Experiment 8

Aim: Understanding of R and its basics.

Steps: This includes understanding of R package as it mathS oriented giving some fast plotting functions using graphs. This also includes making model, building it, training and testing data and validating it across accuracy.

1. Download and Install R
https://cran.rstudio.com/bin/windows/base/R-3.2.3-win.exe

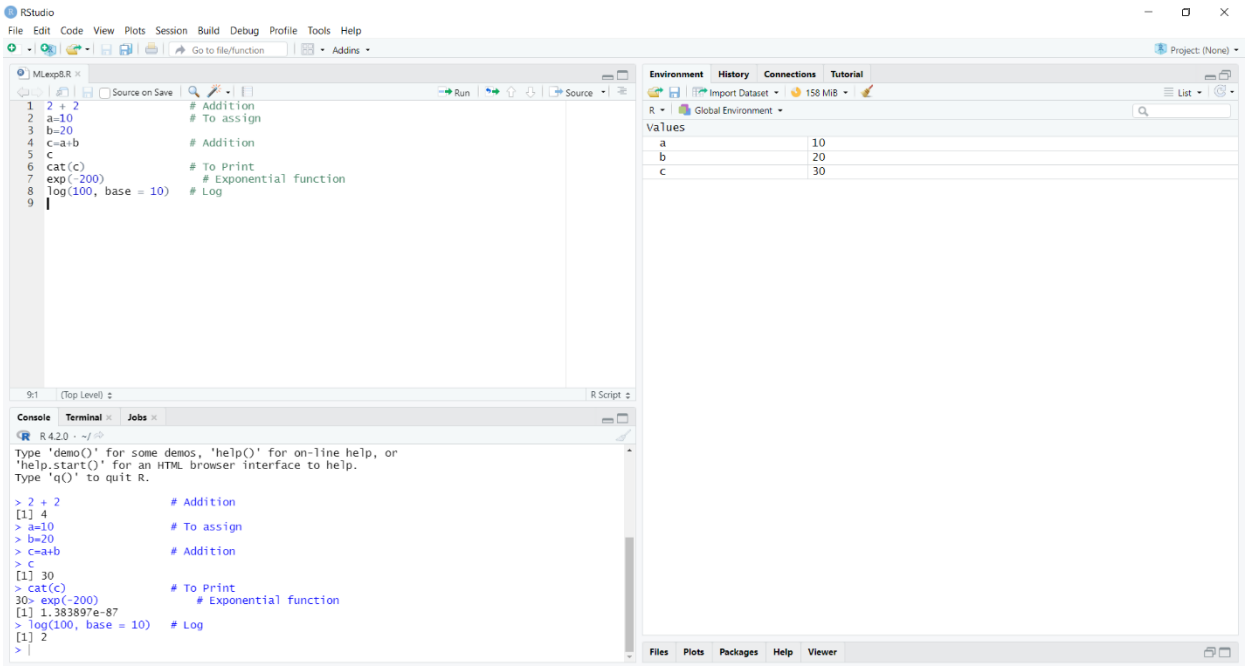
2. Install RStudio
RStudio https://download1.rstudio.org/RStudio-0.99.491.exe

3. Open the Script in RStudio

4. To execute the command just press: ctrl + Enter

Step 1: Simple Examples

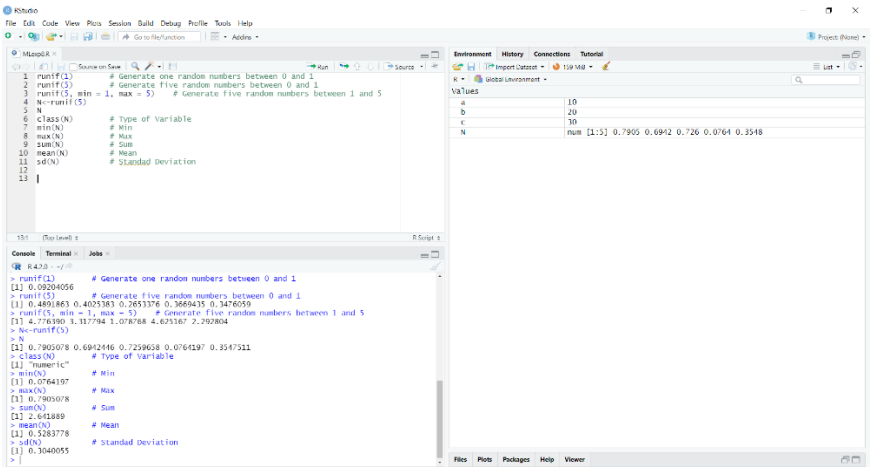
```
2 + 2                    # Addition
a=10                     # to assign
b=20
c=a+b                    # Addition
c
cat(c)                   # To Print
exp(-200)                # Exponential function
log(100, base = 10)      # Log
```



Step 2: Generate Random Number

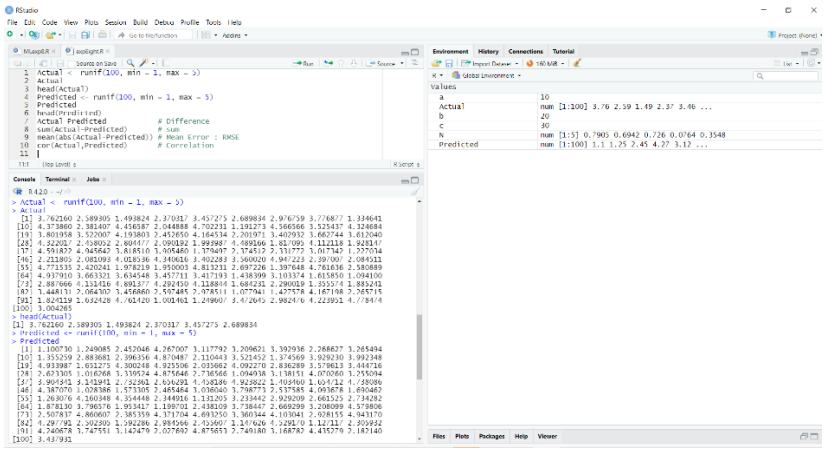
```
runif(1)                 # Generate one random numbers between 0 and 1
```

```
runif(5)                        # Generate five random numbers between 0 and 1
runif(5, min = 1, max = 5)      # Generate five random numbers between 1 and 5
N<-runif(5)
N
class(N)        # Type of Variable
min(N)          # Min
max(N)          # Max
sum(N)          # Sum
mean(N)         # Mean
sd(N)           # Standard Deviation
```



Step 2: Some More Examples

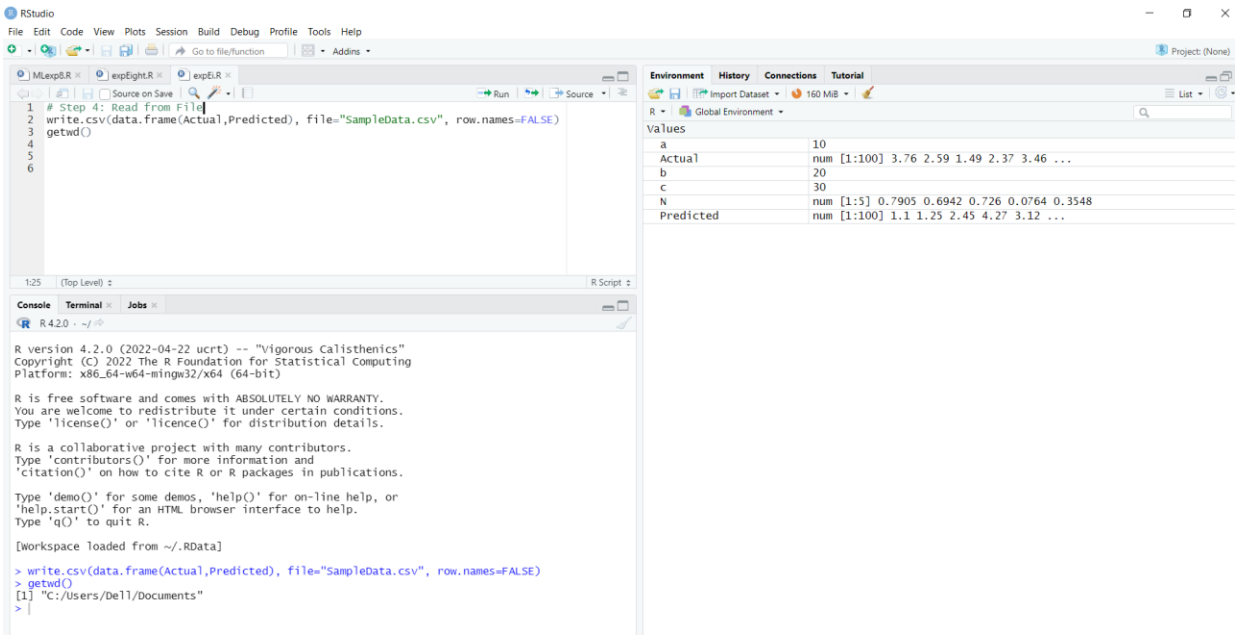
```
Actual <- runif(100, min = 1, max = 5)
Actual
head(Actual)
Predicted <- runif(100, min = 1, max = 5)
Predicted
head(Predicted)
Actual-Predicted                # Difference
sum(Actual-Predicted)           # sum
mean(abs(Actual-Predicted))     # Mean Error : RMSE
cor(Actual,Predicted)           # Correlation
```



Step 3: Writing to File

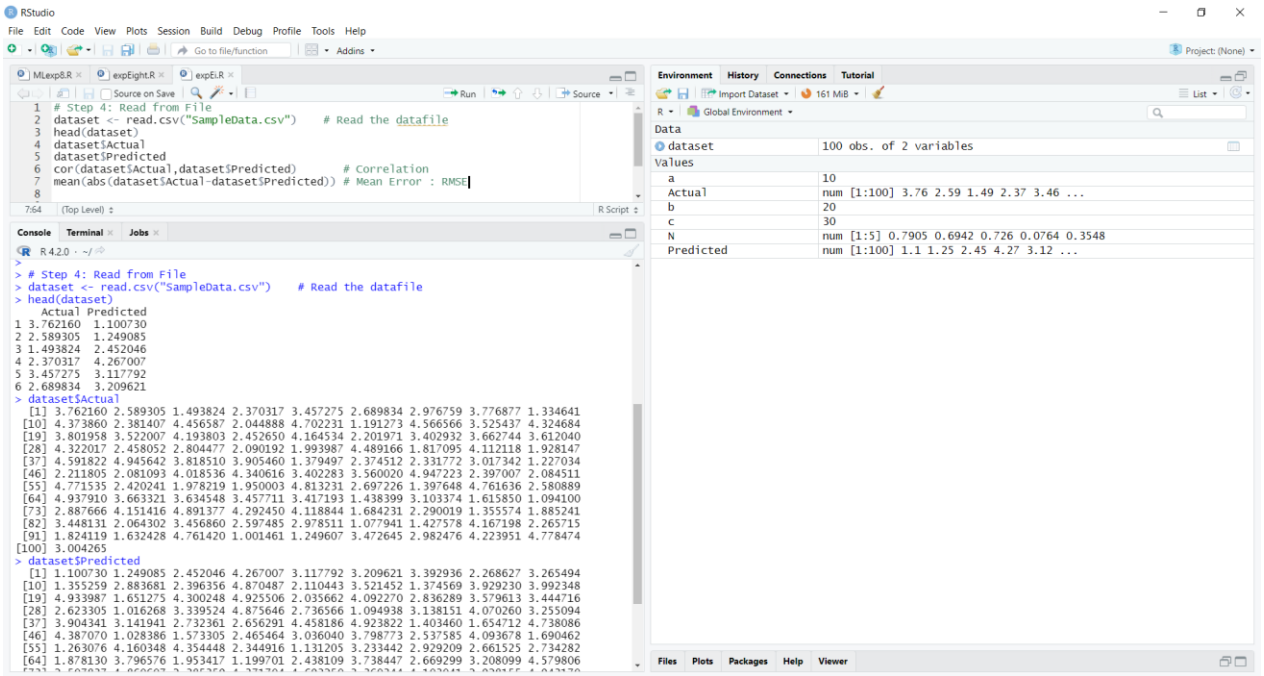
```
write.csv(data.frame(Actual,Predicted), file="SampleData.csv", row.names=FALSE)
```

getwd()



Step 4: Read from File

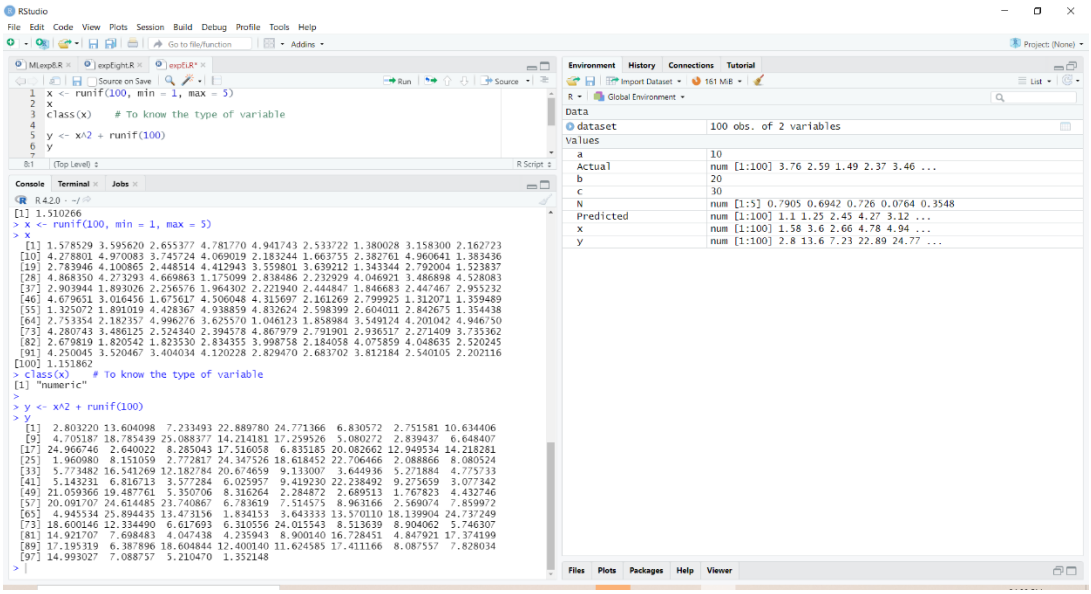
```
dataset <- read.csv("SampleData.csv")                # Read the datafile
head(dataset)
dataset$Actual
dataset$Predicted
cor(dataset$Actual,dataset$Predicted)                # Correlation
mean(abs(dataset$Actual-dataset$Predicted))          # Mean Error : RMSE
```



Step 5: Generate Random Number

```
x <- runif(100, min = 1, max = 5)
```

```
x
class(x)                                # To know the type of variable
y <- x^2 + runif(100)
y
```

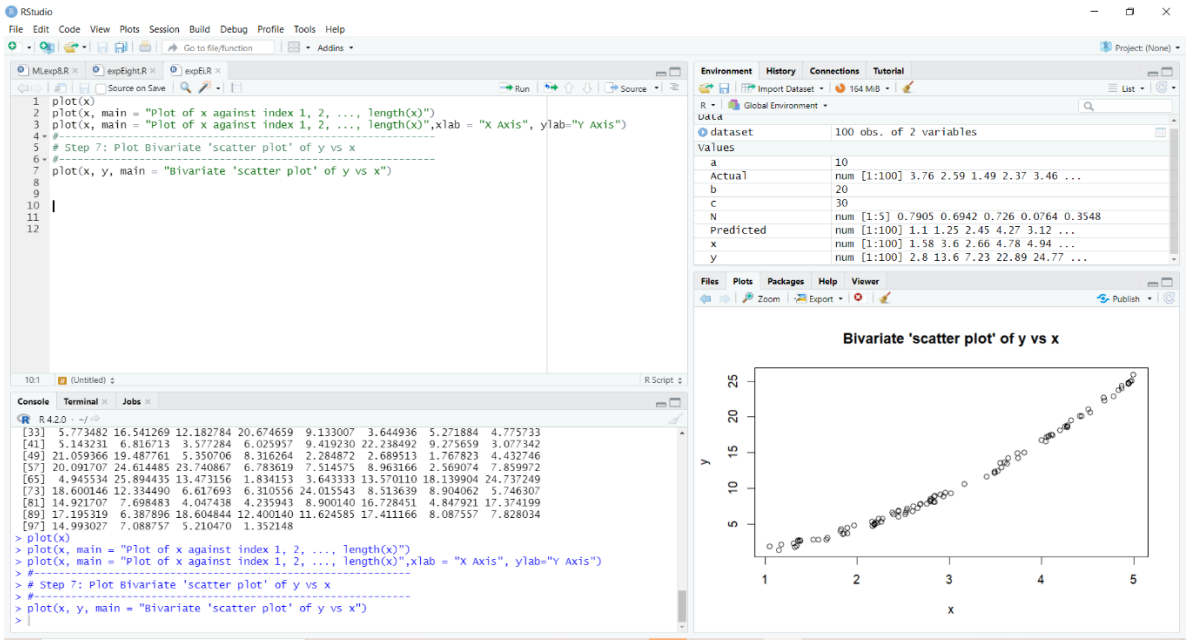


Step 6: Plot x

```
plot(x)
plot(x, main = "Plot of x against index 1, 2, ..., length(x)")
plot(x, main = "Plot of x against index 1, 2, ..., length(x)",xlab = "X Axis", ylab="Y Axis")
```

Step 7: Plot Bivariate 'scatter plot' of y vs x

```
plot(x, y, main = "Bivariate 'scatter plot' of y vs x")
```



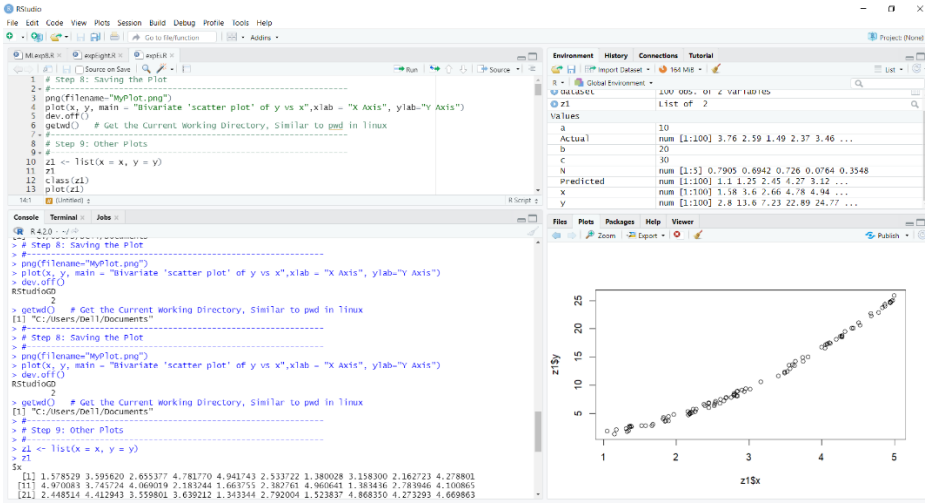
Step 8: Saving the Plot

```
png(filename="MyPlot.png")
```

```
plot(x, y, main = "Bivariate 'scatter plot' of y vs x",xlab = "X Axis", ylab="Y Axis")
dev.off()
getwd()   # Get the Current Working Directory, Similar to pwd in linux
```

Step 9: Other Plots

```
z1 <- list(x = x, y = y)
z1
class(z1)
plot(z1)
```

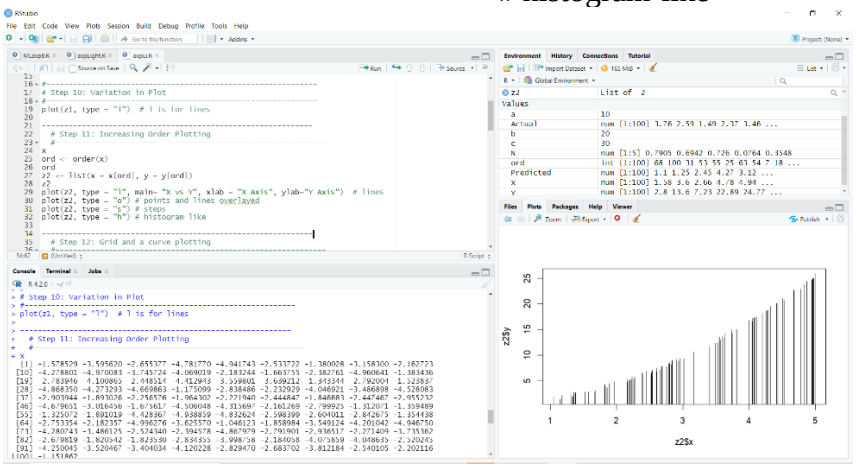


Step 10: Variation in Plot

```
plot(z1, type = "l")  # l is for lines
```

Step 11: Increasing Order Plotting

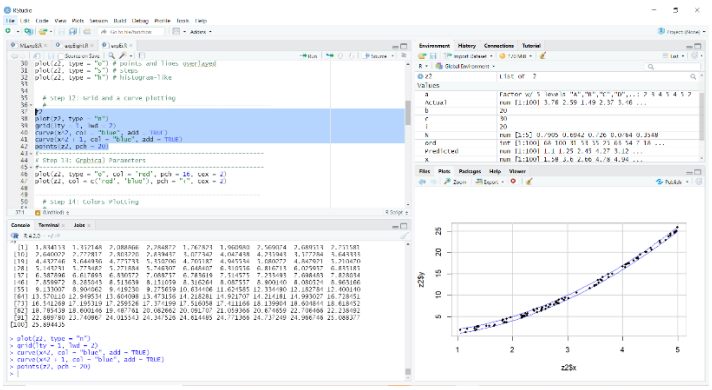
```
x
ord <- order(x)
ord
z2 <- list(x = x[ord], y = y[ord])
z2
plot(z2, type = "l", main= "X vs Y", xlab = "X Axis", ylab="Y Axis")       # lines
plot(z2, type = "o")                                                      # points and lines overlayed
plot(z2, type = "s")                                                      # steps
plot(z2, type = "h")                                                      # histogram-like
```



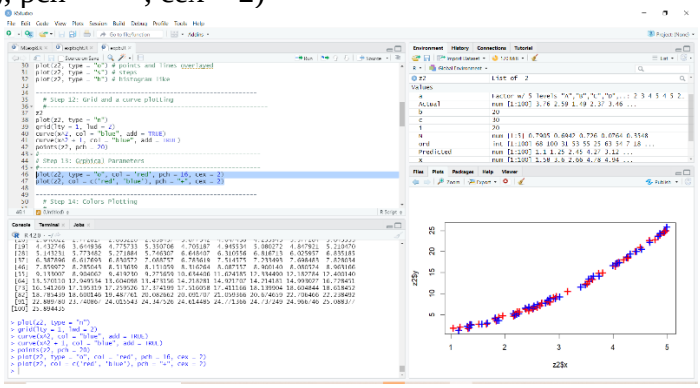
Step 12: Grid and a curve plotting

```
z2
```

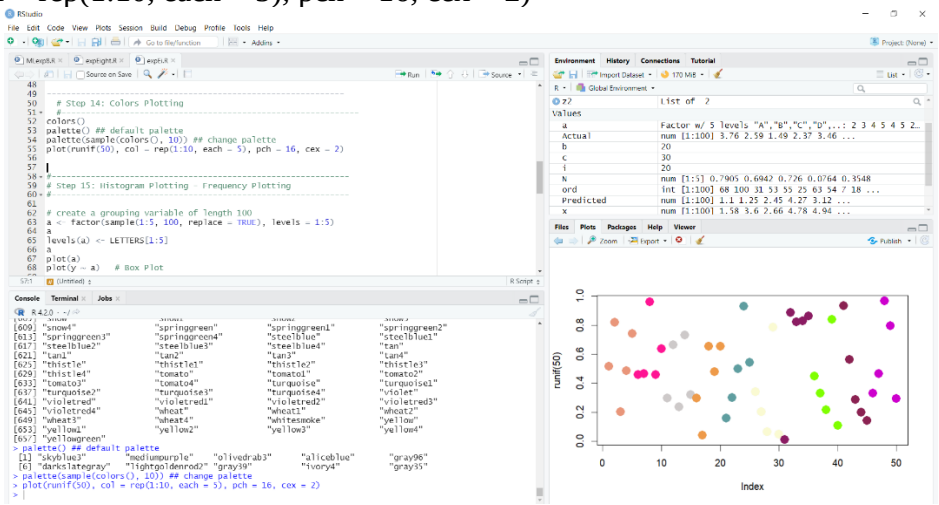
plot(z2, type = "n")
grid(lty = 1, lwd = 2)
curve(x^2, col = "blue", add = TRUE)
curve(x^2 + 1, col = "blue", add = TRUE)
points(z2, pch = 20)



Step 13: Graphical Parameters
plot(z2, type = "o", col = 'red', pch = 16, cex = 2)
plot(z2, col = c('red', 'blue'), pch = "+", cex = 2)

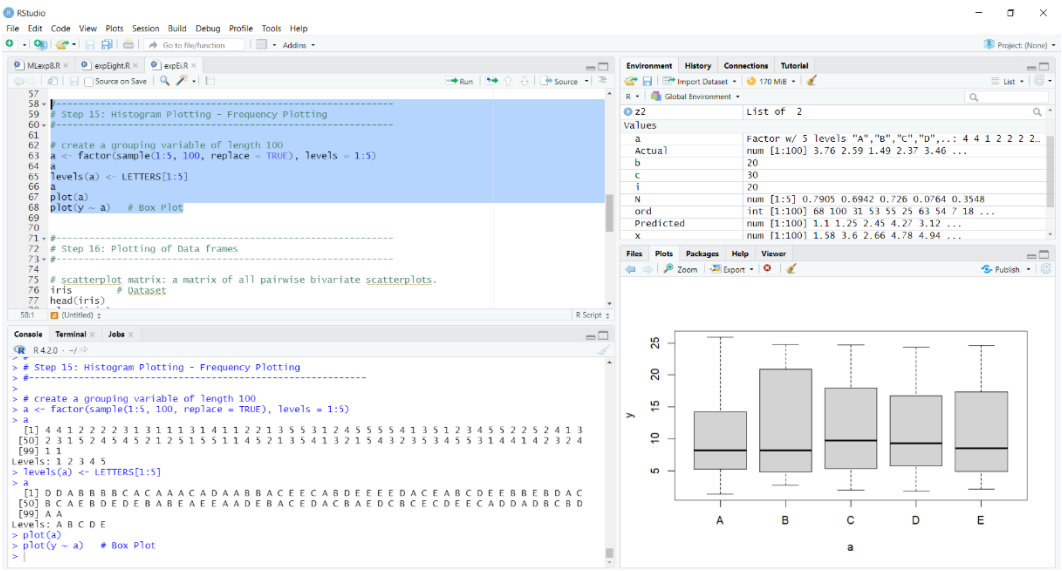


Step 14: Colors Plotting
colors()
palette() ## default palette
palette(sample(colors(), 10)) ## change palette
plot(runif(50), col = rep(1:10, each = 5), pch = 16, cex = 2)



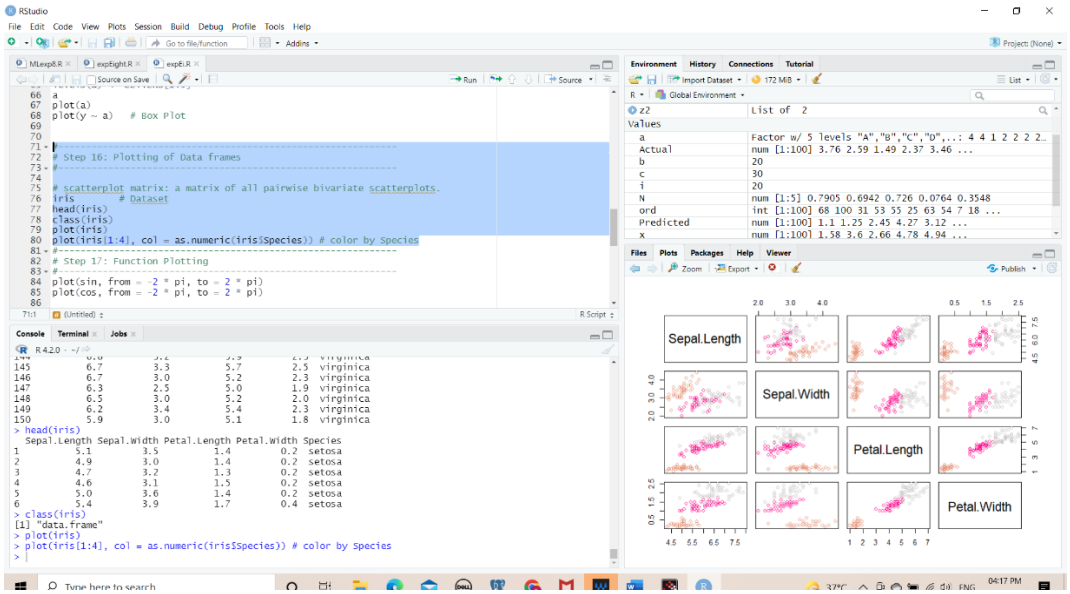
Step 15: Histogram Plotting - Frequency Plotting

```
create a grouping variable of length 100
a <- factor(sample(1:5, 100, replace = TRUE), levels = 1:5)
a
levels(a) <- LETTERS[1:5]
a
plot(a)
plot(y ~ a)                                          # Box Plot
```



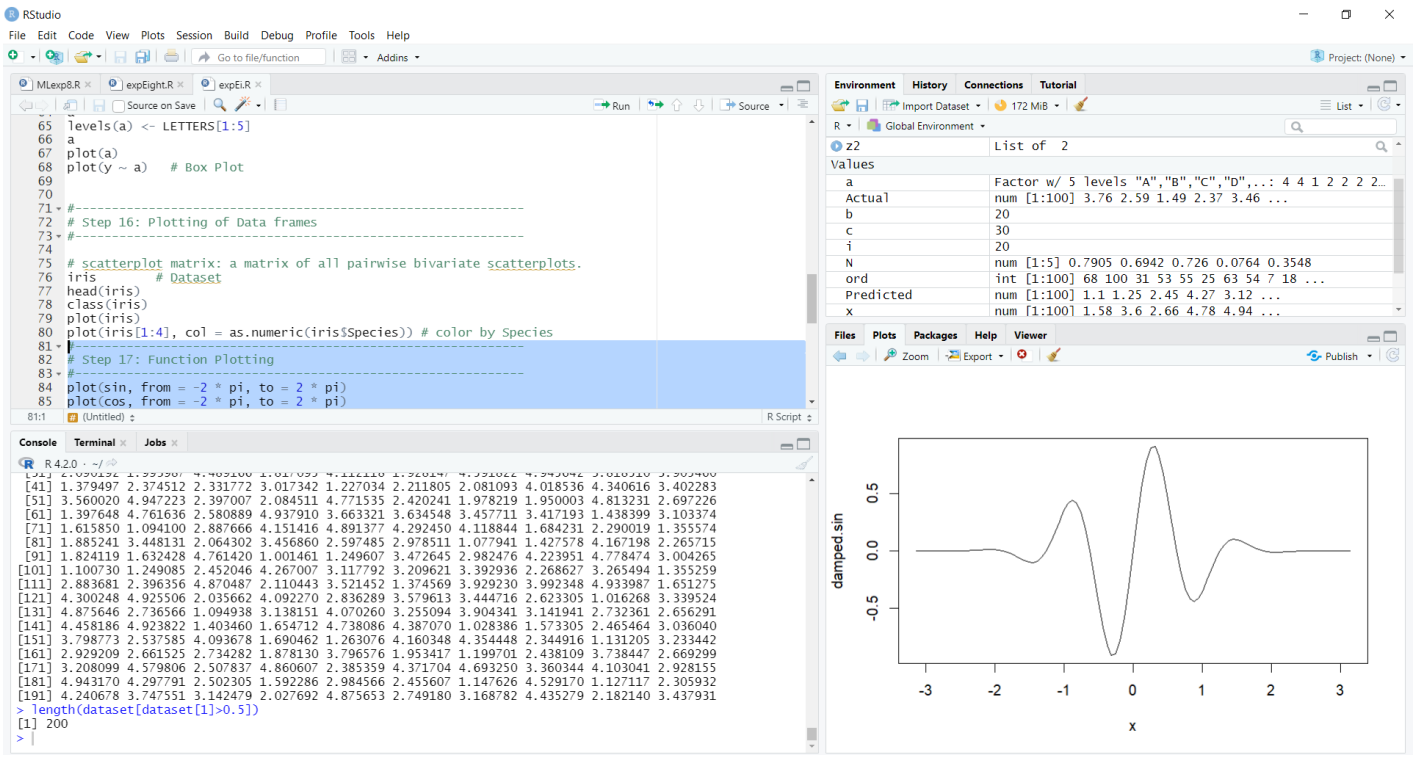
Step 16: Plotting of Data frames

```
scatterplot matrix: a matrix of all pairwise bivariate scatterplots.
iris                                                 # Dataset
head(iris)
class(iris)
plot(iris)
plot(iris[1:4], col = as.numeric(iris$Species))      # color by Species
```



Step 17: Function Plotting

```
plot(sin, from = -2 * pi, to = 2 * pi)
plot(cos, from = -2 * pi, to = 2 * pi)
damped.sin <- function(x) sin(5 * x) * exp(-x^2)      ## New function
class(damped.sin)
plot(damped.sin, from = -pi, to = pi)
for ( i in 1:20){ print (i) }
i=20
if (i==20){ print("Yes") }
else { print("NO") }
head(dataset,10)
head(dataset[1])
head(dataset[2])
head(dataset)
head(dataset[1,1])
head(dataset[1,2])
head(dataset[2:10,2])
head(dataset[1:2])
head(dataset[2:1])
head(dataset[1:2][1])
head(dataset[2:1][1])
dataset[dataset[1]>0.5]
length(dataset[dataset[1]>0.5])
```



Experiment 9

Aim: Using R studio to implement the linear model for regression.

Steps:

Introduction to Regression (Linear Model): This includes an understanding of R package; s basic commands and using commands to build.

Linear Model For Regression

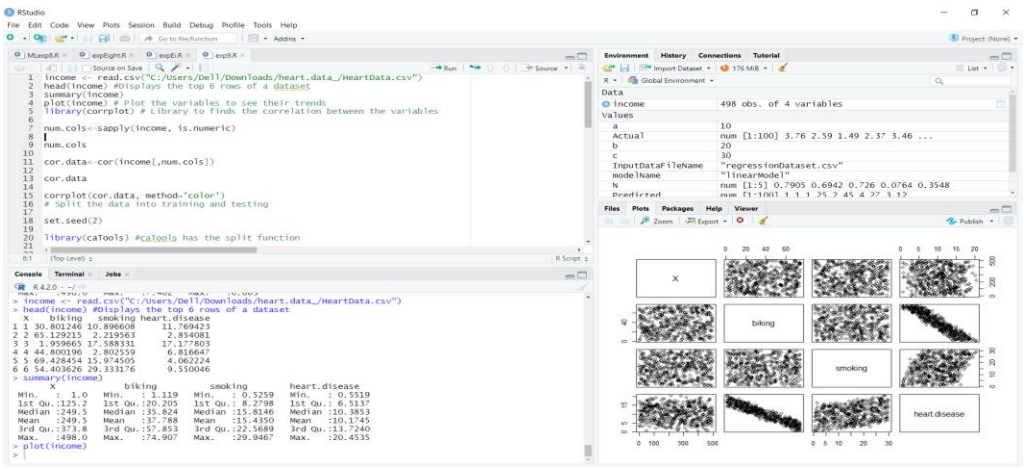
```
income <- read.csv("C:/Users/Dell/Downloads/heart.data_/HeartData.csv")
head(income) #Displays the top 6 rows of a dataset
summary(income)
plot(income) # Plot the variables to see their trends
library(corrplot) # Library to finds the correlation between the variables
num.cols<-sapply(income, is.numeric)
num.cols
cor.data<-cor(income[,num.cols])
cor.data
corrplot(cor.data, method='color')
# Split the data into training and testing
set.seed(2)
library(caTools) #caTools has the split function
split <- sample.split(income, SplitRatio = 0.7) # Assigning it to a variable split, sample.split is one of the functions we are using. With the ration value of 0.7, it states that we will have 70% of the sales data for training and 30% for testing the model
split
train <- subset(income, split = 'TRUE') #Creating a training set
test <- subset(income, split = 'FALSE') #Creating a testing set by assigning FALSE
head(train)
head(test)
View(train)
View(test)
Model <- lm(biking ~., data = train) #Creates the model. Here, lm stands for the linear regression model. Revenue is the target variable we want to track.

summary(Model)
# Prediction
pred <- predict(Model, test) #The test data was kept for this purpose
pred #This displays the predicted values
res<-residuals(Model) # Find the residuals
res<-as.data.frame(res) # Convert the residual into a dataframe
```

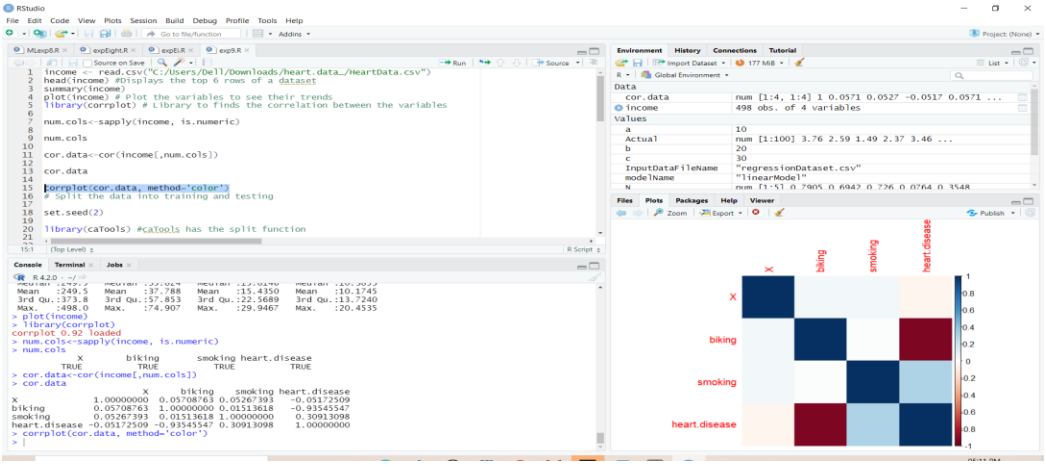
```
res # Prints the residuals
# compare the predicted vs actual values
results<-cbind(pred,test$biking)
results
colnames(results)<-c('predicted','real')
results<-as.data.frame(results)
head(results)
plot(test$biking, type = 'l', lty = 1.8, col = "red")
lines(pred, type = "l", col = "blue") #The output looks like below
plot(pred, type = "l", lty = 1.8, col = "blue") #The output looks like below, this graph shows the expected Revenue
# Calculating the accuracy
rmse <- sqrt(mean(pred-income$biking)^2) # Root Mean Square Error is the standard deviation of the residuals
rmse
```

OUTPUTS:

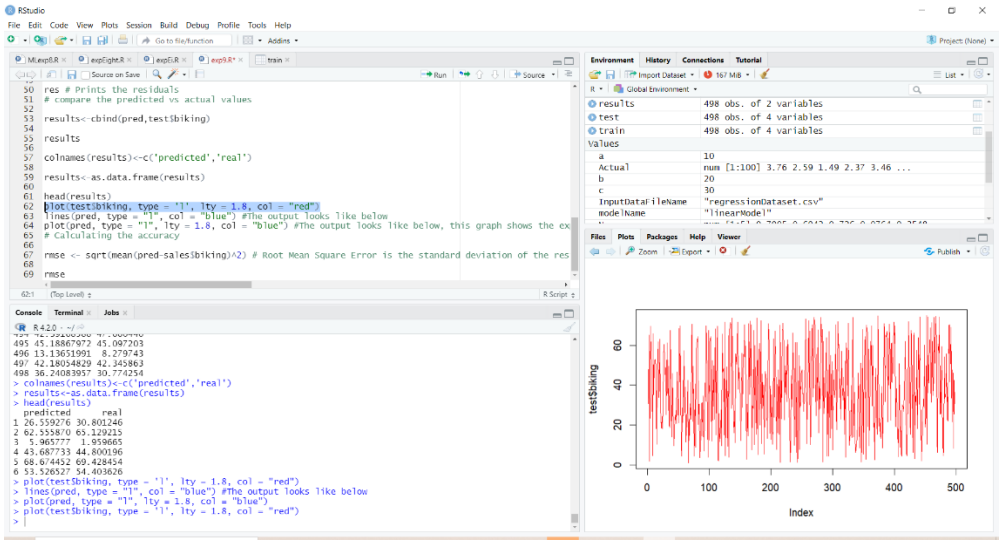
Plotting of variables

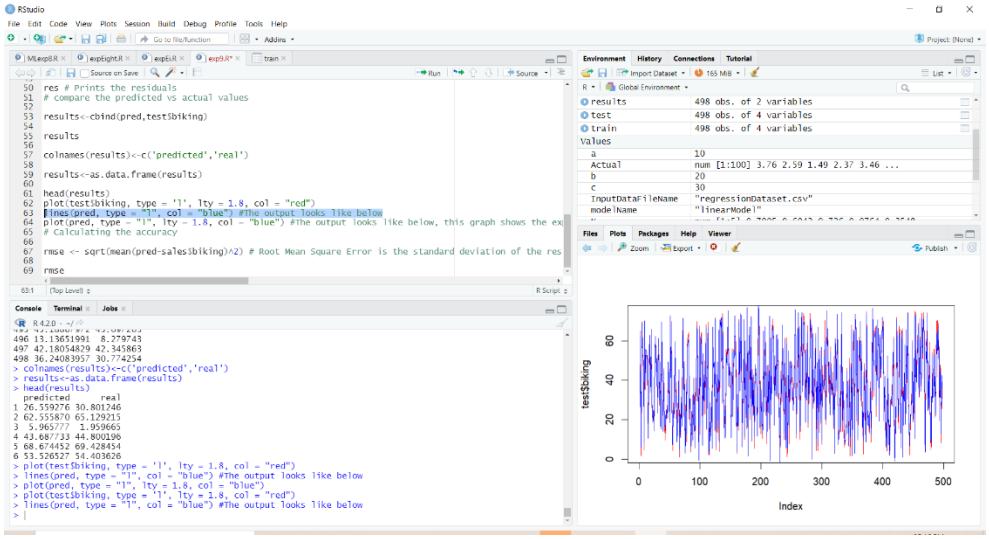


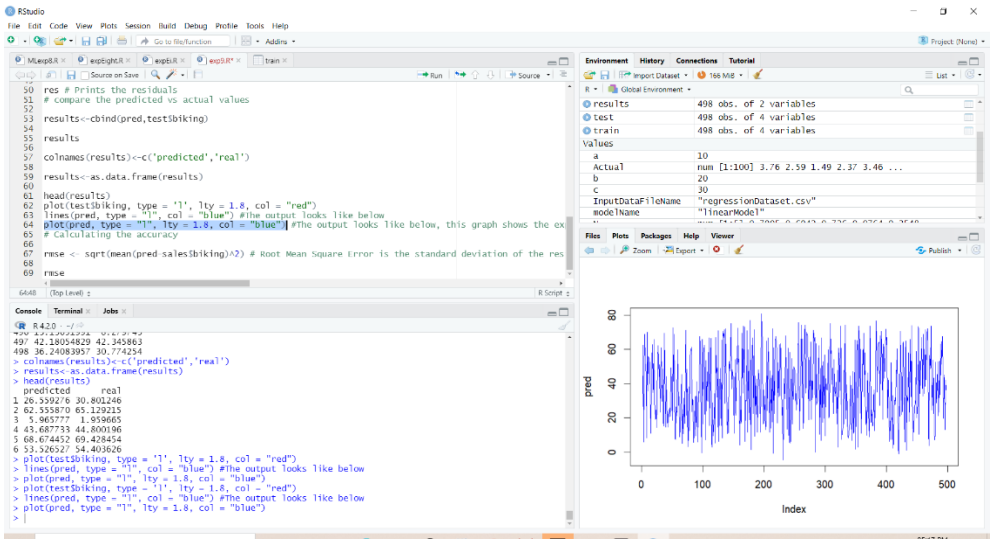
Confusion Matrix

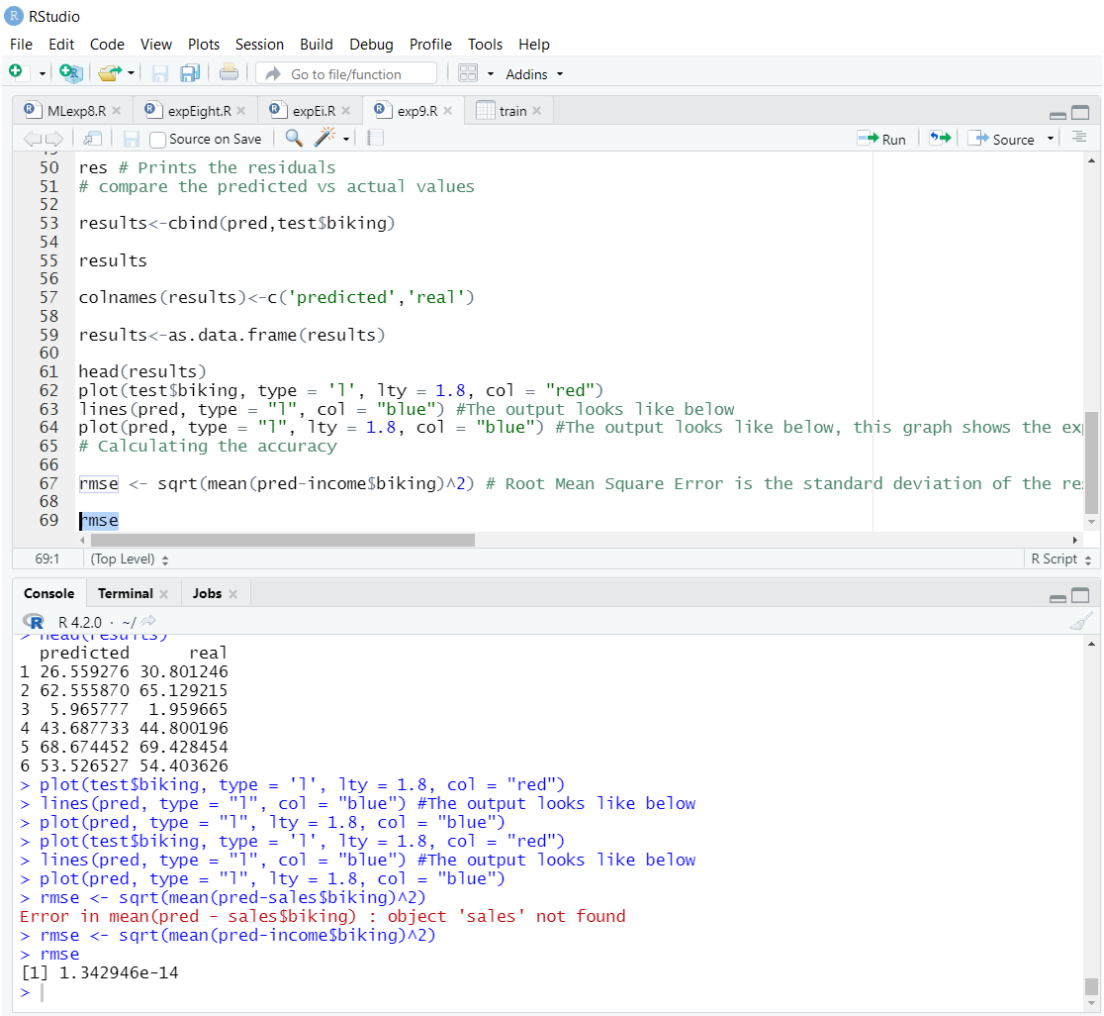


RMSE value


RStudio
res # Prints the residuals
# compare the predicted vs actual values
results<-cbind(pred,test$biking)
results
colnames(results)<-c('predicted','real')
results<-as.data.frame(results)
head(results)
plot(test$biking, type = 'l', lty = 1.8, col = "red")
lines(pred, type = "l", col = "blue") #The output looks like below
plot(pred, type = "l", lty = 1.8, col = "blue") #The output looks like below, this graph shows the ex
# Calculating the accuracy
rmse <- sqrt(mean(pred-sales$biking)^2) # Root Mean Square Error is the standard deviation of the res
rmse
test$biking
Index


RStudio
res # Prints the residuals
# compare the predicted vs actual values
results<-cbind(pred,test$biking)
results
colnames(results)<-c('predicted','real')
results<-as.data.frame(results)
head(results)
plot(test$biking, type = 'l', lty = 1.8, col = "red")
lines(pred, type = "l", col = "blue") #The output looks like below
plot(pred, type = "l", lty = 1.8, col = "blue") #The output looks like below, this graph shows the ex
# Calculating the accuracy
rmse <- sqrt(mean(pred-sales$biking)^2) # Root Mean Square Error is the standard deviation of the res
rmse
test$biking
Index


RStudio
res # Prints the residuals
# compare the predicted vs actual values
results<-cbind(pred,test$biking)
results
colnames(results)<-c('predicted','real')
results<-as.data.frame(results)
head(results)
plot(test$biking, type = 'l', lty = 1.8, col = "red")
lines(pred, type = "l", col = "blue") #The output looks like below
plot(pred, type = "l", lty = 1.8, col = "blue") #The output looks like below, this graph shows the ex
# Calculating the accuracy
rmse <- sqrt(mean(pred-sales$biking)^2) # Root Mean Square Error is the standard deviation of the res
rmse
pred
Index


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
MLexp8.R
expEight.R
expEi.R
exp9.R
train
Source on Save
Run
Source
50 res # Prints the residuals
51 # compare the predicted vs actual values
52
53 results<-cbind(pred,test$biking)
54
55 results
56
57 colnames(results)<-c('predicted','real')
58
59 results<-as.data.frame(results)
60
61 head(results)
62 plot(test$biking, type = 'l', lty = 1.8, col = "red")
63 lines(pred, type = "l", col = "blue") #The output looks like below
64 plot(pred, type = "l", lty = 1.8, col = "blue") #The output looks like below, this graph shows the ex
65 # Calculating the accuracy
66
67 rmse <- sqrt(mean(pred-income$biking)^2) # Root Mean Square Error is the standard deviation of the re
68
69 rmse
69:1
(Top Level)
R Script
Console
Terminal
Jobs
R 4.2.0 · ~/
predicted real
1 26.559276 30.801246
2 62.555870 65.129215
3 5.965777 1.959665
4 43.687733 44.800196
5 68.674452 69.428454
6 53.526527 54.403626
> plot(test$biking, type = 'l', lty = 1.8, col = "red")
> lines(pred, type = "l", col = "blue") #The output looks like below
> plot(pred, type = "l", lty = 1.8, col = "blue")
> plot(test$biking, type = 'l', lty = 1.8, col = "red")
> lines(pred, type = "l", col = "blue") #The output looks like below
> plot(pred, type = "l", lty = 1.8, col = "blue")
> rmse <- sqrt(mean(pred-sales$biking)^2)
Error in mean(pred - sales$biking) : object 'sales' not found
> rmse <- sqrt(mean(pred-income$biking)^2)
> rmse
[1] 1.342946e-14
>

Experiment 10

Aim: Using Python (Anaconda or Pycharm) software, implement a linear model for regression or Neural Network.

Steps: Most of the machine learning algorithms are actually quite simple since they need to be in order to scale to large datasets. The math involved is typically linear algebra, but I will do my best to still explain all of the math. You will also need Scikit-Learn and Pandas installed, along with others that we'll grab along the way. You can find formulas, charts, equations, and a bunch of theory on the topic of machine learning, but very little on the actual "machine" part, where you actually program the machine and run the algorithms on real data. This is mainly due to the history. In the 50s, machines were quite weak, and in very little supply, which remained very much the case for half a century. Machine Learning was relegated to being mainly theoretical and rarely actually employed. The Support Vector Machine (SVM), for example, was created by Vladimir Vapnik in the Soviet Union in 1963, but largely went unnoticed until the 90s when Vapnik was scooped out the Soviet Union to the United States by Bell Labs. The neural network was conceived in the 1940s, but computers at the time were nowhere near powerful enough to run them well, and have not been until the relatively recent times.

Backpropagation Model (Algorithm)

Using just logistic regression we were able to hit a classification accuracy of about 97.5%, which is reasonably good but pretty much maxes out what we can achieve with a linear model. In this blog post, we'll again tackle the hand-written digits data set, but this time using a feed-forward neural network with backpropagation. We'll implement un-regularized and regularized versions of the neural network cost function and compute gradients via the backpropagation algorithm. Finally, we'll run the algorithm through an optimizer and evaluate the performance of the network on the handwritten digits data set. Since the data set is the same one we used in the last exercise, we'll re-use the code from last time to load the data.

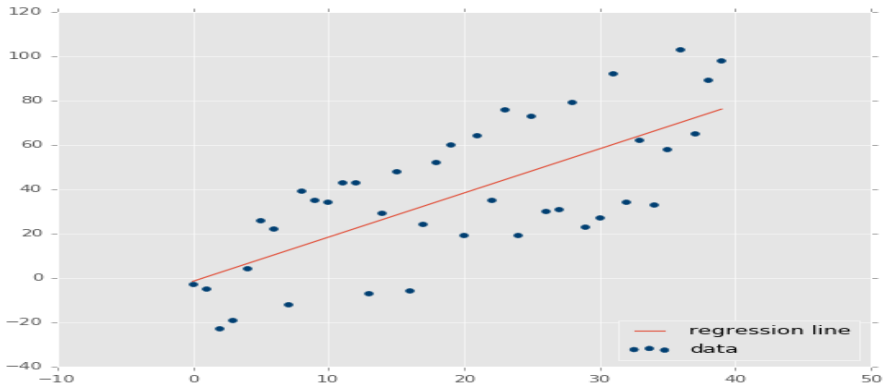
```
import numpy as np
import pandas as PD
```

If you have a pre-compiled scientific distribution of Python like ActivePython from our sponsor, you should already have numpy, scipy, scikit-learn, matplotlib, and pandas installed. If not, do:

```
pip install numpy
pip install scipy
pip install scikit-learn
pip install matplotlib
pip install pandas
```

Along with those tutorial-wide imports, we're also going to be making use of Quandl here, which you may need to separately install, with pip install quandl

To begin, what is a regression in terms of us using it with machine learning? The goal is to take continuous data, find the equation that best fits the data, and be able to forecast out a specific value. With simple linear regression, you are just simply doing this by creating the best fit line:



From here, we can use the equation of that line to forecast out into the future, where the 'date' is the x-axis, what the price will be. A popular use with **regression is to predict stock prices**. This is done because we are considering the fluidity of price over time, and attempting to forecast the next fluid price in the future using a continuous dataset. Regression is a form of supervised machine learning, which is where the scientist teaches the machine by showing its features and then showing it what the correct answer is, over and over, to teach the machine. Once the machine is taught, the scientist will usually "test" the machine on some unseen data, where the scientist still knows what the correct answer is, but the machine doesn't. The machine's answers are compared to the known answers, and the machine's accuracy can be measured. If the accuracy is high enough, the scientist may consider actually employing the algorithm in the real world. Since regression is so popularly used with stock prices, we can start there with an example. To begin, we need data. Sometimes the data is easy to acquire, and sometimes you have to go out and scrape it together, like what we did in an older tutorial series using machine learning with stock fundamentals for investing. In our case, we're able to at least start with simple stock price and volume information from Quandl. To begin, we'll start with data that grabs the stock price for Alphabet (previously Google), with the ticker of GOOGL:

```
import pandas as pd
import Quandl
df = Quandl.get("WIKI/GOOGL")
print(df.head())
```

Note: when filmed, Quandl's module was referenced with an upper-case Q, now it is a lower-case q, so import quandl. At this point, we have:

```
              Open    High     Low   Close    Volume  Ex-Dividend  \
Date
2004-08-19  100.00  104.06   95.96  100.34  44659000            0
2004-08-20  101.01  109.08  100.50  108.31  22834300            0
2004-08-23  110.75  113.48  109.05  109.40  18256100            0
2004-08-24  111.24  111.60  103.57  104.87  15247300            0
2004-08-25  104.96  108.00  103.88  106.00   9188600            0

            Split Ratio  Adj. Open  Adj. High  Adj. Low  Adj. Close  \
Date
2004-08-19            1     50.000      52.03    47.980      50.170
2004-08-20            1     50.505      54.54    50.250      54.155
2004-08-23            1     55.375      56.74    54.525      54.700
2004-08-24            1     55.620      55.80    51.785      52.435
2004-08-25            1     52.480      54.00    51.940      53.000

            Adj. Volume
Date
2004-08-19     44659000
2004-08-20     22834300
2004-08-23     18256100
2004-08-24     15247300
2004-08-25      9188600
```

Awesome, off to a good start, we have the data, but maybe a bit much. To reference the intro, there exists an entire machine learning category that aims to reduce the amount of input that we process. In our case, we have quite a few columns, many are redundant, a couple don't really change. We can most likely agree that having both the regular columns and adjusted columns is redundant. Adjusted columns are the most ideal ones. Regular columns here are prices on the day, but stocks have things called stock splits, where suddenly 1 share becomes something like 2 shares, thus the value of a share is halved, but the value of the company has not halved. Adjusted columns are adjusted for stock splits over time, which makes them more reliable for doing the analysis.

Thus, let's go ahead and pair down our original dataframe a bit:

```
df = df[['Adj. Open',  'Adj. High',  'Adj. Low',  'Adj. Close', 'Adj. Volume']]
```

Now we just have the adjusted columns and the volume column. A couple of major points to make here. Many people talk about or hear about machine learning as if it is some sort of dark art that somehow generates value from nothing. Machine learning can highlight value if it is there, but it has to actually be there. You need meaningful data. So how do you know if you have meaningful data? My best suggestion is to just simply use your brain. Think about it. Are historical prices indicative of future prices? Some people think so, but this has been continually disproven over time. What about historical patterns? This has a bit more merit when taken to the extremes (which machine learning can help with), but is overall fairly weak. What about the relationship between price changes and volume over time, along with historical patterns? Probably a bit better. So, as you can already see, it is not the case that the more data the merrier, but we instead want to use user data. At the same time, raw data sometimes should be transformed.

```
import matplotlib.pyplot as plot
from scipy.io import loadmat
%matplotlib inline

data = loadmat('data/ex3data1.mat')
data
{
```

The neural network we're going to build for this exercise has an input layer matching the size of our instance data (400 + the bias unit), a hidden layer with 25 units (26 with the bias unit), and an output layer with 10 units corresponding to our one-hot encoding for the class labels. The first piece we need to implement is a cost function to evaluate the loss for a given set of network parameters. The source mathematical function is in the exercise text and looks pretty intimidating, but it helps to really break it down into pieces. Here are the functions required to compute the cost.

```
def sigmoid(z):
return 1 / (1 + np.exp(-z))
def forward_propagate(X, theta1, theta2):
m = X.shape[0]
a1 = np.insert(X, 0, values=np.ones(m), axis=1)
z2 = a1 * theta1.T
a2 = np.insert(sigmoid(z2), 0, values=np.ones(m), axis=1)
z3 = a2 * theta2.T
h = sigmoid(z3)
return a1, z2, a2, z3, h
def cost(params, input_size, hidden_size, num_labels, X, y, learning_rate):
m = X.shape[0]
X = np.matrix(X)
y = np.matrix(y)
# reshape the parameter array into parameter matrices for each layer
theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_size,
(input_size + 1))))
theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_labels,
(hidden_size + 1))))
a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)
J = 0
for i in range(m):
first_term = np.multiply(-y[i,:], np.log(h[i,:]))
second_term = np.multiply((1 - y[i,:]), np.log(1 - h[i,:]))
J += np.sum(first_term - second_term)
J = J / m
return J
```

Our next step is to **add regularization to** the cost function, which adds a penalty term to the cost that scales with the magnitude of the parameters. The equation for this looks pretty ugly, but it can be boiled down to just one line of code added to the original cost function. Just add the following right before the return statement.

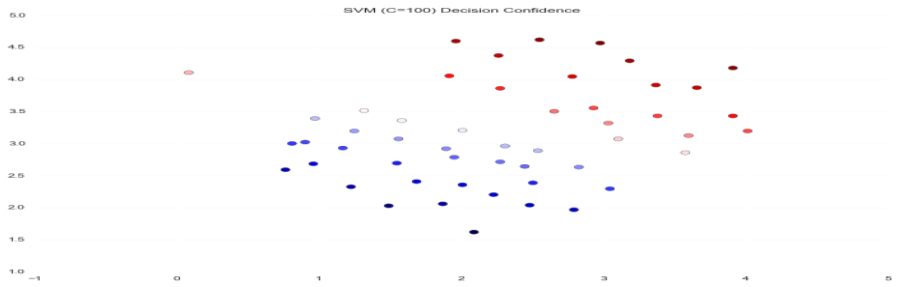
J+=(float(learning_rate)/(2*m))*(np.sum(np.power(theta1[:,1:],2))+np.sum(np.power(theta2[:,1:], 2)))

Next up is the backpropagation algorithm. Backpropagation computes the parameter updates that will reduce the error of the network on the training data. The first thing we need is a function that computes the gradient of the sigmoid function we created earlier.

```
def sigmoid_gradient(z):
    return np.multiply(sigmoid(z), (1 - sigmoid(z)))
```

Now we're ready to implement backpropagation to compute the gradients. Since the computations required for backpropagation are a superset of those required in the cost function, we're actually going to extend the cost function to also perform backpropagation and return both the cost and the gradients. If you're wondering why I'm not just calling the existing cost function from within the backprop function to make the design more modular, it's because backprop uses a number of other variables calculated inside the cost function. Here's the full implementation. I skipped ahead and added gradient regularization rather than first create an un-regularized version.

```
def backprop(params, input_size, hidden_size, num_labels, X, y, learning_rate):
##### this section is identical to the cost function logic we already saw #####
m = X.shape[0]
X = np.matrix(X)
y = np.matrix(y)
# reshape the parameter array into parameter matrices for each layer
theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_size,
(input_size + 1))))
theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_labels,
(hidden_size + 1))))
# run the feed-forward pass
a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)
J = 0
delta1 = np.zeros(theta1.shape)  # (25, 401)
delta2 = np.zeros(theta2.shape)  # (10, 26)
# compute the cost
for i in range(m):
first_term = np.multiply(-y[i,:], np.log(h[i,:]))
second_term = np.multiply((1 - y[i,:]), np.log(1 - h[i,:]))
J += np.sum(first_term - second_term)
J = J / m
# add the cost regularization term
J += (float(learning_rate) / (2 * m)) * (np.sum(np.power(theta1[:,1:], 2)) +
np.sum(np.power(theta2[:,1:], 2)))
##### end of cost function logic, below is the new part #####
# perform backpropagation
for t in range(m):
a1t = a1[t,:]  # (1, 401)
z2t = z2[t,:]  # (1, 25)
a2t = a2[t,:]  # (1, 26)
ht = h[t,:]  # (1, 10)
yt = y[t,:]  # (1, 10)
d3t = ht - yt  # (1, 10)
z2t = np.insert(z2t, 0, values=np.ones(1))  # (1, 26)
d2t = np.multiply((theta2.T * d3t.T).T, sigmoid_gradient(z2t))  # (1, 26)
delta1 = delta1 + (d2t[:,1:]).T * a1t
delta2 = delta2 + d3t.T * a2t
delta1 = delta1 / m
delta2 = delta2 / m
# add the gradient regularization term
delta1[:,1:] = delta1[:,1:] + (theta1[:,1:] * learning_rate) / m
delta2[:,1:] = delta2[:,1:] + (theta2[:,1:] * learning_rate) / m
# unravel the gradient matrices into a single array
grad = np.concatenate((np.ravel(delta1), np.ravel(delta2)))
return J, grad
```



We're going to be working first with the MNIST dataset, which is a dataset that contains 60,000 training samples and 10,000 testing samples of hand-written and labeled digits, 0 through 9, so ten total "classes." I will note that this is a very small dataset in terms of what you would be working within any realistic setting, but it should also be small enough to work on everyone's computers.

The MNIST dataset has the images, which we'll be working with as purely black and white, thresholded, images, of size 28 x 28, or 784 pixels total. Our features will be the pixel values for each pixel, thresholded. Either the pixel is "blank" (nothing there, a 0), or there is something there (1). Those are our features. We're going to attempt to just use this extremely rudimentary data, and predict the number we're looking at (a 0,1,2,3,4,5,6,7,8, or 9). We're hoping that our neural network will somehow create an inner-model of the relationships between pixels, and be able to look at new examples of digits and predict them to a high degree. While the code here will not be all that long, it can be quite confusing if you're not fully understanding what is supposed to be happening, so let's try to condense what we've learned so far, and what we're going to be doing here.

First, we take our input data, and we need to send it to hidden layer 1. Thus, we weigh the input data, and send it to layer 1, where it will undergo the activation function, so the neuron can decide whether or not to fire and output some data to either the output layer or another hidden layer. We will have three hidden layers in this example, making this a Deep Neural Network. From the output we get, we will compare that output to the intended output. We will use a cost function (alternatively called a loss function), to determine how wrong we are. Finally, we will use an optimizer function, Adam Optimizer in this case, to minimize the cost (how wrong we are). The way cost is minimized is by tinkering with the weights, with the goal of hopefully lowering the cost. How quickly we want to lower the cost is determined by the learning rate. The lower the value for learning rate, the slower we will learn, and the more likely we'll get better results. The higher the learning rate, the quicker we will learn, giving us faster training times, but also may suffer on the results. There are diminishing returns here, you cannot just keep lowering the learning rate and always do better, of course. The act of sending the data straight through our network means we're operating a feed-forward neural network. The adjusting of weights backward is our backpropagation.

We do this feeding forward and back propagation however many times we want. The cycle is called an epoch. We can pick any number we like for the number of epochs, but you would probably want to avoid too many, causingover fitment.

After each epoch, we've hopefully further fine-tuned our weights, lowering our cost and improving accuracy. When we've finished all of the epochs, we can test using the testing set.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot = True)
```

We import TensorFlow and the sample data we are going to use. Note the one_hot parameter there. The term comes from electronics where just one element, out of the others, is literally "hot," or on. This is useful for multi-class classification tasks, which we have here (0,1,2,3,4,5,6,7,8, or 9). Thus, rather than a 0's output being just a 0 and a 1 a 1, we have something more like:

```
0 = [1,0,0,0,0,0,0,0,0]
1 = [0,1,0,0,0,0,0,0,0]
2 = [0,0,1,0,0,0,0,0,0]
3 = [0,0,0,1,0,0,0,0,0]
...
```

I chose to use the MNIST dataset because it's a decent dataset to start with, and actually collecting raw data and converting it to something to work with can take more time than creating the machine learning model itself, and I think most people here want to learn neural networks, not web scraping and regular expressions.

Now we're going to begin building the model:

```
n_nodes_hl1 = 500
n_nodes_hl2 = 500
n_nodes_hl3 = 500
n_classes = 10
batch_size = 100
```

We begin by specifying how many nodes each hidden layer will have, how many classes our dataset has, and what our batch size will be. While you *can* in theory, train the entire network all at once, it's impractical. Many of you probably have computers that can handle the MNIST dataset in full, but most of you do not have computers, or access to computers, that can do realistically sized datasets all at once. Thus, we do the optimization in batches. In this case, we will do batches of 100.

```
x = tf.placeholder('float', [None, 784])
y = tf.placeholder('float')
```

These are our placeholders for some values in our graph. Recall that you simply build the model in your TensorFlow graph. From there, TensorFlow manipulates everything, you do not. This will be even more obvious once we finish and you try to look for where we modify weights! Notice that I have used [None,784] as a 2nd parameter in the first placeholder. This is an optional parameter. It can be useful, however, to be explicit like this. If you are not explicit, TensorFlow will stuff anything in there. If you are explicit about the shape,

TensorFlow will throw an error if something out of shape attempts to hop into that variable's place.

We're now complete with our constants and starting values. Now we can actually build the Neural Network Model:

```
def neural_network_model(data):
hidden_1_layer = {'weights':tf.Variable(tf.random_normal([784, n_nodes_hl1])),
'biases':tf.Variable(tf.random_normal([n_nodes_hl1]))}

hidden_2_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl1, n_nodes_hl2])),
'biases':tf.Variable(tf.random_normal([n_nodes_hl2]))}

hidden_3_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl2, n_nodes_hl3])),
'biases':tf.Variable(tf.random_normal([n_nodes_hl3]))}

output_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl3, n_classes])),
'biases':tf.Variable(tf.random_normal([n_classes]))}
```

Here, we begin defining our weights and our... HOLD on, wait a sec, what are these biases!? The bias is a value that is added to our sums, before being passed through the activation function, not to be confused with a bias node, which is just a node that is always on. The purpose of the bias here is mainly to handle scenarios where all neurons fired a 0 into the layer. A bias makes it possible that a neuron still fires out of that layer. Bias is as unique as the weights and will need to be optimized too.

All we've done so far is create a starting definition for our weights and biases. These definitions are just random values, for the shape that the layer's matrix should be (this is what tf.random_normal does for us, it outputs random values for the shape we want). Nothing has actually happened yet, and no flow (feed forward) has occurred yet. Let's start the flow:

```
    l1 = tf.add(tf.matmul(data,hidden_1_layer['weights']), hidden_1_layer['biases'])
    l1 = tf.nn.relu(l1)

    l2 = tf.add(tf.matmul(l1,hidden_2_layer['weights']), hidden_2_layer['biases'])
    l2 = tf.nn.relu(l2)

    l3 = tf.add(tf.matmul(l2,hidden_3_layer['weights']), hidden_3_layer['biases'])
    l3 = tf.nn.relu(l3)

    output = tf.matmul(l3,output_layer['weights']) + output_layer['biases']

    return output
```

Here, we take values into layer one. What are the values? They are the multiplication of the raw input data multiplied by their unique weights (starting as random but will be optimized): tf.matmul (l1,hidden_2_layer['weights']). We then are adding, with tf.add the bias. We repeat this process for each of the hidden layers, all the way down to our output, where we have the final values still being the multiplication of the input and the weights, plus the output layer's bias values.

Beyond List experiments:

Experiment 1

Aim: Understanding of RMS Titanic Dataset to predict survival by training a model and predict the required solution.

Steps:

The sinking of the **RMS Titanic** is one of the most infamous shipwrecks in history. On April 15, 1912, during her maiden voyage, the Titanic sank after colliding with an iceberg, killing 1502 out of 2224 passengers and crew. This sensational tragedy shocked the international community and led to better safety regulations for ships. One of the reasons that the shipwreck led to such loss of life was that there were not enough lifeboats for the passengers and crew. Although there was some element of luck involved in surviving the sinking, some groups of people were more likely to survive than others, such as women, children, and the upper-class.

	Survived	Pclass	Sex	Age	Fare	Embarked	Title	IsAlone	Age*Class
0	0	3	0	1	0	0	1	0	3
1	1	1	1	2	3	1	3	0	2
2	1	3	1	1	1	0	2	1	3
3	1	1	1	2	3	0	3	0	2
4	0	3	0	2	1	0	1	1	6
5	0	3	0	1	1	2	1	1	3
6	0	1	0	3	3	0	1	1	3
7	0	3	0	0	2	0	4	0	0
8	1	3	1	1	1	0	3	0	3
9	1	2	1	0	2	1	3	0	0

Survived (Target Variable) - Binary categorical variable where 0 represents not survived and 1 represents survived.
Pclass - Categorical variable. It is a passenger class.
Sex - Binary Variable representing the gender the of passenger
Age - Feature engineered variable. It is divided into 4 classes.
Fare - Feature engineered variable. It is divided into 4 classes.
Embarked - Categorical Variable. It tells the Port of embarkation.
Title - New feature created from names. The title of names is classified into 4 different classes.
isAlone - Binary Variable. It tells whether the passenger is traveling alone or not.
Age*Class - Feature engineered variable.

Model, predict and solve

Now we are ready to train a model and predict the required solution. There are 60+ predictive modeling algorithms to choose from. We must understand the type of problem and solution requirement to narrow down to a select few models which we can evaluate. Our problem is a classification and regression problem. We want to identify the relationship between output (Survived or not) with other variables or features (Gender, Age, Port...). We are also performing a category of machine learning which is called supervised learning as we

are training our model with a given dataset. With these two criteria - Supervised Learning plus Classification and Regression, we can narrow down our choice of models to a few. These include:

- Logistic Regression
- KNN or k-Nearest Neighbours
- Support Vector Machines
- Naive Bayes classifier
- Decision Tree

```
X_train = train_df.drop("Survived", axis=1)
Y_train = train_df["Survived"]
X_test  = test_df.drop("PassengerId", axis=1).copy()
X_train.shape, Y_train.shape, X_test.shape
```

```
((891, 8), (891,), (418, 8))
```

Size of the training and testing dataset

Logistic Regression is a useful model to run early in the workflow. Logistic regression measures the relationship between the categorical dependent variable (feature) and one or more independent variables (features) by estimating probabilities using a logistic function, which is the cumulative logistic distribution.

```
# Logistic Regression

logreg = LogisticRegression()
logreg.fit(X_train, Y_train)
Y_pred = logreg.predict(X_test)
acc_log = round(logreg.score(X_train, Y_train) * 100, 2)
acc_log
```

```
80.359999999999999
```

Note the confidence score generated by the model based on our training dataset.

In pattern recognition, the k-Nearest Neighbours algorithm (or k-NN for short) is a non-parametric method used for classification and regression. A sample is classified by a majority vote of its neighbors, with the sample being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor.
KNN confidence score is better than Logistics Regression but worse than SVM.

```
knn = KNeighborsClassifier(n_neighbors = 3)
knn.fit(X_train, Y_train)
Y_pred = knn.predict(X_test)
acc_knn = round(knn.score(X_train, Y_train) * 100, 2)
acc_knn
```

```
84.739999999999995
```

Next, we model using Support Vector Machines which are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training samples, each marked as belonging to one or the other of **two categories**, an SVM training algorithm builds a model that assigns new test samples to one category or the other, making it a non-probabilistic binary linear classifier.

```
# Support Vector Machines

svc = SVC()
svc.fit(X_train, Y_train)
Y_pred = svc.predict(X_test)
acc_svc = round(svc.score(X_train, Y_train) * 100, 2)
acc_svc
```

```
83.840000000000003
```

Note that the model generates a confidence score which is higher than the Logistics Regression model.

In machine learning, naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Naive Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features) in a learning problem.
The model generated confidence score is the lowest among the models evaluated so far.

```
# Gaussian Naive Bayes

gaussian = GaussianNB()
gaussian.fit(X_train, Y_train)
Y_pred = gaussian.predict(X_test)
acc_gaussian = round(gaussian.score(X_train, Y_train) * 100, 2)
acc_gaussian
```

```
72.280000000000001
```

This model uses a decision tree as a predictive model which maps features (tree branches) to conclusions about the target value (tree leaves). Tree models where the target variable can take a finite set of values are called classification trees; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values (typically real numbers) are called regression trees. The model confidence score is the highest among models evaluated so far.

```
# Decision Tree

decision_tree = DecisionTreeClassifier()
decision_tree.fit(X_train, Y_train)
Y_pred = decision_tree.predict(X_test)
acc_decision_tree = round(decision_tree.score(X_train, Y_train) * 100, 2)
acc_decision_tree
```

```
86.760000000000005
```

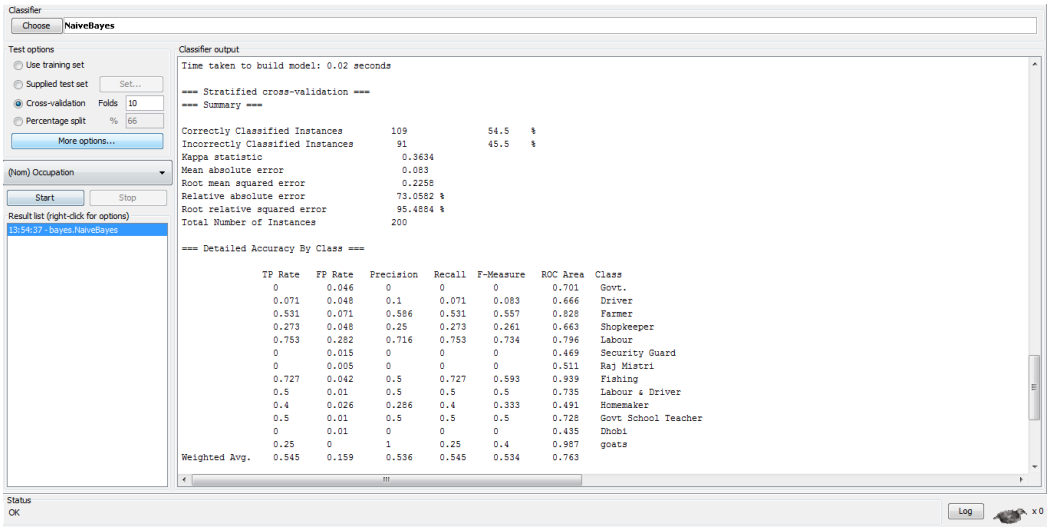
Experiment 2

Aim: Understanding of Indian education in Rural villages to predict whether a girl child will be sent to school or not.

Steps:

The data is focused on rural India. It primarily looks into the fact whether the villagers are willing to send the girl children to school or not and if they are not sending their daughters to school the reasons have also been mentioned. The district is Gwalior. Various details of the villagers such as village, gender, age, education, occupation, category, caste, religion, land, etc. have also been collected.

Naïve Bayes Classifier

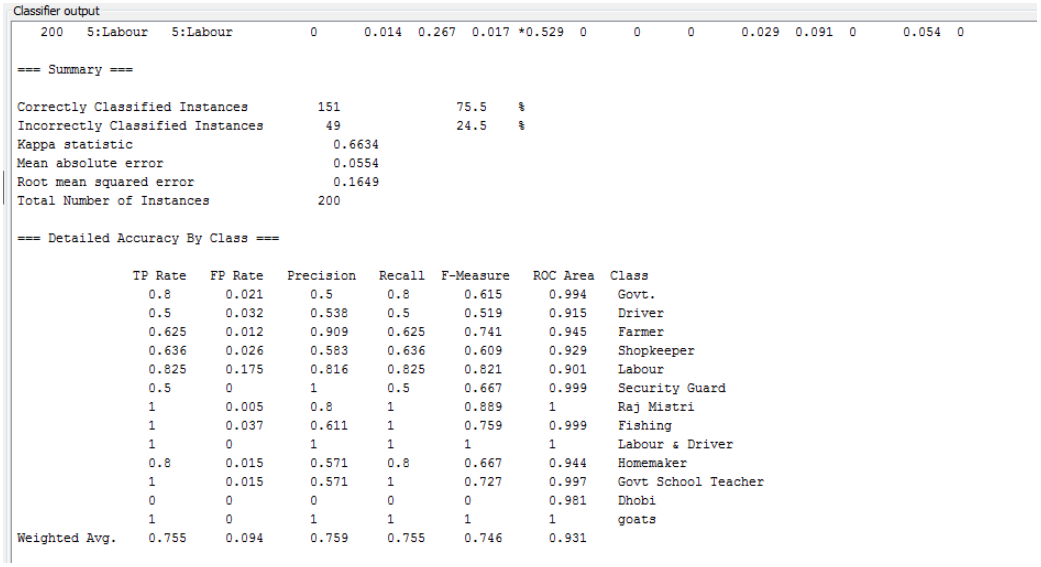


The algorithm was run with 10-fold cross-validation: this means it was given an opportunity to make a prediction for each instance of the dataset (with different training folds) and the presented result is a summary of those predictions. Firstly, I noted the Classification Accuracy. The model achieved a result of 109/200 correct or 54.5%.

```
=== Confusion Matrix ===

  a  b  c  d  e  f  g  h  i  j  k  l  m   <-- classified as
  0  0  1  1  0  1  0  2  0  0  0  0  0 |  a = Govt.
  2  1  1  1  8  0  0  0  0  1  0  0  0 |  b = Driver
  2  0 17  2  9  0  0  2  0  0  0  0  0 |  c = Farmer
  0  0  4  3  2  0  1  0  0  1  0  0  0 |  d = Shopkeeper
  1  8  2  3 73  1  0  1  1  3  2  2  0 |  e = Labour
  3  0  0  0  0  0  0  1  0  0  0  0  0 |  f = Security Guard
  0  1  0  1  0  0  0  2  0  0  0  0  0 |  g = Raj Mistri
  1  0  0  0  1  1  0  8  0  0  0  0  0 |  h = Fishing
  0  0  2  0  0  0  0  0  2  0  0  0  0 |  i = Labour & Driver
  0  0  2  0  1  0  0  0  0  2  0  0  0 |  j = Homemaker
  0  0  0  0  1  0  0  0  1  0  2  0  0 |  k = Govt School Teacher
  0  0  0  1  4  0  0  0  0  0  0  0  0 |  l = Dhobi
  0  0  0  0  3  0  0  0  0  0  0  0  1 |  m = goats
```

The confusion matrix shows the precision of the algorithm showing that 1,1,1,2 Government officials were misclassified as Farmer, Shopkeeper, Security Guard, and Fishermen respectively, 2,1,1,8,1 Drivers were misclassified as Government officials, Farmer, Shopkeeper, Labour, Homemaker, and so on. This table can help to explain the accuracy achieved by the algorithm.
Now when we have a model, we need to load the test data we've created before. For this, select Supplied test set and click button Set. Click More Options, wherein a new window, choose Plain Text from Output predictions. Then click left mouse button on the recently created model on result list and select Re-evaluate model on the current test set.
After re-evaluation



Classifier output

```
   200   5:Labour   5:Labour         0      0.014  0.267  0.017 *0.529  0      0      0      0.029  0.091  0      0.054  0

=== Summary ===

Correctly Classified Instances         151               75.5    %
Incorrectly Classified Instances        49               24.5    %
Kappa statistic                          0.6634
Mean absolute error                      0.0554
Root mean squared error                  0.1649
Total Number of Instances              200

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.8       0.021      0.5       0.8       0.615      0.994    Govt.
                 0.5       0.032      0.538     0.5       0.519      0.915    Driver
                 0.625     0.012      0.909     0.625     0.741      0.945    Farmer
                 0.636     0.026      0.583     0.636     0.609      0.929    Shopkeeper
                 0.825     0.175      0.816     0.825     0.821      0.901    Labour
                 0.5       0          1         0.5       0.667      0.999    Security Guard
                 1         0.005      0.8       1         0.889      1        Raj Mistri
                 1         0.037      0.611     1         0.759      0.999    Fishing
                 1         0          1         1         1          1        Labour & Driver
                 0.8       0.015      0.571     0.8       0.667      0.944    Homemaker
                 1         0.015      0.571     1         0.727      0.997    Govt School Teacher
                 0         0          0         0         0          0.981    Dhobi
                 1         0          1         1         1          1        goats
Weighted Avg.    0.755     0.094      0.759     0.755     0.746      0.931
```

Now the Classification Accuracy is 151/200 correct or 75.5%.
TP = true positives: number of examples predicted positive that are actually positive
FP = false positives: number of examples predicted positive that are actually negative
TN = true negatives: number of examples predicted negative that are actually negative
FN = false negatives: number of examples predicted negative that are actually positive

The recall is the TP rate (also referred to as sensitivity) what fraction of those that are actually positive were predicted positive? : TP / actual positives Precision is TP / predicted Positive What fraction of those predicted positive is actually positive? **Precision** is also referred to as Positive predictive value (PPV); other related measures used in classification include True Negative Rate and Accuracy: True Negative Rate is also called **Specificity**. (TN / actual negatives) 1-specificity is x-axis of ROC curve: this is the same as the FP rate (FP / actual negatives) **F-measure** A measure that combines precision and recall is the harmonic mean of precision and recall, the traditional F-measure or balanced F-score:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

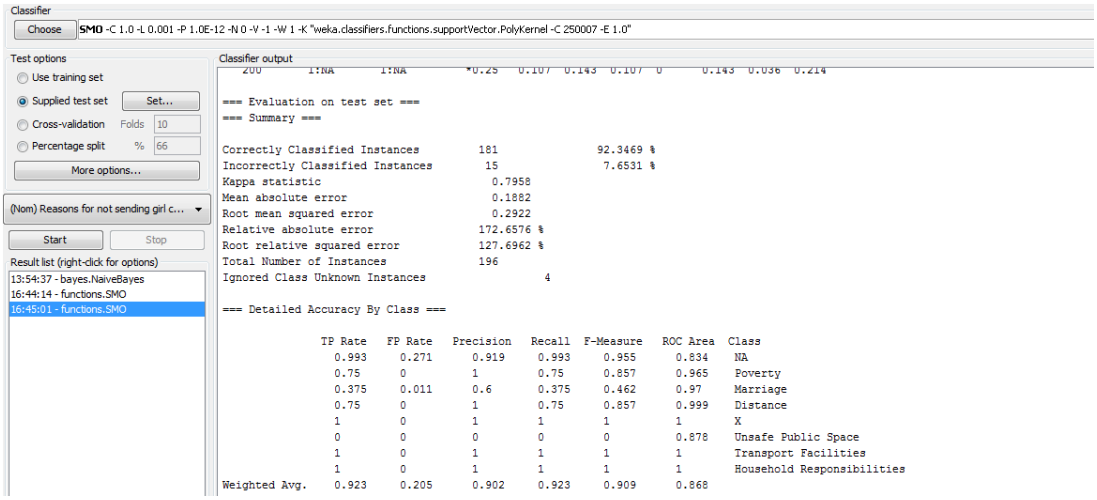
$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

Mean absolute error (MAE): The MAE measures the average magnitude of the errors in a set of forecasts, without considering their direction. It measures accuracy for continuous variables. The equation is given in the library references. Expressed in words, the MAE is the average over the verification sample of the absolute values of the differences between forecast and the corresponding observation. The MAE is a linear score which means that all the individual differences are weighted equally in the average;

Root mean squared error (RMSE): The RMSE is a quadratic scoring rule which measures the average magnitude of the error. The equation for the RMSE is given in both of the references. Expressing the formula in words, the difference between forecast and corresponding observed values are each squared and then averaged over the sample. Finally, the square root of the average is taken. Since the errors are squared before they are averaged, the RMSE gives a relatively high weight to large errors. This means the RMSE is most useful when large errors are particularly undesirable.

Support Vector Machine



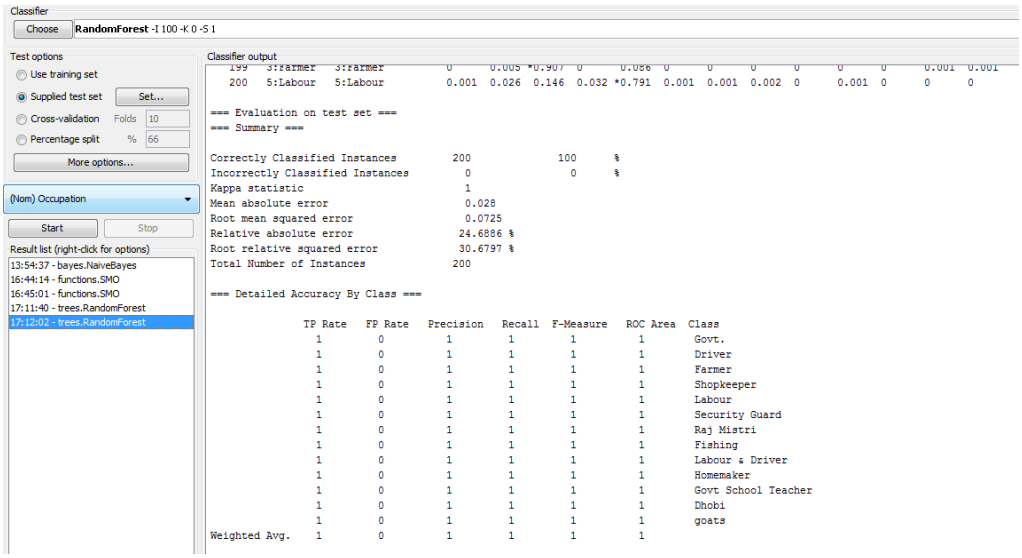
The model achieved a result of 181/200 correct or 92.3469%.
We have classified the dataset on the basis of the reasons why the villagers are unwilling to send girl children to schools in Gwalior village. The different classes are NA, Poverty, Marriage, Distance, X, Unsafe Public Space, Transport Facilities, and Household Responsibilities.
The weighted average true positive rate is 0.923 that isnear all the predicted positive values are actually positive. The weighted average false positive rate is 0.205 that is few of them are predicted as positive values but are actually negative. The precision in 0.902 that is the algorithm is nearly accurate.

```
=== Confusion Matrix ===

   a   b   c   d   e   f   g   h   <-- classified as
 147   0   1   0   0   0   0   0 |   a = NA
   4  12   0   0   0   0   0   0 |   b = Poverty
   5   0   3   0   0   0   0   0 |   c = Marriage
   0   0   1   3   0   0   0   0 |   d = Distance
   0   0   0   0   8   0   0   0 |   e = X
   4   0   0   0   0   0   0   0 |   f = Unsafe Public Space
   0   0   0   0   0   0   4   0 |   g = Transport Facilities
   0   0   0   0   0   0   0   4 |   h = Household Responsibilities
```

The confusion matrix shows that majority of the reasons were not available and out of the reasons which were available people did not send their daughters to school because of poverty and very few of them considered Distance as a major factor for not sending their girl children to school.

Random Forest



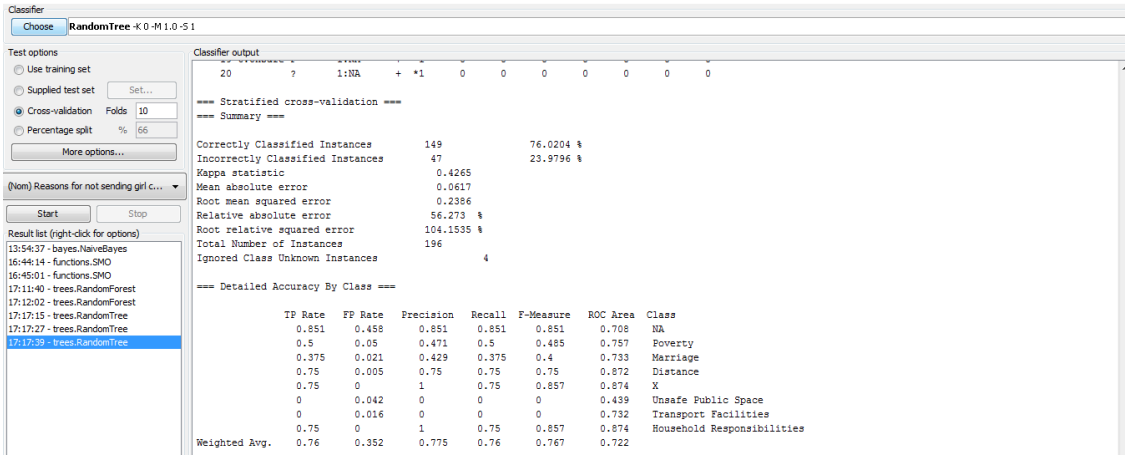
The accuracy of this algorithm is 100% that is 200/200 have been correctly classified

```
=== Confusion Matrix ===

  a  b  c  d  e  f  g  h  i  j  k  l  m   <-- classified as
  5  0  0  0  0  0  0  0  0  0  0  0  0 |  a = Govt.
  0 14  0  0  0  0  0  0  0  0  0  0  0 |  b = Driver
  0  0 32  0  0  0  0  0  0  0  0  0  0 |  c = Farmer
  0  0  0 11  0  0  0  0  0  0  0  0  0 |  d = Shopkeeper
  0  0  0  0 97  0  0  0  0  0  0  0  0 |  e = Labour
  0  0  0  0  0  4  0  0  0  0  0  0  0 |  f = Security Guard
  0  0  0  0  0  0  4  0  0  0  0  0  0 |  g = Raj Mistri
  0  0  0  0  0  0  0 11  0  0  0  0  0 |  h = Fishing
  0  0  0  0  0  0  0  0  4  0  0  0  0 |  i = Labour & Driver
  0  0  0  0  0  0  0  0  0  5  0  0  0 |  j = Homemaker
  0  0  0  0  0  0  0  0  0  0  4  0  0 |  k = Govt School Teacher
  0  0  0  0  0  0  0  0  0  0  0  5  0 |  l = Dhobi
  0  0  0  0  0  0  0  0  0  0  0  0  4 |  m = goats
```

There is no observation which has been misclassified. The maximum number of villagers are labourers.

Random Tree



The classification accuracy is 76.0204% that is 149/200 have been classified correctly.
The false positive rate is 0.352 that is highest of all the four algorithms applied above. Here 35.2% of the values which should have been classified negatively have been assigned a positive value.

```
=== Confusion Matrix ===

   a   b   c   d   e   f   g   h   <-- classified as
 126   7   3   1   0   8   3   0 |   a = NA
   7   8   1   0   0   0   0   0 |   b = Poverty
   4   1   3   0   0   0   0   0 |   c = Marriage
   1   0   0   3   0   0   0   0 |   d = Distance
   2   0   0   0   6   0   0   0 |   e = X
   4   0   0   0   0   0   0   0 |   f = Unsafe Public Space
   3   1   0   0   0   0   0   0 |   g = Transport Facilities
   1   0   0   0   0   0   0   3 |   h = Household Responsibilities
```

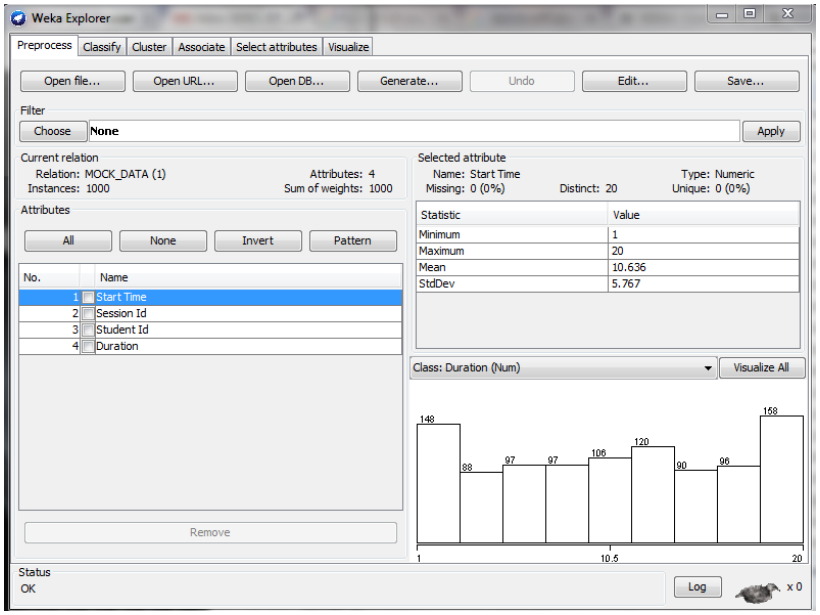
22 NA, 8 Poverty, 5 Marriage, 1 Distance, 2 X, 4 Unsafe Public Space, 4 Transport Facilities and 1 Household Responsibilities class values have been misclassified.
The best algorithm out of the above algorithms is Random Forest with 100% accuracy rate and the worst is the Naïve Bayes algorithm with 75.5% accuracy rate.

Experiment 3

Aim: Understanding of Dataset of contact patterns among students collected in the National University of Singapore.

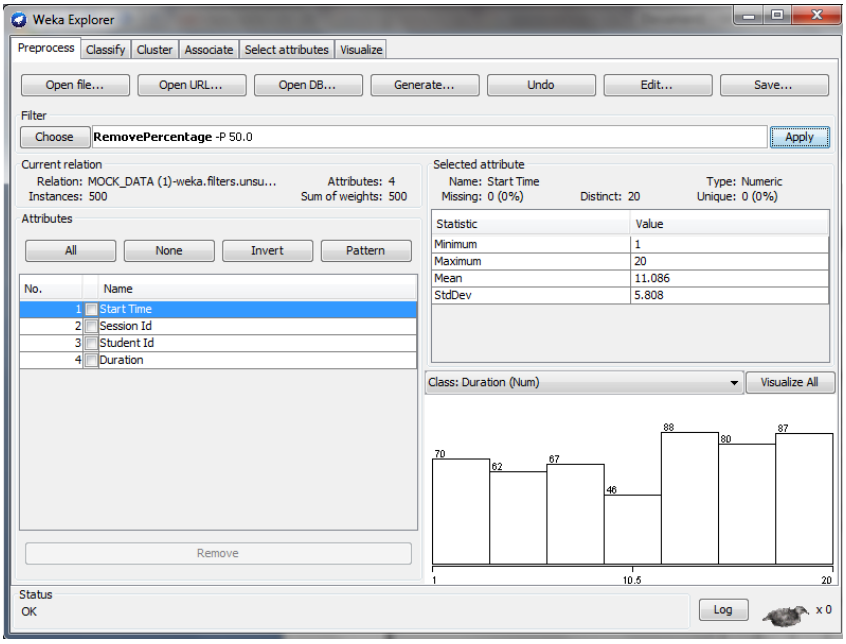
Steps:

This is dataset collected from contact patterns among students collected during the spring semester 2006 in National University of Singapore:

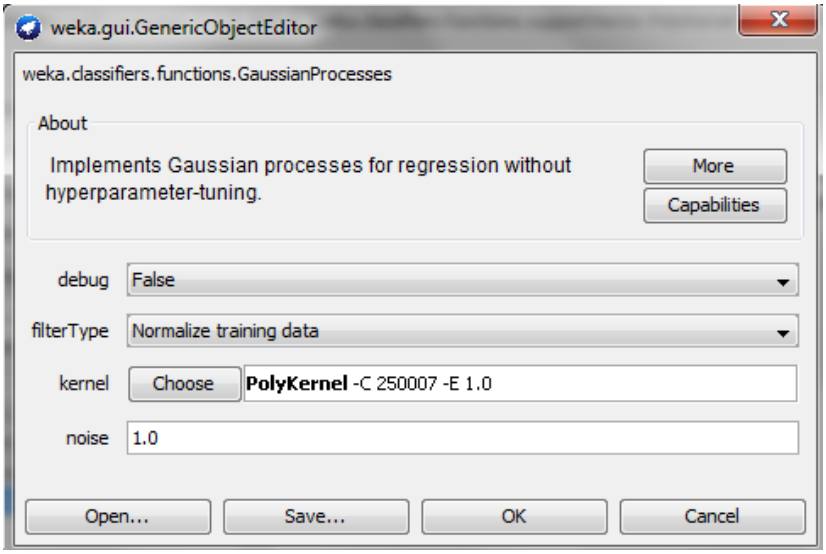


Using RemovePercentage filter, instances have been reduced to 500

This data has been taken and saved as a training data set and then used for further classification.



Algorithm 1: Gaussian processes



=== Run information ===
Scheme: weka.classifiers.functions.SimpleLinearRegression
Relation: MOCK_DATA (1)-weka.filters.unsupervised.instance.RemovePercentage-P50.0
Instances: 500
Attributes: 4
Start Time
Session Id
Student Id
Duration
Test mode: evaluate on training data

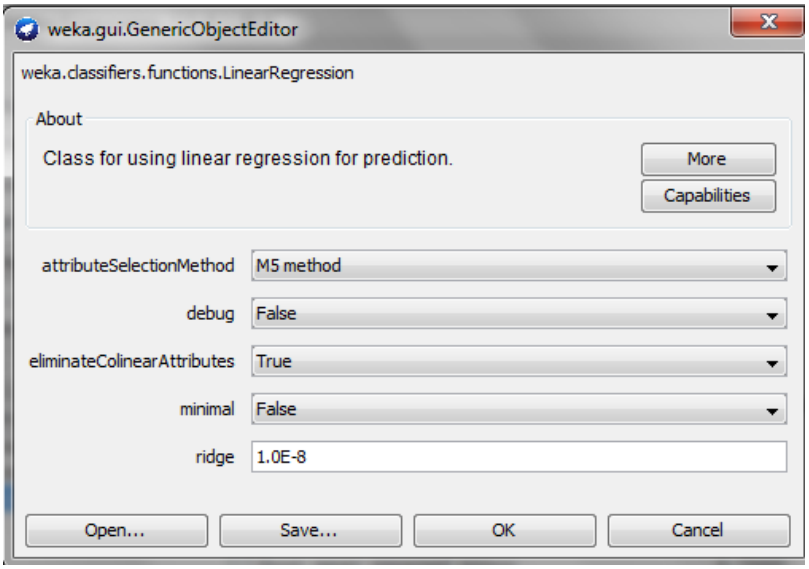
=== Classifier model (full training set) ===
Linear regression on Session Id
0.03 * Session Id + 10.38
Time is taken to build model: 0 seconds

=== Evaluation on training set ===
Time is taken to test model on training data: 0 seconds

=== Summary ===

Correlation coefficient	0.0677
Mean absolute error	4.9869
Root mean squared error	5.7893
Relative absolute error	99.7326 %
Root relative squared error	99.7708 %
Total Number of Instances	500

Algorithm 2: Linear Regression



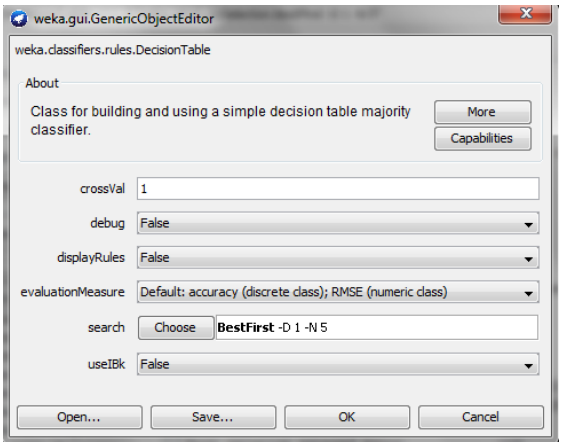
Linear Regression Model
Start Time = 0.0274 * Session Id + 10.3846
Time is taken to build model: 0.01 seconds

=== Evaluation on training set ===
Time is taken to test model on training data: 0 seconds

=== Summary ===

Correlation coefficient	0.0677
Mean absolute error	4.9869
Root mean squared error	5.7893
Relative absolute error	99.7326 %
Root relative squared error	99.7708 %
Total Number of Instances	500

Algorithm 3: Decision Table



Merit of best subset found: 5.814
Evaluation (for feature selection): CV (leave one out)
Feature set: 1

Time taken to build model: 0.02 seconds

=== Evaluation on training set ===
Time taken to test model on training data: 0 seconds

=== Summary ===

Correlation coefficient	0
Mean absolute error	5.0003
Root mean squared error	5.8026
Relative absolute error	100%
Root relative squared error	100%
Total Number of Instances	500

CONCLUSION:

Six algorithms have been used to measure the best classifier. Depending on various attributes, a performance of various algorithms can be measured via mean absolute error and correlation coefficient.

Depending on the results above, the worst correlation has been found by Decision Table and best correlation has been found by Decision Stump.

=== Run information ===
Scheme: weka.classifiers.rules.DecisionTable -X 1 -S "weka.attributeSelection.BestFirst -D 1 -N 5"
Relation: MOCK_DATA (1)-weka.filters.unsupervised.instance.RemovePercentage-P50.0
Instances: 500
Attributes: 4
Start Time
Session Id
Student Id
Duration
Test mode: evaluate on training data

=== Classifier model (full training set) ===
Decision Table:
Number of training instances: 500
Number of Rules: 1
Non matches covered by Majority class.

Best first.
Start set: no attributes
Search direction: forward
Stale search after 5 node expansions
Total number of subsets evaluated: 9