the coefficients, we can add an additional constraint such that the regression objective minimizes the RSS + λ (L2 norm of β), i.e. minimizes $\Sigma(\hat{y} - y)^2 + \lambda \Sigma \beta^2$.

Here, lambda (λ) is called as the regularization parameter and it controls the trade-off between minimizing the RSS and the L2 norm.

As with least squares, ridge regression tries to minimize the first factor i.e. RSS, however, the second factor called the shrinkage penalty is small when β is close to zero. The regularization parameter serves to control the relative impact of these two parameters. The L2 norm is applied to all the regression coefficients excluding the intercept.

When λ = 0, the ridge regression takes the form of least squares estimates. However, when λ = ∞, the impact of shrinkage penalty grows and the least squares estimates approaches to zero. Therefore, to choose the best value of λ, we need to depend upon the cross validation and choose the value of λ which has least cross-validation error.

Let us now regularize model 3 (over-fitted model) using ridge regression to minimize the effects of over-fitting. In other terms, ridge regression will control the variance of the model 3 by establishing a trade-off between variance and bias.

In the upcoming module we rely upon **glmnet** library to access the functionality of Ridge and further Lasso regularization.

Before we begin with to interpret the meaning of L2 regularization, let us build a general framework to be used ahead:

```
# Necessary Libraries
library(glmnet)
library(ggplot2)

# Reading data set
house_train <- read.csv("hp_train_old.csv")
house_test <- read.csv("hp_test.csv")

# Model 3 – Polynomial regression
model3 <- lm(Price~poly(Size,10,raw=T),house_train)
```

```
# Input data for Ridge regression
train_matrix <- poly(house_train$Size, 10, raw=F)
test_matrix <- poly(house_test$Size, 10, raw=F)
```

Moving ahead, as per the cost function of ridge regression, let us observe the implications which happens when we assign the value of λ = 0.

```
lam_0 = glmnet(train_matrix, house_train$Price, alpha = 0)
lam_0_pre <- predict(lam_0,
                     s = 0,           # Value for penalty parameter lambda
                     train_matrix)
```

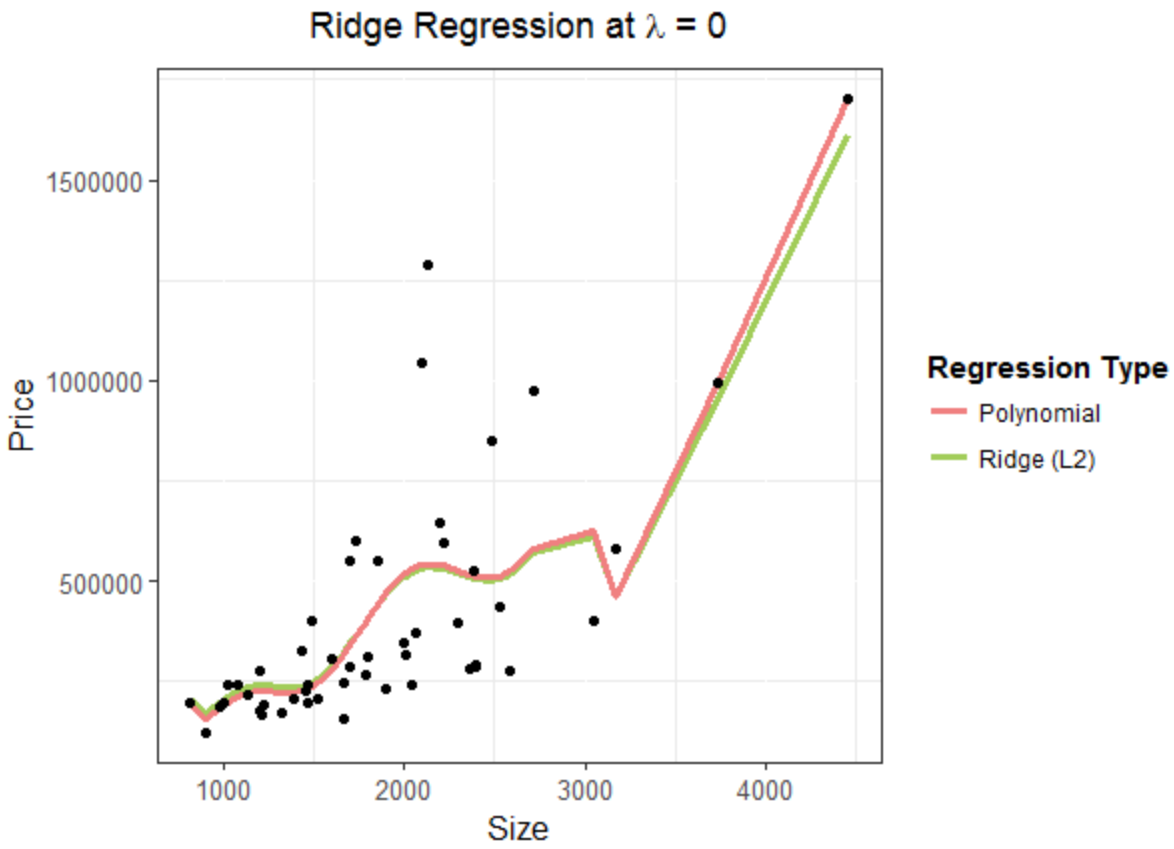
glmnet – function accepts a value for argument alpha in the range of [0, 1] apart from input and output data. Given table interprets the meaning of alpha values –

Alpha value	Regression Type
0	Ridge (L2)
1	Lasso (L1)
[0, 1] – All the decimal values between 0 and 1	Elastic Net

The value of lambda needs to be passed when we're about to do the prediction. The variable *lam_0_pre* consists of predicted output which arrives by passing the required value of lambda to penalty argument "s" (here 0).

Next, we plot a graph of model 3 fitted over training data along with our new model at λ = 0, code of which is given below followed with graph.

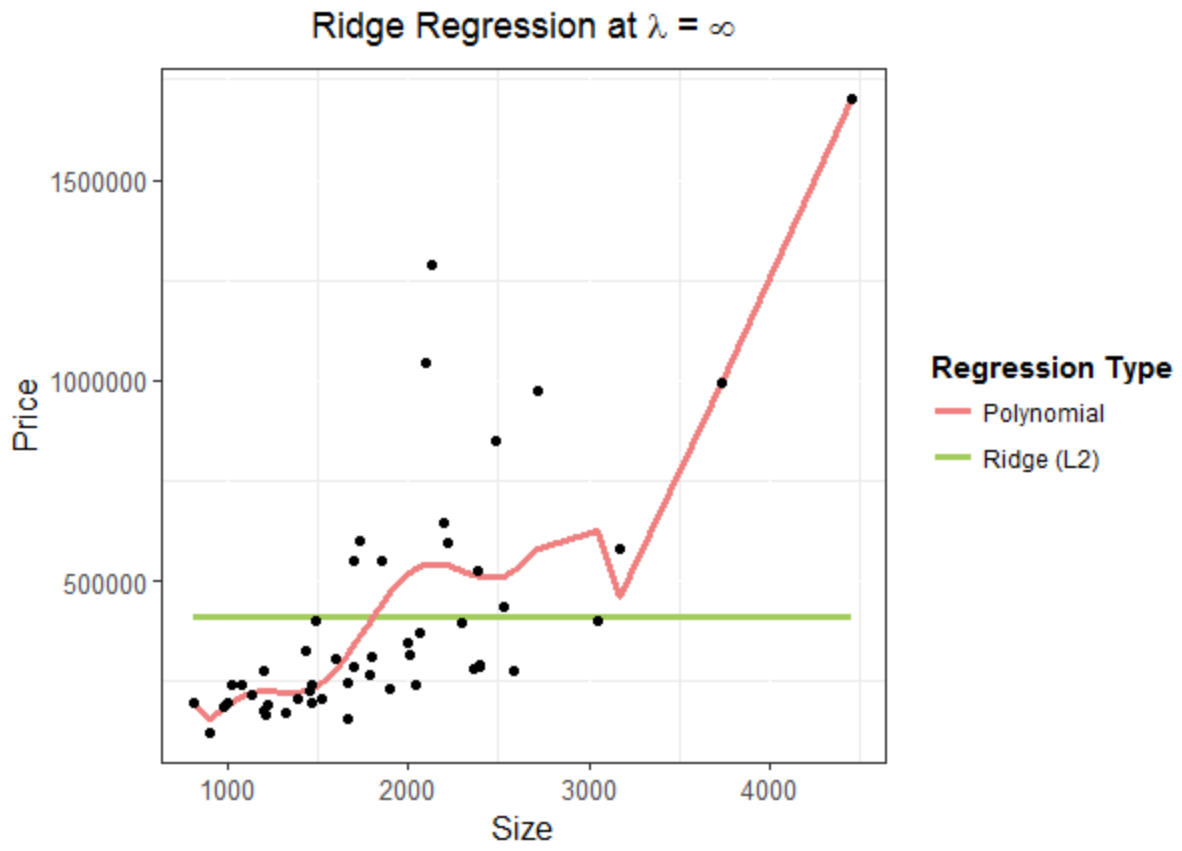
```
plot_func <- function(pre_val, title){
  ggplot() +
    geom_line(aes(x = house_train$Size, y = pre_val, col = 'Ridge (L2)'), size = 1.2) +
    geom_line(aes(x = house_train$Size, y = fitted(model3), col = 'Polynomial'), size = 1.2) +
    geom_point(aes(x = house_train$Size, y = house_train$Price), size = 1.5) +
    labs(col = 'Regression Type') +
    scale_color_manual(values = c("Ridge (L2)" = "darkolivegreen3",
                                  "Polynomial" = "lightcoral")) +
    title +
    xlab("Size") + ylab("Price") +
    theme_bw() +
    theme(panel.grid.major = element_line(colour = "white"),
          axis.text = element_text(size = 10),
          axis.title = element_text(size = 12),
          plot.title = element_text(hjust = 0.5),
          legend.title = element_text(face = 'bold'))
}

title <- ggtitle(expression(paste("Ridge Regression at ", lambda," = ", 0)))
plot_func(lam_0_pre, title)
```



The L2 fitted curve follows an estimated curve of original polynomial regression. Since at λ = 0 the ridge regression cost function equation makes the shrinkage penalty term equals to zero and hence follows the least squares estimates.

Moving ahead with a pretty high value (substitution for infinity) of λ and observe the result:

```
lam_inf = glmnet(train_matrix, house_train$Price, alpha = 0)
lam_inf_pre <- predict(lam_inf, s = 9999999999, train_matrix)

title <- ggtitle(expression(paste("Ridge Regression at ", lambda," = ", infinity)))
plot_func(lam_inf_pre, title)
```



This leads to suppressing all the least squares estimates.

Therefore, the challenge is to find a best value of λ that fits the data without overfitting. This can be solved using CV. We prefer to choose a value of λ where the mean cross validation error is minimum. For instance, we use 10-fold cross validation to choose the best value of λ as shown:

```
lam_best <- cv.glmnet(train_matrix, house_train$Price, alpha = 0)
lam_best_pre <- predict(lam_best,s = lam_best$lambda.min, train_matrix)
```

The function *cv.glmnet* can go up to 100 values of lambda, however not mandatory, it can also stop given there isn't any radical change in any consecutive "*%Dev*" which can be observed for given model as shown below for top 10 lambda values:

```
lam_best$glmnet.fit

# Call:  glmnet(x = train_matrix, y = house_train$Price, alpha = 0)
#
#       Df      %Dev    Lambda
#  [1,] 10 1.257e-36 226400000
#  [2,] 10 1.891e-03 206300000
#  [3,] 10 2.075e-03 188000000
#  [4,] 10 2.277e-03 171300000
#  [5,] 10 2.498e-03 156000000
#  [6,] 10 2.741e-03 142200000
#  [7,] 10 3.007e-03 129600000
#  [8,] 10 3.299e-03 118000000
```

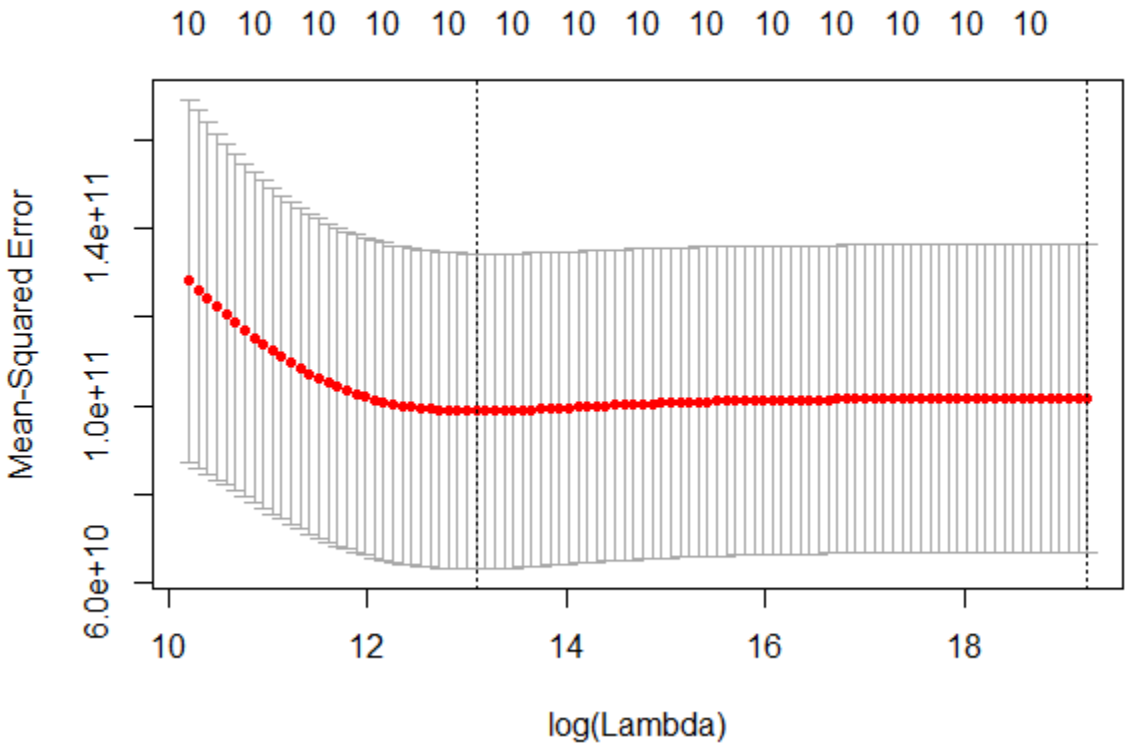
```
#  [9,] 10 3.619e-03 107600000
# [10,] 10 3.971e-03  98000000
```

lambda.min gives the best value of lambda with the minimum mean cross-validated error. We can also select a lambda value which gives a regularized model such that its error lies within the range of one standard error of the minimum using *lambda.1se*.

```
log(lam_best$lambda.min, base = exp(1))
# 13.09

log(lam_best$lambda.1se, base = exp(1))
# 19.23
```

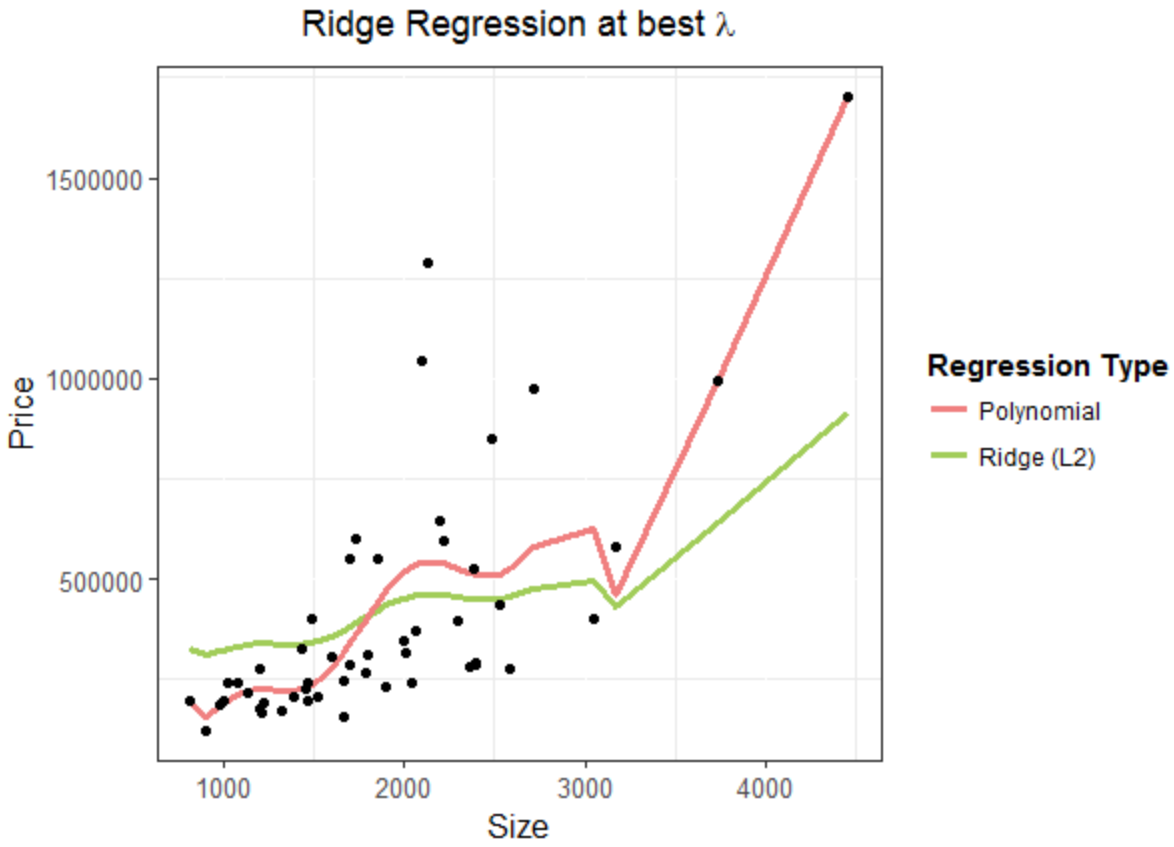
These two values of lambdas (marked by two vertical dotted lines) can be visualized in the figure given below which plots the *lam_best* model for generated cross-validation curve along with upper and lower standard deviation curve for lambda sequence (error bars).

```
plot(lam_best)
```



We can now visualize our final ridge regression model at its best lambda.

```
title <- ggtitle(expression(paste("Ridge Regression at best ", lambda )))
plot_func(lam_best_pre, title)
```



Ridge regression reduces the coefficient values but don't make them completely zero. Within our established model we can observe which coefficients are subtly more important by finding the values of β corresponding to the *lambda.min*.

```
# Building model without orthogonal polynomials
train_matrix <- poly(house_train$Size,10,raw = T)
lam_best <- cv.glmnet(train_matrix, house_train$Price, alpha = 0)
lam_best_pre <- predict(lam_best,s = lam_best$lambda.min, train_matrix)

# Locating index of min lambda
index <- which(lam_best$lambda == lam_best$lambda.min); index

# Forming a matrix of all 10 coefficients corresponding to each lambdas
beta <- t(matrix(lam_best$glmnet.fit$beta, ncol = 100, byrow = F))
beta[index,] # Beta values
lam_best$glmnet.fit$a0[index] # Intercept, unchanged (with orthogonal polynomial)
```

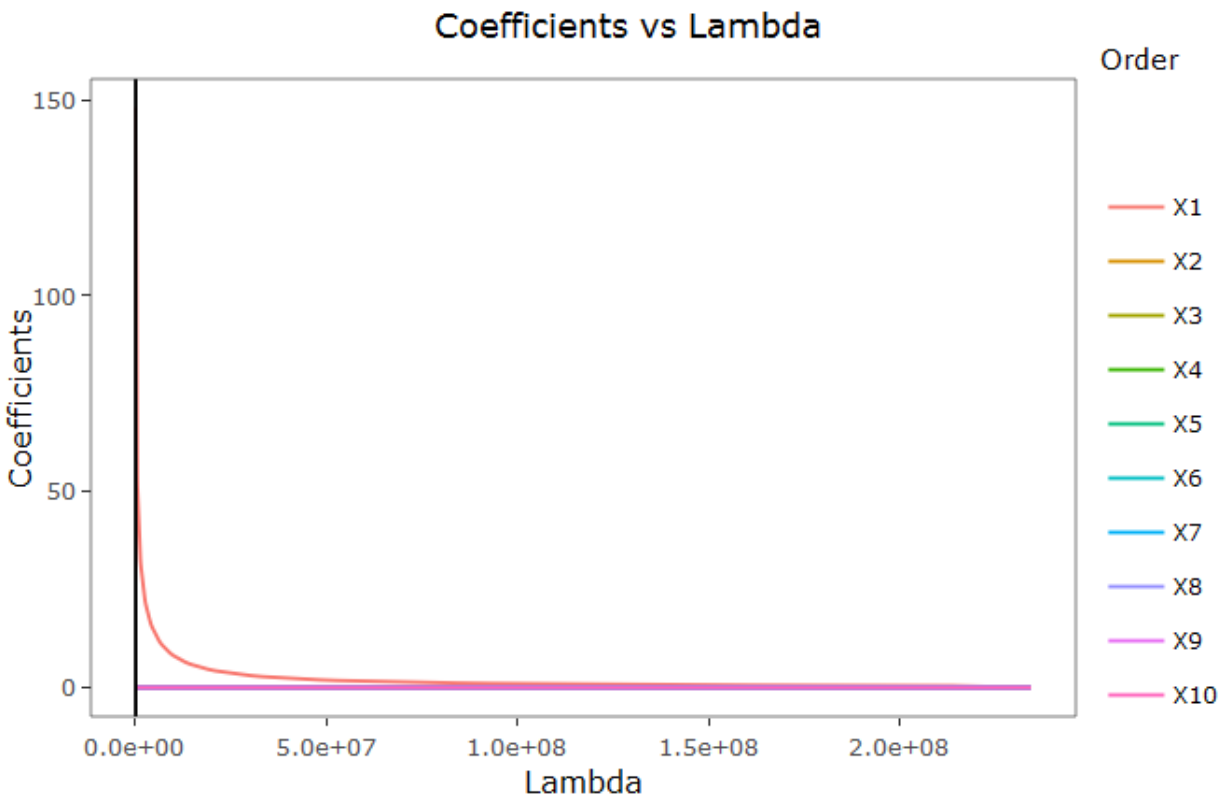
The table compares the coefficients of orthogonal polynomial model 3 as well as orthogonal polynomial ridge regression at its best lambda.

Coefficients	**Model 3**	**Ridge Regression**
Intercept	411109 (raw = F)	411109 (raw = F)
β_1	-2.186e+05	1.319185e+02
β_2	3.085e+02	1.436669e-02
β_3	-8.229e-02	1.231186e-06

β_4	-2.620e-04	6.968691e-11
β_5	3.627e-07	1.030027e-14
β_6	-2.290e-10	5.803875e-18
β_7	8.314e-14	2.396669e-21
β_8	-1.777e-17	7.668471e-25
β_9	2.081e-21	2.124906e-28
β_{10}	-1.030e-25	5.405537e-32

As can be inferred from the table in ridge regression coefficient β_1 dominates whereas other coefficient values diminishes rapidly. The same inference can even be obtained visually as shown –

```
library(tidyverse)
library(plotly)

lambda_col <- lam_best$lambda

# Adding lambda values after all coeff. columns
temp <- data.frame(t(matrix(lam_best$glmnet.fit$beta, ncol = 100, byrow = F))[1:99,], lambda_col)

temp <- temp %>%  gather(lambda_col)
names(temp) <- c('Lambda','Order','Coefficients')

# Reordering levels to bring X10 in the end on legend
temp$Order <- as.factor(temp$Order)
levels(temp$Order) <- c('X1',  'X2', 'X3', 'X4', 'X5', 'X6', 'X7', 'X8', 'X9','X10')

# save below plot in a variable called "graph" for better Viz.
graph <- ggplot(temp) +
  geom_line(aes(Lambda,Coefficients,col = Order)) +
  geom_vline(xintercept = lam_best$lambda.min) +
  theme_bw() +
  ggtitle("Coefficients vs Lambda") +
  theme(panel.grid.major = element_line(colour = "white"),
        axis.text = element_text(size = 10),
        axis.title = element_text(size = 12),
        plot.title = element_text(hjust = 0.5),
        legend.title = element_text(face = 'bold'))

ggplotly(graph)
```



Here, the vertical black line is drawn at *lambda.min*. We can now observe that coefficient X1 ($\boldsymbol{\beta_1}$) is the only dominating term at given lambda rest are all approx. to zero.

With the specific regularized value of lambda in hand, we can now compare the RMSE of train and test data for both of the models.

```
train_model <- predict(lam_best, s = lam_best$lambda.min, train_matrix)
rmse_train <- sqrt(mean((train_model - house_train$Price)^2))
rmse_train # 204739.8

test_model <- predict(lam_best, s = lam_best$lambda.min, test_matrix)
rmse_test <- sqrt(mean((test_model - house_test$Price)^2))
rmse_test # 304099.6


rmse_train_orig <- sqrt(mean((model3$residuals)^2))
rmse_train_orig # 193216.9

rmse_test_orig <- sqrt(mean((predict(model3,house_test[,-3]) - house_test$Price)^2))
rmse_test_orig # 133415346
```

We can infer from the given RMSE table that there is quite a remarkable improvement in the regularized model when compared to an over-fitted model.

RMSE	**Regularized**	**Un-regularized**
Train Data	204739.8	193216.9
Test Data	304099.6	133415346

L1 regularization

So far we are able to realize the potential of ridge regression in reducing the overfitting. However, we observed from our last section that all of the coefficients are still intact in the model irrespective of their importance which abides to an increased complexity with no better results.

The Lasso regularization chooses the predictor(s) along with their important coefficient(s) which has/have relatively high significant effect on prediction with increasing values of lambda. The lasso regularization performs L1 norm as compared with L2 regularization as shown below:

$$\sum(\hat{y} - y)^2 + \lambda \sum \beta$$

As compared with ridge regression, with the sufficient large value of λ, lasso does forces some of the coefficient estimates to equals zero. Hence, acting as a variable selector and reducing the complexity of the overall model.

To depict the variable selection property of lasso, we consider IMDB dataset (selecting only 15 attributes of whole). Making it again a regression problem, we have to predict the IMDB score based on given 14 predictors. To begin with let's split the dataset into training and testing halves and apply linear regression.

```
orig <- read.csv("imdb.csv")

# Splitting original data into training and testing parts
smp_size <- floor(0.7 * nrow(orig))
set.seed(42)
train_ind <- sample(seq_len(nrow(orig)), size = smp_size)
train_data <- orig[train_ind, ]
test_data <- orig[-train_ind, ]

# Building linear regression model
model_lr <- lm(formula = imdb_score~., data = train_data)
```

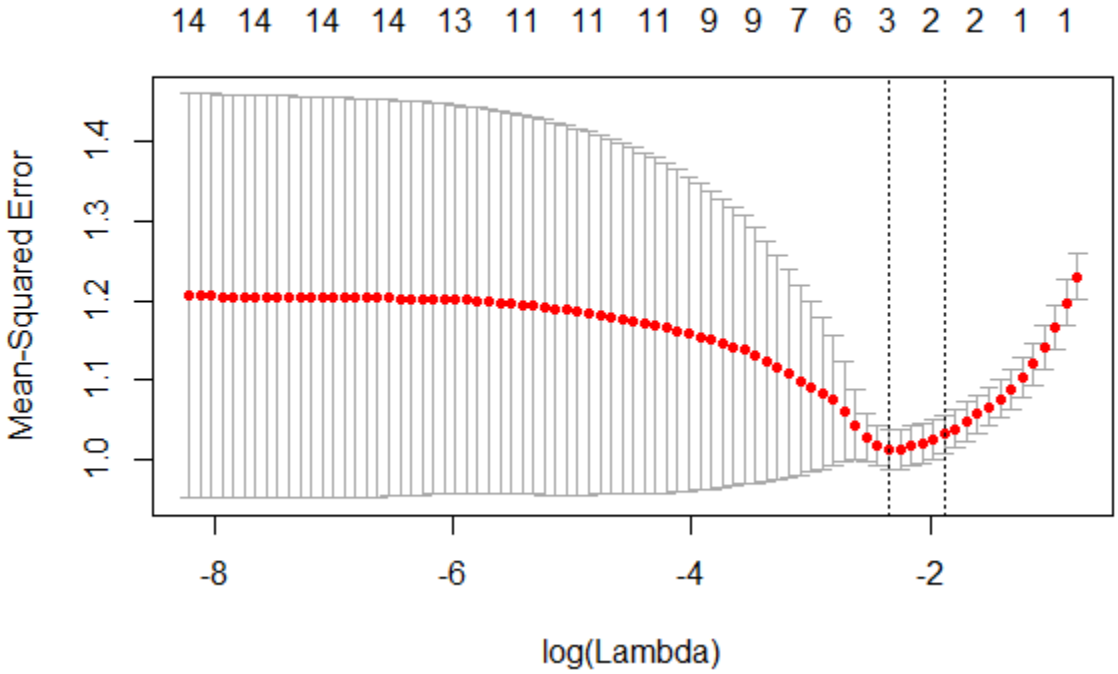
Next, we build our L1 regularization model, the procedure of which is exactly similar to that of L2 regularization except passing the value of argument *alpha* equal to 1.

```
library(glmnet)
set.seed(42)

# Converting predictors class to matrix
l1_mat <- as.matrix(train_data[-15])
model_l1 = cv.glmnet(l1_mat,
                     train_data$imdb_score,
                     alpha = 1)
```

Again, the best value for lambda can be located using *lambda.min* as well as through visualization –

```
log(model_l1$lambda.min, base = exp(1))
# -2.349933

log(model_l1$lambda.1se, base = exp(1))
# -1.884765
```

```
plot(model_l1)
```



L1 regularization is known to remove the unimportant predictors from the built model. Let us observe what all predictors are not needed for our model.

```
best_pred_coef <- predict(model_l1,s = model_l1$lambda.min,
                          l1_mat,type = "coefficients")
best_pred_coef
```

The table list the output of *best_pred_coef* variable which clearly depicts that there are only 3 important predictors at the best value of lambda namely –

- num_critic_for_reviews
- duration
- num_voted_users

Therefore, rest of the 11 predictors can be ignored by still sustaining the RMSE (to be dealt later).

Predictors	Coefficients
(Intercept)	5.900897e+00
num_critic_for_reviews	8.062270e-05
duration	3.315895e-03
director_facebook_likes	0
actor_3_facebook_likes	0
actor_1_facebook_likes	0
gross	0
num_voted_users	2.376529e-06

cast_total_facebook_likes	0
facenumber_in_poster	0
num_user_for_reviews	0
budget	0
actor_2_facebook_likes	0
aspect_ratio	0
movie_facebook_likes	0

The same conclusion can be derived visually if we plot the coefficients of all the predictors against the increasing values of lambda and mark the threshold at best value of lambda.

```
library(ggplot2)
library(plotly)

lamb_all <- predict(model_l1,s = model_l1$lambda,l1_mat,type="coefficients")
df <- cbind(as.data.frame(as.matrix(t(lamb_all))),as.data.frame(model_l1$lambda))
l1 = df[,c(-1)] # Excluding the Intercept
colnames(l1)[15] <- 'lambda'

l1 <- l1 %>%
  gather(column,value,-lambda)

graph <- ggplot(l1) + geom_line(aes(lambda,value, col = column)) +
  geom_vline(xintercept = model_l1$lambda.min) +
  ggtitle("L1 variable selection") + xlab("Lambda") + ylab("Coefficients") +
  theme_bw() +
  theme(panel.grid.major = element_line(colour = "white"),
        axis.text = element_text(size = 10),
        axis.title = element_text(size = 12),
        plot.title = element_text(hjust = 0.5),
        legend.title = element_text(face = 'bold'))

ggplotly(graph)
```

Figure 1 shows the actual plot and figure 2 shows the hover plot, highlighting predictors values at lambda threshold.

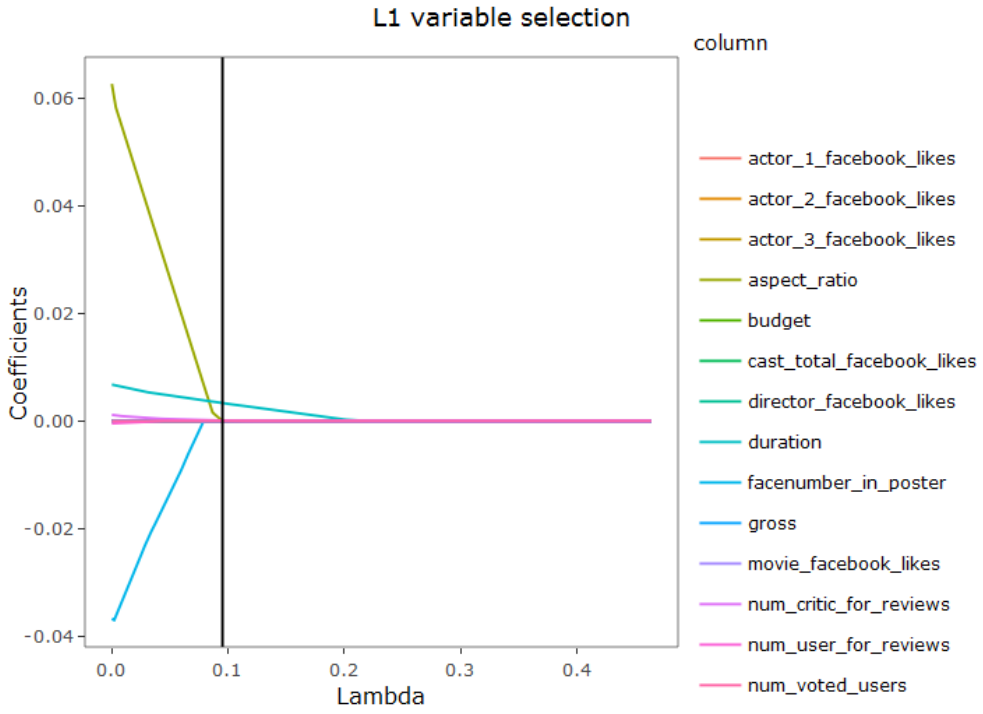

L1 variable selection
column
Coefficients
Lambda
0.06
0.04
0.02
0.00
-0.02
-0.04
0.0
0.1
0.2
0.3
0.4
actor_1_facebook_likes
actor_2_facebook_likes
actor_3_facebook_likes
aspect_ratio
budget
cast_total_facebook_likes
director_facebook_likes
duration
facenumber_in_poster
gross
movie_facebook_likes
num_critic_for_reviews
num_user_for_reviews
num_voted_users

Fig.1

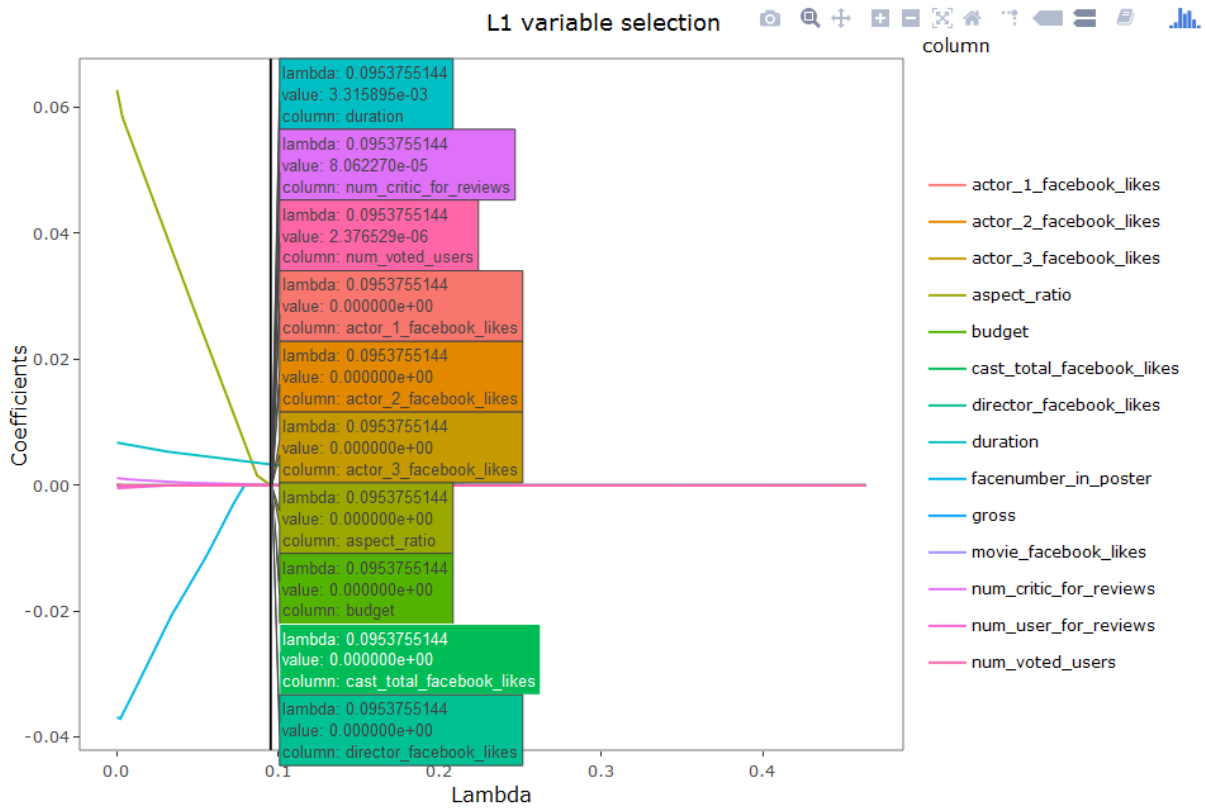

L1 variable selection
column
Coefficients
Lambda
lambda: 0.0953755144
value: 3.315895e-03
column: duration
lambda: 0.0953755144
value: 8.062270e-05
column: num_critic_for_reviews
lambda: 0.0953755144
value: 2.376529e-06
column: num_voted_users
lambda: 0.0953755144
value: 0.000000e+00
column: actor_1_facebook_likes
lambda: 0.0953755144
value: 0.000000e+00
column: actor_2_facebook_likes
lambda: 0.0953755144
value: 0.000000e+00
column: actor_3_facebook_likes
lambda: 0.0953755144
value: 0.000000e+00
column: aspect_ratio
lambda: 0.0953755144
value: 0.000000e+00
column: budget
lambda: 0.0953755144
value: 0.000000e+00
column: cast_total_facebook_likes
lambda: 0.0953755144
value: 0.000000e+00
column: director_facebook_likes
0.06
0.04
0.02
0.00
-0.02
-0.04
0.0
0.1
0.2
0.3
0.4
actor_1_facebook_likes
actor_2_facebook_likes
actor_3_facebook_likes
aspect_ratio
budget
cast_total_facebook_likes
director_facebook_likes
duration
facenumber_in_poster
gross
movie_facebook_likes
num_critic_for_reviews
num_user_for_reviews
num_voted_users

Fig. 2

As can be inferred from the graph, only 3 predictors have value not equal to zero at lambda threshold which verifies the output of *best_pred_coef* variable.

Let us now work on the RMSE values of the original as well as the L1 regularization model. To do that we calculate individual RMSE for both training and testing data respectively.

```
# Original Model RMSE
rmse_train_lr <- sqrt(mean((model_lr$residuals)^2))
rmse_train_lr # 0.9649056

rmse_test_lr <- sqrt(mean((predict(model_lr,test_data) - test_data$imdb_score)^2))
rmse_test_lr # 1.0331

# L1 Model RMSE
pred_train <- predict(model_l1,s = model_l1$lambda.min,l1_mat)
pred_test <- predict(model_l1,s = model_l1$lambda.min,l1_pre)

rmse_train_l1 <- sqrt(mean((pred_train - train_data$imdb_score)^2))
rmse_train_l1 # 1.003375

rmse_test_l1 <- sqrt(mean((pred_test - test_data$imdb_score)^2))
rmse_test_l1 # 1.063083
```

As can be inferred from the RMSE table, both the models provide similar RMSE values, however L1 model does so with significant improvement in computation time and memory utilization by building model as a function of just three predictors.

RMSE	**Regularized**	**Un-regularized**
Train Data	1.00	0.96
Test Data	1.06	1.03

Ridge and Lasso on Classification Problem

Consider the scenario 2 of spam dataset where a particular email is classified either spam or not. We have built 2 models previously, one with 5 predictors and other with all 57 predictors.

```
# Reading data
train_data <- read.csv("spam_train_data.csv")
test_data <- read.csv("spam_test_data.csv")

# Building models
model1 <- glm(spam~word_freq_free+word_freq_credit+word_freq_receive+
              word_freq_money+capital_run_length_total,
            train_data,family = binomial(link = "logit"))
model2 <- glm(spam~.,train_data,family = binomial(link = "logit"))
```

To continue, we accept model2 and deploy Ridge as well as Lasso regularization on it.

```
library(glmnet)

# L2 Regularization
train_mat <- as.matrix(train_data[-58])
l2_model <- cv.glmnet(train_mat, train_data$spam, alpha=0)
l2_model
```

```
# L1 Regularization
l1_model <- cv.glmnet(train_mat, train_data$spam, alpha=1)
l1_model
```

For *glmnet*, all the concepts for L2 and L1 on classification problems remains the same as that of regression. Therefore, we can find best lambda for either method using *lambda.min* and number of required predictors (in case of L1) using *predict* function and passing type as *coefficients*.

We can now compare the accuracies of the build models on model2.

```
# Confusion matrix on train data for Model2
m2_train <- ifelse(predict(model2,type = "response",train_data) >= 0.5,1,0)
m2_cm <- addmargins(table(m2_train,train_data$spam))
train2_acc <- (m2_cm[1] + m2_cm[5]) / m2_cm[9] * 100
train2_acc #100

# Confusion matrix on test data for Model2
m2_test <- ifelse(predict(model2,type = "response",test_data) >= 0.5,1,0)
m2_cm <- addmargins(table(m2_test,test_data$spam))
test2_acc <- (m2_cm[1] + m2_cm[5]) / m2_cm[9] * 100
test2_acc # 66

# L2 confusion matrix on train data for Model2
m2_train <- ifelse(predict(l2_model,type = "response",train_mat) >= 0.5,1,0)
m2_cm <- addmargins(table(m2_train,train_data$spam))
l2_train2_acc <- (m2_cm[1] + m2_cm[5]) / m2_cm[9] * 100
l2_train2_acc # 82.60

# L2 confusion matrix on test data for Model2
test_mat <- as.matrix(test_data[-58])
m2_test <- ifelse(predict(l2_model,type = "response", test_mat) >= 0.5,1,0)
m2_cm <- addmargins(table(m2_test,test_data$spam))
l2_test2_acc <- (m2_cm[1] + m2_cm[5]) / m2_cm[9] * 100
l2_test2_acc # 78

# L1 confusion matrix on train data for Model2
m2_train <- ifelse(predict(l1_model,type = "response", train_mat) >= 0.5,1,0)
m2_cm <- addmargins(table(m2_train,train_data$spam))
l1_train2_acc <- (m2_cm[1] + m2_cm[5]) / m2_cm[9] * 100
l1_train2_acc # 80.86

# L1 confusion matrix on test data for Model2
m2_test <- ifelse(predict(l1_model,type = "response", test_mat) >= 0.5,1,0)
m2_cm <- addmargins(table(m2_test,test_data$spam))
l1_test2_acc <- (m2_cm[1] + m2_cm[5]) / m2_cm[9] * 100
l1_test2_acc # 80
```

The table shows the improvement in the model using L2 and L1 regularization. An un-regularized model shows worst synchronization in training and testing accuracy, however with the deployment of L2 and further L1, we observe that there is a lot of improvement in the accuracy of training and testing data.

Accuracy	**Un-Regularized Model 2**	**L2 Regularization**	**L1 Regularization**
Train Data	100	82.60	80.86
Test Data	66	78	80

Model selection – which regularization method is best?

Ridge regression is useful when almost every predictor is important (even slightly) for the model. However, it is better to choose lasso regression when just few predictors are worthy enough to build the model.

Let us understand it through two illustrations (reference: ISLR, PP 223 [1] [2]). Consider first scenario in which all the given predictors are related to the response so that none can be equaled to zero. On the other hand, consider another scenario in which response is a function of few out of all the predictors.

The Bias-Variance plot for each scenario is shown below:

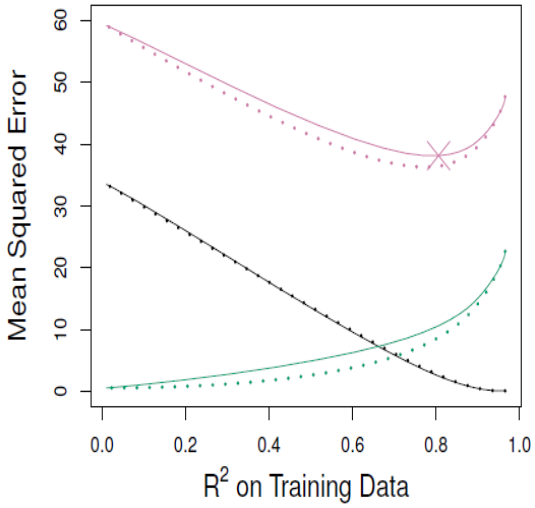


Fig.1

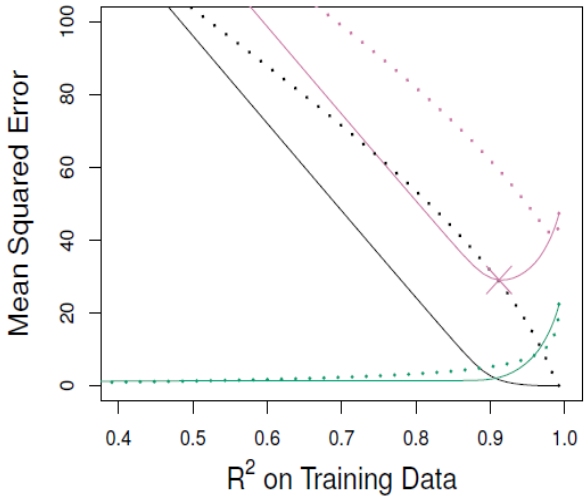


Fig.2

In both the figures, green lines indicate variance; black lines, squared bias and red lines, test MSE. The dotted lines resemble data for ridge regression and solid lines resemble data for lasso. Figure 1, constructed on the data of scenario 1 reveals that variance as well as test MSE of ridge regression is lesser than that of lasso regression. However, figure 2 constructed on the data of scenario 2, lasso outperforms ridge on all 3 metrics.

This indicates that none of the method is superior to another. Lasso which inherently does variable selection improving interpretability performs well when few predictors are enough to decide the response, whereas ridge performs well when response is a function of many variables with coefficients of roughly equal size.

Therefore, for an un-regularized dataset, it's better to check both the methods or a trade-off between them. This approach is well known as elastic-net regularization.

Elastic-Net Regularization

So far we have seen dealt with the values of alpha either 0 (L2) or 1 (L1). However, an efficient model can be formed by combining the properties of both ridge and lasso regularization. Elastic-Net minimizes the effect of least squares estimates along with the penalty of both ridge and lasso or minimizes the following equation –

$$\sum(\hat{y} - y)^2 + \lambda_2 \sum \beta^2 + \lambda_1 \sum \beta$$

Elastic-Net chooses the best values of alpha and so forth lambda using cross-validation. Therefore, the only drawback over Ridge and Lasso is that it is computationally expensive and is best suitable for unknown dataset to cover whole ground.

For an illustration, we take the same spam dataset to further improve model2. We start by applying 10 fold cross-validation repeated 5 times.

```
# Reading data
train_data <- read.csv("spam_train_data.csv")

# Cross-validation
library(caret)
tc <- trainControl(method = "repeatedCV", number = 10, repeats = 5)
```

Next, we form tuning values for alpha ranging from 0 to 1 and lambda values ranging from 0 to 0.5.

```
# Tuning values for alpha and lambda
lambda.grid <- seq(0, 0.5, length = 100)
alpha.grid <- round(seq(0, 1, length = 5),2)

# Forming a dataframe for all tuning alpha values against each tuning lambda values
tune_grid <- expand.grid(.alpha = alpha.grid, .lambda = lambda.grid)
```

Let's now fit predictive model over tuning parameters with method to be used as *glmnet*.

```
elas_net_model <- train(spam~., data = train_data, method = "glmnet",
        tuneGrid = tune_grid, trControl = tc, standardize = TRUE, maxit = 1000)
```

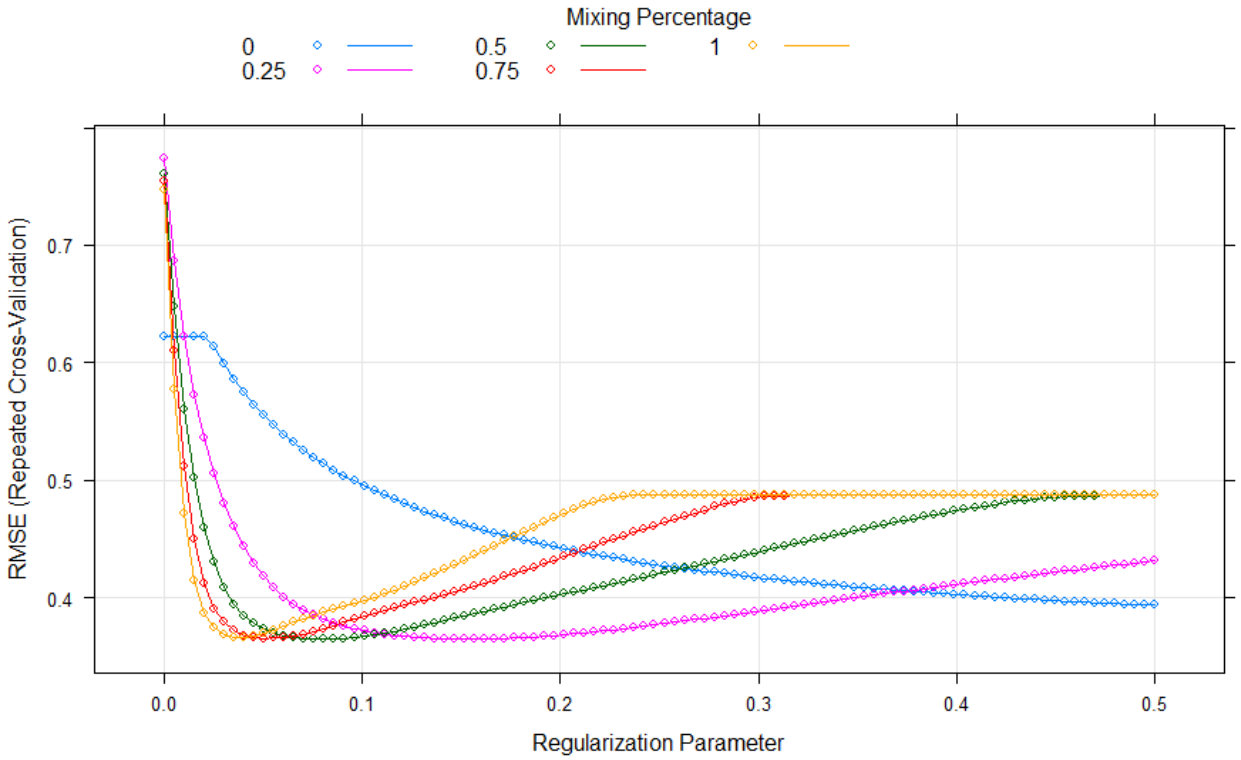
After performing CV and fitting the model, we can look for the best values of alpha and lambda as follows –

```
elas_net_model$bestTune
# alpha     lambda
# 0.5       0.08080808
```

The lambda (0.08) for alpha (0.5) against RMSE (repeated CV) can be observed through a simple plot.

```
plot(elas_net_model)
```

The green line depicts the least RMSE at lambda = 0.08. With more values of alpha, better insights can be drawn but again on the cost of increased computation.



As we know Elastic-Net combines the properties of its predecessors, therefore few predictors can be made zero too as can be found out using following command –

```
coef(elas_net_model$finalModel, s = elas_net_model$bestTune$lambda)
```

Now, let us observe the final accuracy of given model and compare it with last models.

```
en_model <- cv.glmnet(train_mat,train_data$spam,alpha= elas_net_model$bestTune$alpha)

# Confusion matrix on train data for Model2
m2_train <- ifelse(predict(en_model,type = "response",train_mat) >= 0.5,1,0)
m2_cm <- addmargins(table(m2_train,train_data$spam))
en_train2_acc <- (m2_cm[1] + m2_cm[5]) / m2_cm[9] * 100
en_train2_acc # 82.6087

# Confusion matrix on test data for Model2
m2_test <- ifelse(predict(en_model,type = "response",test_mat) >= 0.5,1,0)
m2_cm <- addmargins(table(m2_test,test_data$spam))
en_test2_acc <- (m2_cm[1] + m2_cm[5]) / m2_cm[9] * 100
en_test2_acc # 82
```

The table given below compares the resultant accuracies of L2, L1 and Elastic-Net regularization. Clearly, elastic-net performs well on given data but again owing to its drawback of computation.

Accuracy	**Un-Regularized Model 2**	**Ridge**	**Lasso**	**Elastic-Net**
Train Data	100	82.60	80.86	82.61
Test Data	66	78	80	82

Section II – Bagging

Consider the similar dataset of predicting house price used earlier. Let's try to predict the house price using decision trees (again a case of regression problem, to be performed using decision tree regressor).

To begin with, we first split our training dataset of 50 observations into 2 sets (25 each). For this problem, we ignore first predictor named "ID".

```
# Reading data
train_data <- read.csv("hp_train_old.csv")
test_data <- read.csv("hp_test.csv")[-1]

# Necessary library
library(tree)

# Splitting training data into 2 parts
set.seed(42)
smp_size <- floor(0.5 * nrow(train_data))
train_ind <- sample(seq_len(nrow(train_data)), size = smp_size)
train1 <- train_data[train_ind, ][-1]
train2 <- train_data[-train_ind, ][-1]
```

Next, we deploy regression tree on to each of the training set and further calculate the corresponding train and test data RMSE.

```
rmse_t1_train <- sqrt(mean((predict(t1, train1[-2]) - train1["Price"])^2));
rmse_t1_train # 146846.4
rmse_t1_test <- sqrt(mean((predict(t1, test_data[-2]) - test_data["Price"])^2));
rmse_t1_test # 219506.9

rmse_t2_train <- sqrt(mean((predict(t2, train2[-2]) - train2["Price"])^2));
rmse_t2_train # 217557.1
rmse_t2_test <- sqrt(mean((predict(t2, test_data[-2]) - test_data["Price"])^2));
rmse_t2_test # 591291.5
```

Let us compare and observe the RMSE –

RMSE	**Tree 1**	**Tree 2**
Train Data	146846.4	217557.1
Test Data	219506.9	591291.5

As can be inferred from the table, the RMSE of both trees are quite different yet both trees are arrived from same training data. This problem occurs due to ***high variance*** in given trees. To reduce such high variance, we use bagging. Let us discuss about bagging and one of its special approach namely random forest.

Bagging

Bootstrap aggregation is a general purpose procedure for reducing the variance of a statistical learning model. It works on a principle which states that averaging a set of observations reduces variance. Since it is not possible to attain a new set of observations for every run, therefore averaging relies upon the

fundamental concept of bootstrap. Let us understand what and how does bootstrap works before we start deploying it into bagging.

Bootstrap helps in obtaining new distinct sample sets by repeatedly sampling observations from the original data set. The sampling is performed with replacement i.e. similar observation values can occur in the same sample set.

The idea behind forming lots of sample repeated dataset is to arrive at a final prediction of standard error which is close to the standard error if we use individual dataset. To understand this aspect, let us take a dummy data shown below –

X	Y
2.3	4.5
5.6	9.6
3.6	8.0

Bootstrap works to minimize the term –

$$Var(\propto X + (1-\propto)Y)$$

where, the parameter $\propto$ is given by –

$$\propto = \frac{\sigma_Y^2 - \sigma_{XY}}{\sigma_X^2 + \sigma_Y^2 - 2\sigma_{XY}}$$

Let us create 3 different bootstrap datasets to be marked as B. For each of the B datasets, we'll get corresponding $\propto$ values. Table given below depicts the B bootstrap datasets along with their $\propto$.

	Original		$B=1$		$B=2$		$B=3$	
	X	Y	X	Y	X	Y	X	Y
Observation 1	2.3	4.5	2.3	4.5	2.3	4.5	5.6	9.6
Observation 2	5.6	9.6	3.6	8.0	5.6	9.6	3.6	8.0
Observation 3	3.6	8.0	3.6	8.0	5.6	9.6	3.6	8.0
$\propto$	**1.97**		**1.59**		**2.83**		**-4.0**	

We can now find standard error estimate (here 3.63) between bootstrapped $\propto$ values and original $\propto$ value using the equation given below –

$$SE_B(\alpha_{orig}) = \sqrt{\frac{1}{B-1}\sum_{i=1}^{B}\left(\alpha_i - \frac{1}{B}\sum_{j=1}^{B}\alpha_i\right)^2}$$

This signifies that we can depend upon bootstrap sampling technique for our real-world data.

To perform similar aspect in R, we use **boot** library to automate the process and pass number of bootstrap sample dataset to be made (here 3).

```
# Necessary library
library(boot)

# Dataset creation
vec <- c(2.3, 4.5, 5.6, 9.6, 3.6, 8)
df <- as.data.frame(matrix(vec, ncol = 2, byrow = T))
colnames(df) <- c('X', 'Y')
df

#   X   Y
# 2.3 4.5
```

```
# 5.6 9.6
# 3.6 8.0

# Function to calculate alpha for each B
alpha.fn <- function(data, index){
  X <- data$X[index]
  Y <- data$Y[index]
  return((var(Y) - cov(X, Y))/(var(X) + var(Y) - 2*cov(X, Y)))
}

# Deploying bootstrap
bootstrap <- boot(mat, alpha.fn, R = 3)

# Alpha values for each B
bootstrap$t

#             [,1]
# [1,] 1.590909
# [2,] 2.833333
# [3,] -4.00000

# Original alpha and standard error
bootstrap

# Bootstrap Statistics :
#       original      bias    std. error
# t1* 1.970874 -1.82946    3.639971
```

Note – The given illustration is delivering results based on current bootstrap dataset. With each run, dataset might change and so does the SE.

Bagging (after BS explanation)

As we have observed through bootstrap that we can create distinct data samples from a same dataset with a variance approximately equal to when compared with model if built with individual dataset. The fundamental concept of bagging is to take many training sets from the population, build a separate prediction model using each training set, and average the resulting predictions.

In terms of tree, governing on the principle of bootstrap, we make B separate training sets with functions $f_1(x)$, $f_2(x)$ …..$f_B(x)$ and average them in order to obtain a single model with least variance given by –

$$f_{bag}(x) = \frac{1}{B}\sum_{i=1}^{B} f_i(x)$$

Bagging can be applied to both regression and classification models. In this course, we'll apply bagging to decision trees (regressor/classifier). A bagged decision tree is not pruned and forms quite deep. Therefore, each individual tree has high variance and low bias. However, averaging in the end reduces variance. In case of classification trees, we can mark the category predicted by each B trees and finally the majority vote wins.

Let us dive into the concept by taking up the scenario of regression trees applied upon predicting the house price.

We use the **randomForest** library to implement bagging (and further random forest) on the data. Continuing with the variables defined while building tree on house price dataset we pass all the predictors to be used while forming bagged tree as shown –

```
# Necessary library
library(randomForest)

# Bagged Tree 1
# here mtry = 4 means we're considering all 4 predictors
bag_t1 <- randomForest(Price~.,train1, mtry = 4, importance = TRUE)

# Bagged Tree 2
bag_t2 <- randomForest(Price~.,train2, mtry = 4, importance = TRUE)
```

Here, for each training data a total of 500 trees are formed by default and for each tree we have their corresponding MSE. The RMSE of the bagged trees model can be compared with simple tree as given below –

```
# Storing test data in temporary variable for future use
temp <- test_data

# RMSE of bagged tree 1

# Accepting only those locations from test data which are also avail. in training data
test_data$Location <- factor(test_data$Location, levels = levels(train1$Location))

rmse_t1_bag_train <- sqrt(mean(((bag_t1$predicted - train1["Price"])^2)))
rmse_t1_bag_train # 209371.5

rmse_t1_bag_test <- sqrt(mean(((predict(bag_t1, test_data[-2]) - test_data["Price"])^2),na.rm =
T))
rmse_t1_bag_test # 173166.7


# RMSE of bagged tree 2

test_data <- temp

test_data$Location <- factor(test_data$Location, levels = levels(train2$Location))

rmse_t2_bag_train <- sqrt(mean(((bag_t2$predicted - train2["Price"])^2)))
rmse_t2_bag_train # 311188.8

rmse_t2_bag_test <- sqrt(mean(((predict(bag_t2, test_data[-2]) - test_data["Price"])^2),na.rm =
T))
rmse_t2_bag_test # 199828.8
```

	RMSE	**Simple tree**	**Bagged tree**
Tree 1	**Train Data**	146846.4	209371.5
	Test Data	219506.9	173166.7
Tree 2	**Train Data**	217557.1	311188.8
	Test Data	591291.5	199828.8

As can be inferred from the table that RMSE values of bagged tree for training and testing data are quite closer for both trees as compared with simple tree. Further, we can also observe that range of RMSE values of bagged tree is quite less than as compared to simple tree.

Tree 1 + Tree 2	Minimum, Maximum	Range
Simple Tree	146846.4, 591291.5	444445.1
Bagged Tree	173166.7, 311188.8	138022.1

Although building bagged trees losses the interpretability of the model but significant predictor can be retrieved using *importance* variable in R. For instance, important variable for bagged tree 1 can be listed in sequence of their mean decrease in accuracy.

```
bag_t1$importance
#                %IncMSE IncNodePurity
# Location    1490618893  1.025350e+12
# Bedrooms   -407450061  8.014018e+09
# Bathrooms   499315256  1.213503e+10
# Size        9652204990  1.290238e+11
```

---

random forests

So far, we have observed the significance of bagging over simpler decision trees. However, this procedure too faces an issue of partiality. Recall, that we've passed all the predictors to the bagged model, so in case if there happens to be a dominating predictor then for each tree made its effect will overcome the effect of other relevant predictors. Therefore, there's a need to remove this static predictor which is done usually by passing only square root number of predictors to the model. This will ensure that for each new run different predictors are chosen and hence suppressing the dominating effect of one or two predictors. This phenomenon is also coined as random forests.

Following the similar coding style of bagging except updating the value of argument *mtry* to 2 (square root of 4).

```
# RF Tree 1
rf_t1 <- randomForest(Price~.,train1, mtry = 2, importance = TRUE)

# RF Tree 2
rf_t2 <- randomForest(Price~.,train2, mtry = 2, importance = TRUE)
```

Proceeding with the calculations of RF RMSE –

```
# RMSE RF Tree 1

test_data <- temp
test_data$Location <- factor(test_data$Location, levels = levels(train1$Location))

rmse_t1_rf_train <- sqrt(mean(((rf_t1$predicted - train1["Price"])^2)))
rmse_t1_rf_train # 209653.3

rmse_t1_rf_test <- sqrt(mean(((predict(rf_t1, test_data[-2]) - test_data["Price"])^2),na.rm = T))
rmse_t1_rf_test # 119622.6

# RMSE RF Tree 2

test_data <- temp
test_data$Location <- factor(test_data$Location, levels = levels(train2$Location))
```

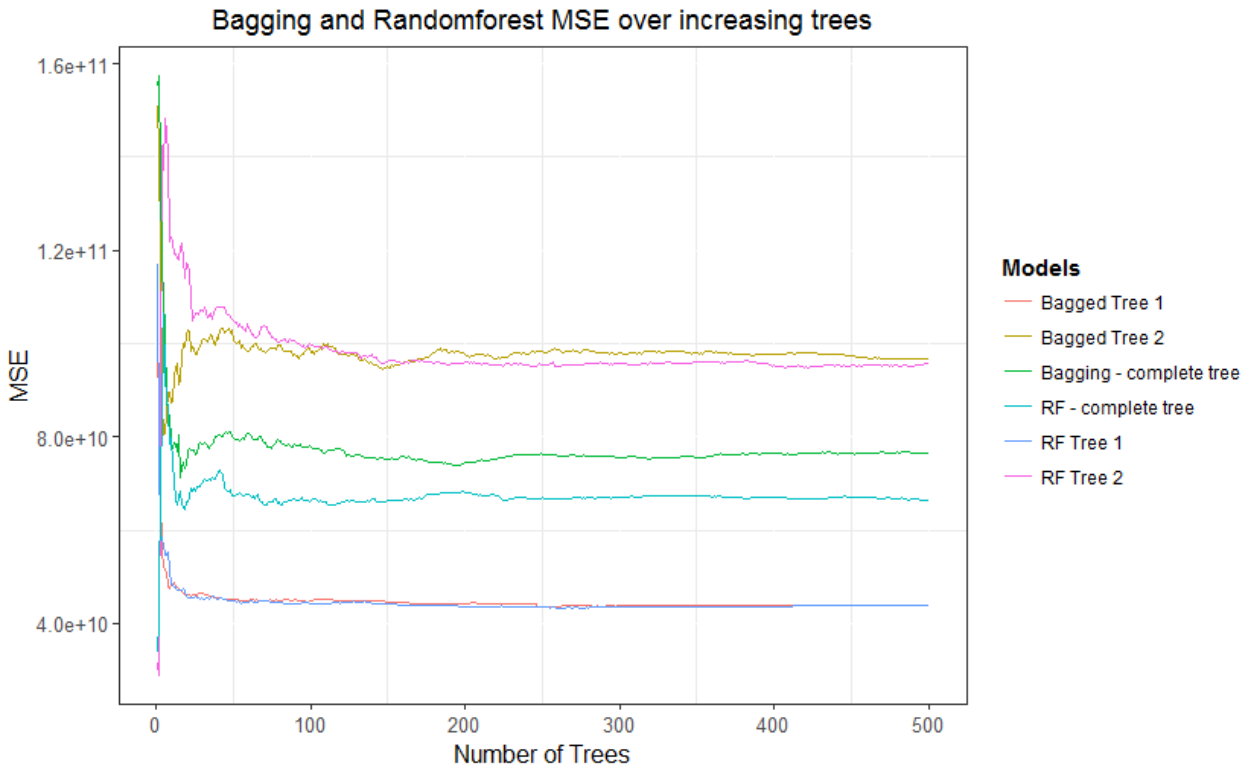
```
rmse_t2_rf_train <- sqrt(mean(((rf_t2$predicted - train2["Price"])^2)))
rmse_t2_rf_train # 309596.7

rmse_t2_rf_test <- sqrt(mean(((predict(rf_t2, test_data[-2]) - test_data["Price"])^2),na.rm = T))
rmse_t2_rf_test # 158575.1
```

Let us compare final RMSE values to previous models (decision tree and bagging) –

	RMSE	Simple tree	Bagged tree	Random Forest
Tree 1	**Train Data**	146846.4	209371.5	209653.3
	Test Data	219506.9	173166.7	119622.6
Tree 2	**Train Data**	217557.1	311188.8	309596.7
	Test Data	591291.5	199828.8	158575.1

The RMSE values of random forest are less suitable for given model when compared with bagged trees but on the other hand, as the number of trees increases the random forest shows better results in MSE as compared with bagged trees. This can be verified by plotting a graph between each of these models against their corresponding trees' MSE. We have also applied bagging and random forest over complete training data and it is quite clear that right from starting and with increasing trees random forest shows significant improvement over bagging.

```
library(ggplot2)

bag <- randomForest(Price~.,train_data, mtry = 4, importance = TRUE)
rf <- randomForest(Price~.,train_data, mtry = 2, importance = TRUE)

no_trees <- seq(1,500,length.out = 500)
ggplot() +
  geom_line(aes(no_trees, bag_t1$mse, col = "Bagged Tree 1")) +
  geom_line(aes(no_trees, bag_t2$mse, col = "Bagged Tree 2")) +
  geom_line(aes(no_trees, rf_t1$mse, col = "RF Tree 1")) +
  geom_line(aes(no_trees, rf_t2$mse, col = "RF Tree 2")) +
  geom_line(aes(no_trees, bag$mse, col = "Bagging - complete tree")) +
  geom_line(aes(no_trees, rf$mse, col = "RF - complete tree")) +
  labs(col = 'Models') +
  xlab("Number of Trees") + ylab("MSE") +
  ggtitle("Bagging and Randomforest MSE over increasing trees") +
  theme_bw() +
  theme(panel.grid.major = element_line(colour = "white"),
        axis.text = element_text(size = 10),
        axis.title = element_text(size = 12),
        plot.title = element_text(hjust = 0.5),
        legend.title = element_text(face = 'bold'))
```



---

bagging and RF on Classification data

We have observed the benefit of bagging and RF on regression models, now let us take up our spam dataset and apply these concepts to observe improvement in the accuracy and thence prediction from the model.

As we discussed earlier, for classification models, bagging accepts the predicted class of each tree and finally go for majority vote. For spam dataset, classes can be either spam or not spam.

The coding format for applying bagging and random forest remains same as with previous section except we should pass input data as factor class to make clear that we are going to perform classification.

```
# Necessary library
library(tree)
library(randomForest)

# Reading data
train_data <- read.csv("spam_train_data.csv")
test_data <- read.csv("spam_test_data.csv")

# Splitting Training data into 2 parts
smp_size <- floor(0.5 * nrow(train_data))
set.seed(42)
train_ind <- sample(seq_len(nrow(train_data)), size = smp_size)
train1 <- train_data[train_ind, ]
train2 <- train_data[-train_ind, ]

# Deploying tree on each training data
```

```
t1 <- tree(spam~., train1)
t2 <- tree(spam~., train2)

# Bagged Tree 1
# here mtry = 30 means we're considering all 30/57 predictors
# We convert "spam" as "factor" to avoid Regression and perform Classification
bag_t1 <- randomForest(as.factor(spam)~.,train1, mtry = 30, importance = TRUE)

# Bagged Tree 2
bag_t2 <- randomForest(as.factor(spam)~.,train2, mtry = 30, importance = TRUE)

# RF Tree 1
# here mtry = 8 means we're considering approx. sqrt. of 57 predictors
rf_t1 <- randomForest(as.factor(spam)~.,train1, mtry = 8, importance = TRUE)

# RF Tree 2
rf_t2 <- randomForest(as.factor(spam)~.,train2, mtry = 8, importance = TRUE)
```

Once models are built, we can find the corresponding accuracies particular to each model.

```
# Accuracy of Tree 1

bin <- ifelse(predict(t1,train1) >= 0.5, 1,0)
cm <- addmargins(table(bin, train1$spam))
acc_t1_train <- (cm[1] + cm[5]) / cm[9] * 100
acc_t1_train # 87.7193

bin <- ifelse(predict(t1,test_data) >= 0.5, 1,0)
cm <- addmargins(table(bin,test_data$spam))
acc_t1_test <- (cm[1] + cm[5]) / cm[9] * 100
acc_t1_test # 78


# Accuracy of Tree 2

bin <- ifelse(predict(t2,train2) >= 0.5, 1,0)
cm <- addmargins(table(bin, train2$spam))
acc_t2_train <- (cm[1] + cm[5]) / cm[9] * 100
acc_t2_train # 91.37931

bin <- ifelse(predict(t2,test_data) >= 0.5, 1,0)
cm <- addmargins(table(bin,test_data$spam))
acc_t2_test <- (cm[1] + cm[5]) / cm[9] * 100
acc_t2_test # 76

# Accuracy function for Bagging and RF models
acc_func <- function(user_model,tr_data){
  bin <- predict(user_model,tr_data)
  cm <- addmargins(table(bin, tr_data$spam))
  return((cm[1] + cm[5]) / cm[9] * 100)
}

# Accuracy of Bagged Tree 1
acc_func(bag_t1, train1) # 100
acc_func(bag_t1, test_data) # 86

# Accuracy of Bagged Tree 2
acc_func(bag_t2, train2) # 100
```

```
acc_func(bag_t2, test_data) # 88


# Accuracy of RF Tree 1
acc_func(rf_t1, train1) # 100
acc_func(rf_t1, test_data) # 88

# RF Tree 2
acc_func(rf_t2, train2) # 100
acc_func(rf_t2, test_data) # 92
```

Next, let's compare the accuracies for discussed models –

	RMSE	Simple tree	Bagged tree	Random Forest
Tree 1	**Train Data**	87.71	100	100
	Test Data	78	86	88
Tree 2	**Train Data**	91.37	100	100
	Test Data	76	88	92

As can be inferred from the table, accuracy as well as the range of accuracies (synchronization) improves with bagging followed by random forests.

Section III - Boosting

Boosting is an important portion of ML which helps to improve the prediction of a model. It can be applied both on classification as well as on regression problems. In this course, we'll discuss about three main boosting algorithms namely –

- Adaptive Boosting
- Gradient Boosting
- Extra Gradient Boosting

The fundamental principle behind all boosting algorithm is that they improve model accuracy by boosting weak learners to become strong learners. The way they convert a weak learner into a strong learner has given rise to different algorithms.

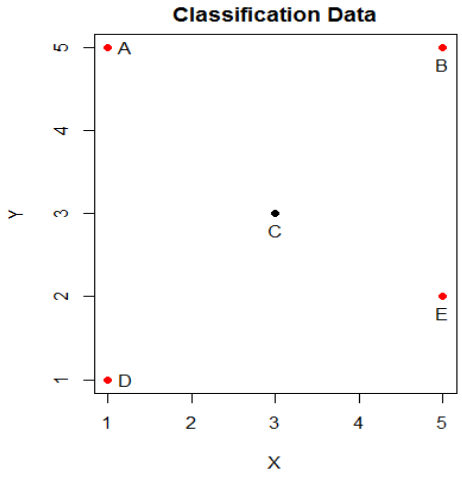
Consider a dataset of 5 points as shown below (stored in *boost_data.csv*) –

X	Y	T
3	3	0
1	1	1
5	2	1
1	5	1
5	5	1

where, X and Y are independent variables are T is a dependent variable. The dataset can be visualized and labelled in R as follows –

```
# Reading data
df <- read.csv("boost_data.csv")

# Plotting and labelling data
plot(df$X, df$Y,main = 'Classification Data', col = df$T, xlab = 'X', ylab = 'Y', pch = 19)
```

```
text(1.2, 1,'D')
text(5, 1.8,'E')
text(5, 4.8,'B')
text(1.2, 5,'A')
text(3, 2.8,'C')
```



Our aim is to build a decision tree on given dataset using AdaBoost. Adaptive Boosting is adaptive in the sense that it keeps on updating its weight corresponding to each data points assigning higher weights to weak learners and lower weights to strong learners. The algorithm for AdaBoost is given below –

1. Initialize equal weights to all training points
$$W = \frac{1}{N}$$
2. Calculate error rate for each classifier
$$\varepsilon = \sum W_i$$
3. Pick best classifier with smallest error rate
4. Calculate voting power for classifier
$$\propto = \frac{1}{2} ln\left(\frac{1-\varepsilon}{\varepsilon}\right)$$
5. Reached end? If no, update weights to emphasize points that were misclassified as shown below and go back to step 2 for recalculating error rate.
$$W_{new} = \begin{cases} \frac{1}{2}\frac{1}{1-\varepsilon} W_{old} & if, right \\ \frac{1}{2}\frac{1}{\varepsilon} W_{old} & if, wrong \end{cases}$$

Let us deploy the algorithm on our dataset and verify our solutions in R. We utilize ***boosting*** function available under R's ***adabag*** library to deploy adaboost with maximum trees fixed at 2.

Step 1 –

Initialize equal weights to all training points. Since we've 5 data points, therefore each data point get assigned with an equal weight of 0.2.

Weights	**Iteration 1**
W_A	0.2
W_B	0.2
W_C	0.2
W_D	0.2
W_E	0.2

Step 2 –

Defining the classifier boundaries and misclassified points corresponding to them:

Iteration	**Classifier boundary**	**Misclassified points**	**Error rate (ε)**	**Comment**
1	X>=2	A C D	0.6	Considering all the points greater than or equal to 2 in the category of 1 and 0 for the rest. **A and D:** are misclassified as 0 **C**: is misclassified as 1
	X<2	B E	0.4	Considering all the points less than 2 in the category of 1 and 0 for the rest. **B and E:** are misclassified as 0

Step 3 –

Next up, let us pick the classifier with least error rate, here X<2 with $\boldsymbol{\varepsilon} = \frac{2}{5}$ or 0.4

Step 4 –

To calculate the voting power as shown below:

$$\propto = \frac{1}{2} ln\left(\frac{1-\varepsilon}{\varepsilon}\right)$$

Since $\boldsymbol{\varepsilon} = \mathbf{0.4}$, therefore $\propto = 0.2027$

Step 5 –

Moving further with second iteration and hence reassigning new weights by formula:

$$W_{new} = \begin{cases} \frac{1}{2}\frac{1}{1-\varepsilon} W_{old} & if, right \\ \frac{1}{2}\frac{1}{\varepsilon} W_{old} & if, wrong \end{cases}$$

With X<2 only 2 training points i.e. B and E aren't classified in their right category rest 3 points i.e. A, C and D are classified correctly.

Weights	**Iteration 1**	**Iteration 2**
W_A	0.2	0.166
W_B	0.2	0.250
W_C	0.2	0.166
W_D	0.2	0.166
W_E	0.2	0.250

Recalculating error rate –

Iteration	Classifier boundary	Misclassified points	Error rate (ε)	Comment
2	X<4	B C E	0.666	Considering all the points less than 2 in the category of 1 and 0 for the rest. **B and E:** are misclassified as 0 **C**: is misclassified as 1
	X>=4	0	0	Considering all the points greater than or equal to 4 in the category of 1 and 0 for the rest. **No misclassified points**

Let us now do the same using R functions.

```
df$T <- factor(df$T)
ada_model <- boosting(T~., data=df, mfinal =
2,boos=TRUE,control=rpart.control(minsplit=0,cp=0))

ada_model$trees

# [[1]]
# n= 5
#
# node), split, n, loss, yval, (yprob)
# * denotes terminal node
#
# 1) root 5 1 1 (0.2000000 0.8000000)
# 2) X< 4 1 0 0 (1.0000000 0.0000000) *
#   3) X>=4 4 0 1 (0.0000000 1.0000000) *
#
#   [[2]]
# n= 5
#
# node), split, n, loss, yval, (yprob)
# * denotes terminal node
#
# 1) root 5 1 1 (0.2000000 0.8000000)
# 2) X>=2 2 1 0 (0.5000000 0.5000000)
# 4) X< 4 1 0 0 (1.0000000 0.0000000) *
#   5) X>=4 1 0 1 (0.0000000 1.0000000) *
#   3) X< 2 3 0 1 (0.0000000 1.0000000) *

ada_model$weights[1]
# 0.2027326

errorevol(ada_model,df)$error
# 0.4 0.0
```