MA - 471

STOCK PRICE PREDICTION MODEL

[Stock Modelling App Using R Shiny]

Introduction

Group 11 members -

- Aayush Bansal (170123001)
- Aditya Raj (170123004)
- Aryan Raj (170123010)
- Mayank Saharan (170123033)
- Sumedh Jouras (170123050)

In this project we have created a stock model with the help of R shiny and R shiny dashboard which predicts the future price of the corresponding stock. We have taken time series data of stocks of 1 year (2019-2020) and used an Autoregressive integrated moving average (ARIMA) model and forecasted it, we have also used accuracy parameters to check the output.

Link to the code -
https://drive.google.com/drive/u/0/folders/1QiUJw3ctIuEVi83GcLC6lX3_ExM7Wd1g

What is R language?

R is a language and environment for statistical computing and graphics. It is a GNU Project which is similar to the S language and environment which was developed at Bell Laboratories. Nowadays, it is used by almost all the tech giants. The unique feature which makes it unique and specifically useful for analysts and statisticians is the notion of vectors. These vectors allow you to perform sometimes complex operations on a set of values in a single command. Programming constructs like vectors, lists, frames, data tables, matrices etc. perform transformations on data in bulk.Since,R is open source and free, a lot of packages have been developed for it which simplifies the task of the users. One such package in R is **shiny**.

What is the Shiny package in R?

Shiny is an open source R package that provides an elegant and powerful web framework for building web applications using R. It allows us to amalgamate the statistical power of R and the interactivity of the modern web. Shiny saves space and time in the construction, automation, and distribution of data visualizations and statistical analysis. It does not require HTML, CSS, or JavaScript knowledge.

```
library(shiny) # start by loading the necessary packages (shiny is always necessary)

ui <- fluidPage( # lay out the UI with a ui object that controls the appearance of our app

  )

server <- function(input,output) { # server object contains instructions needed to build the app

  }

shinyApp(ui = ui, server = server) # combines ui and server objects to create the Shiny app
```

Using this fluidPage layout, the app uses all available browser width regardless of the viewer's screen size, resolution, or type, so you, the app developers, don't need to worry about defining relative widths for individual app components. Also, under the hood, shiny implements layout features available in Bootstrap 2, a popular HTML/CSS framework, but the nice thing about working with shiny is that no HTML, CSS, or even Bootstrap experience is necessary to make an app.

Host your Shiny Apps online

Shiny lets you share your apps with the world , by allowing you to host it on a server. You can have your private shiny server or deploy your app on shiny's open server called shinyapps.io. There are other independent hosting platforms like **AWS EC2 Instance** which can be used.

Shinyapps.io is integrated into RStudio so that you can publish your apps with the click of a button, and it has a free version. The free version allows a certain number of apps per user and a certain number of activities on each app, but it should be good enough for just starting out with Shiny. It also lets you see some basic stats about the usage of your app.It is the easy and recommended way of getting the app online. In this project, we have used this method to host our app.

The other option for hosting is on private Shiny server. Instead of RStudio hosting the app for you, you'll have it on your private server. This means a lot more freedom and flexibility, but it also means the need to have a server and be comfortable administering a server.

Creating a Shiny App from Scratch

First of all, we need to install the Shiny Package into our system.

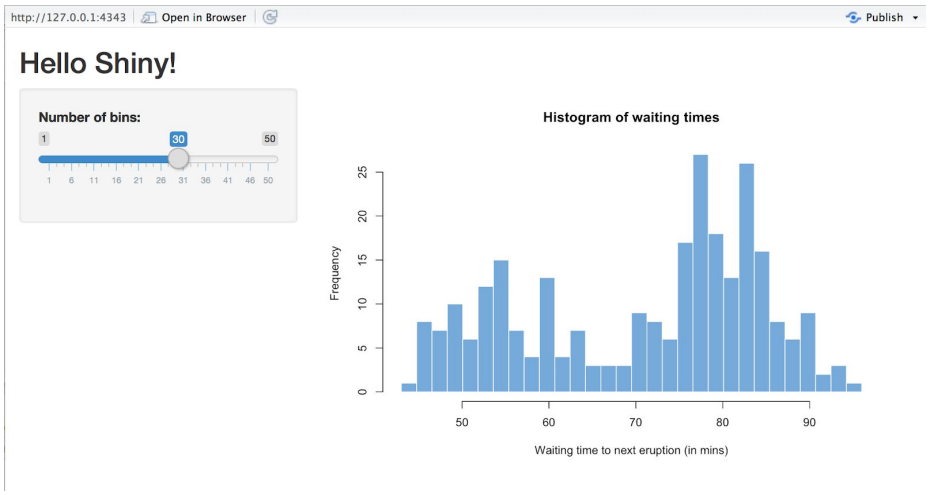
```
install.packages("shiny")
```

The Shiny package has eleven built in examples that each demonstrate how Shiny works. Each example is a self-contained Shiny app. The **Hello Shiny** example plots a histogram of R's faithful dataset with a configurable number of bins. Users can change the number of bins with a slider bar, and the app will immediately respond to their input.

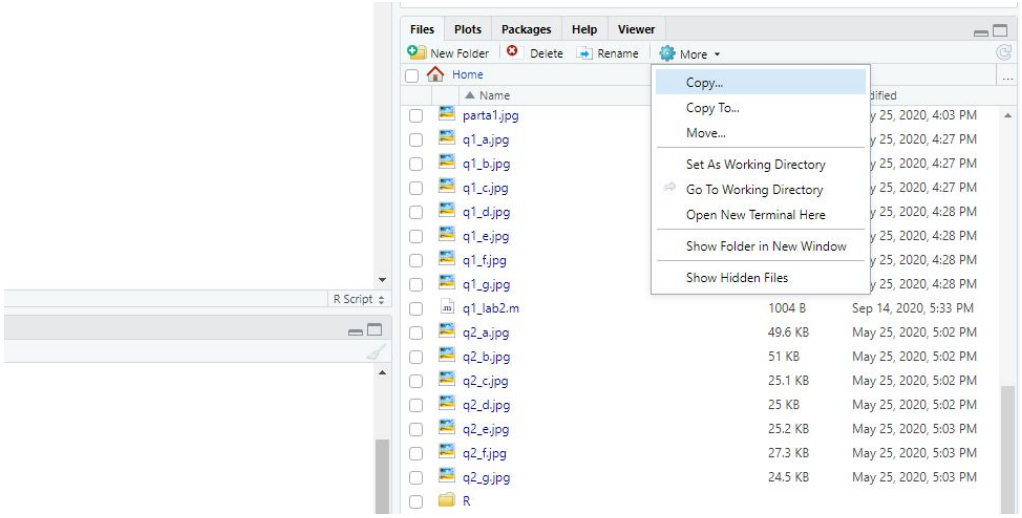
Command to run the Hello Shiny example program -

```
library(shiny)
runExample("01_hello")
```

The output is a Shiny App which opens in a new window -



Shiny apps are contained in a single script called app.R. Let's say this file app.R is in the directory /newdir, then the app can be run by command runApp("newdir"), provided the parent directory of newdir is set as the working directory.



This section appears on the right hand corner of RStudio.

Structure of Shiny App

app.R has three components:

- a user interface object
- a server function
- a call to the shinyApp function

The user interface (ui) object controls the layout and appearance of your app. The server function contains the instructions that your computer needs to build your app. Finally the shinyApp function creates Shiny app objects from an explicit UI/server pair.

By default, Shiny apps display in "normal" mode, like the app pictured above. **Hello Shiny** and the other built in examples display in "showcase mode", a different mode that displays the app.R script alongside the app.

If you would like your app to display in showcase mode, you can run runApp("App-1", display.mode = "showcase").

```
library(shiny)

# Define UI ----
ui <- fluidPage(

)

# Define server logic ----
server <- function(input, output) {

}

# Run the app ----
shinyApp(ui = ui, server = server)
```

This is the basic structure of the App where ui, server and ShinyApp parts are very distinctly separated.

THE UI PART -

Shiny uses the function **fluidPage** to create a display that automatically adjusts to the dimensions of your user's browser window. You lay out the user interface of your app by placing elements in the fluidPage function. **titlePanel** and **sidebarLayout** are the two most popular elements to add to fluidPage. They create a basic Shiny app with a sidebar.

```
ui <- fluidPage(
  titlePanel("title panel"),

  sidebarLayout(position = "right",
                sidebarPanel("sidebar panel"),
                mainPanel("main panel")
  )
)
```

We preferably use **fluidRow** and **column** to build your layout up from a grid system.

All HTML tags can be used in the UI part for text formatting, hyperlinking,etc. They are present in shiny in terms of tag functions. These functions parallel common HTML5 tags.

To insert images in your app, Shiny looks for the img function. To insert an image, give the img function the name of your image file as the src argument (e.g., img(src = "my_image.png")). You must spell out this argument since img passes your input to an HTML tag, and src is what the tag expects. The img function looks for your image file in a specific place. Your file *must* be in a folder named **www** in the same directory as the app.R script. Shiny treats this directory in a special way.

Shiny will share any file placed here with your user's web browser, which makes www a great place to put images, style sheets, and other things the browser will need to build the web components of your Shiny app.

```
ui <- fluidPage(
  titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      img(src = "rstudio.png", height = 140, width = 400)
    )
  )
)
```

We can also add widgets to Shiny apps for improved access and control. Since, shiny is a reactive app, when a user changes the widget, the value will change as well.

Shiny comes with a family of pre-built widgets, each created with a transparently named R function. For example, Shiny provides a function named **actionButton** that creates an Action Button and a function named **sliderInput** that creates a slider bar. We have used some of these widgets in our App. These widgets are made using the Twitter Bootstrap

project. Each widget function requires several arguments. The first two arguments for each widget are 1.) name of the widget and 2.) its label.

For example, for an action button widget , this can be the code - actionButton("action", label = "Action") .

REACTIVE OUTPUT OF SHINY

We can create reactive output with a two step process.

1. Add an R object to your user interface .Shiny provides a family of functions that turn R objects into output for your user interface, e.g. **dataTableOutput**, **tableOutput** ,etc.
2. Tell Shiny how to build the object in the server function. The object will be reactive if the code that builds it calls a widget value. We do this by providing the R code that builds the object in the server function. The server function plays a special role in the Shiny process; it builds a list-like object named output that contains all of the code needed to update the R objects in your app. Each R object needs to have its own entry in the list. In the server function below, output$selected_var matches textOutput("selected_var") in the UI -

```
server <- function(input, output) {

  output$selected_var <- renderText({
    "You have selected this"
  })

}
```

LOADING DATA, R SCRIPTS AND PACKAGES TO THE APP

First, we need to download the data, script or package to the system. If it is data , we can make a folder named *data* and put the data file in there.

Packages need to be installed by running codes within the RStudio console. Some R Scripts use certain packages, which need to be installed to use the downloaded script, for example, we have **helpers.R** ,uses the **maps** and **mapproj** packages in R. We install these packages by running -

```
install.packages(c("maps", "mapproj"))
```

In our App, we have used several packages like **ggplot2, forecast, shinydashboard, zoo** etc.

```
library(shiny)
library(googlevis)
library(quantmod)
library(ggplot2)
library(dplyr)
library(forecast)
library(TTR)
library(data.table)
library(zoo)
library(shinydashboard)
```

GGPLOT2

ggplot2 is a plotting package that makes it simple to create complex plots from data in a data frame. It provides a more programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties.

The easiest way to get ggplot2 is to install the whole tidyverse-

```
install.packages("tidyverse")
```

To start working with **ggplot2** , supply a dataset and aesthetic mapping (with aes()). You then add on layers (like **geom_point()** or **geom_histogram()**), scales (like **scale_colour_brewer()**), faceting specifications (like f**acet_wrap()**) and coordinate systems (like **coord_flip()**).

```
library(ggplot2)

ggplot(mpg, aes(displ, hwy, colour = class)) +
  geom_point()
```

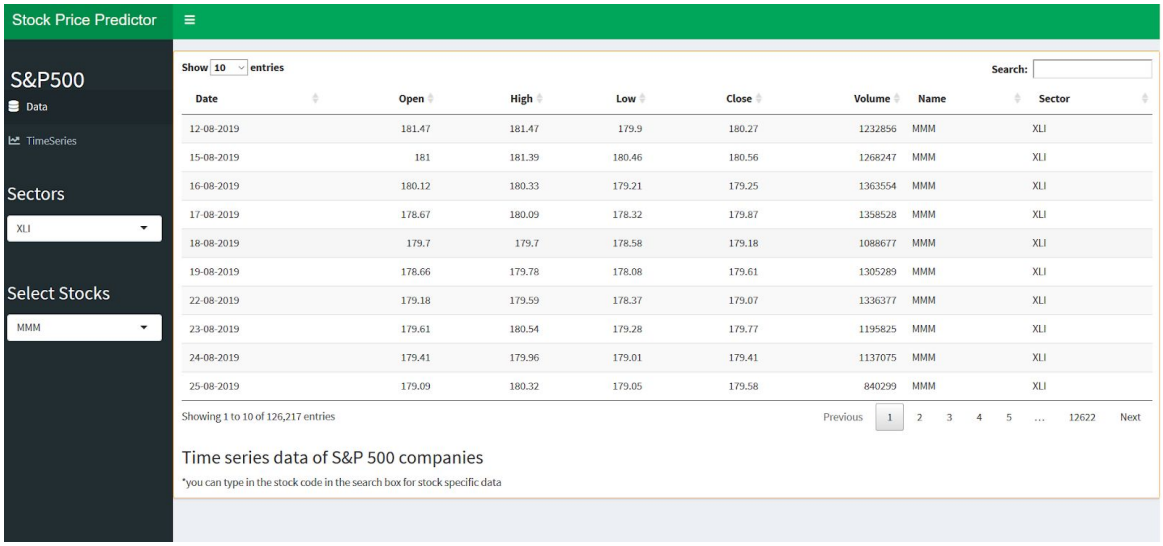
FORECAST

Forecast package contains methods and tools for displaying and analysing univariate time series forecasts including exponential smoothing via state space models and automatic *ARIMA* modelling.

First, we fit a model to the data, and then use the forecast() function to produce forecasts from the model. The forecast() function works with many different types of inputs. It generally takes a time series or time series model as its main argument, and produces forecasts appropriately. It always returns objects of class forecast.

SHINYDASHBOARD

shinydashboard makes it easy to use Shiny to create dashboards.



A dashboard has three parts: a header, a sidebar, and a body -

```
library(shiny)
  library(shinydashboard)
  ui <- dashboardPage(
    dashboardHeader(),
    dashboardSidebar(),
    dashboardBody()
  )
  server <- function(input, output) { }
  shinyApp(ui, server)
```

It has several components like Header, sidebar, search form, slider input, etc.

ZOO

It is particularly aimed at irregular time series of numeric vectors/matrices and factors. Zoo's key design goals are the independence of a particular index/date/time class and consistency with ts and base R by providing methods to extend standard generics. Here, it is used to smooth the plots whenever we do *moving average*.

Time series data

A time series is a set of observations on the values that a variable takes a different times.

Search data may be collected at regular time intervals search as monthly, weekly, quarterly, or annually.

Time series are used in statistics, econometrics, mathematical Finance, weather forecasting, earthquake prediction, and many other applications.

- Univariate time series:
  A "univariate time series" refers to a time series that consists of single observations recorded over regular time intervals.

Examples: Monthly returns data of a stock

- Cross-sectional Data:
  Such a type of data is collected by observing many subjects (such as individuals, firms, countries, or regions) at the same point of time or during the same time.

  Example: Suppose an analyst wants to know the number of cars a household has brought in the past year. To do so, he collects data on a sample of, say, 500 families from the population and notes the data on how many cars they have brought in the past year. This "cross-sectional" sample provides a glimpse of the population for the duration.

Patterns emerging in time-series data

Depending on the frequency of the data (hourly, daily, weekly, monthly, quarterly, annually, etc.), different patterns emerge in the data set, which forms the component to be modeled.

Sometimes the time series may just increase or decrease over time with a constant slope, or there may be patterns around the increasing slopes.

Components of a time series

The pattern in a time series is sometimes classified into trend, seasonal, cyclical, and random components.

- Trend: A long-term relative that usually persists for more than one year.

- Seasonal: A pattern that appears in regular intervals wherein the frequency of occurrence is within a year or even shorter. For example. Quarterly GDP series for India.

- Cyclical: Repeated pattern that appears in a time series but beyond the frequency of one year. It is a wavelike pattern about the long-term Trend over a number of years. Cycles are rarely regular and appear in combination with other components. Example: Business cycles that record periods of economic recession and inflation, cycles in the monetary and financial sectors.

- Random: The component of the time series that is obtained after these three patterns have been 'extracted' out of the series is the random component. Therefore, when we plot the Residual series, then the scatter plot should be devoid of any pattern and would be indicating only a random pattern around the mean value.

IDEA BEHIND UNIVARIATE TIME SERIES MODELLING

The need to use univariate modeling arises in situations where:

(a) An appropriate economic theory to the relationship between series may not be available and hence one considers only the statistical relationship of the given series with its past values.

(b) Sometimes even when the set of explanatory variables may be known it may not be possible to obtain the entire set of such variables required to estimate a regression model and one would then have to use only a single series of the dependent variable to forecast the future values.

APPLICATIONS OF UNIVARIATE TIME SERIES IN TERMS OF FORECASTING

A. Forecasting inflation rate or unemployment rates or the net inflow of foreign funds in the near future could be of interest to the government.
B. Firms may be interested in demand for their product (e.g. two-wheelers, soft drinks bottles, or soaps etc.) or the market share of their product.
C. Housing finance companies may want to forecast both the mortgage interest rate and the demand for housing loans.
D. Forecasting gold or silver prices by the jewel merchant.

Different Time Series Processes

1. White Noise:

A series is called white noise if it is purely random in nature. Let $\{\varepsilon_t\}$ denote such a series then it has zero mean ($E(\varepsilon_t)=0$], has a constant variance [$V(\varepsilon_t)=\sigma^2$] and is an uncorrelated [$E(\varepsilon_t\varepsilon_s)=0$) random variable.

The scatter plot of such a series across time will indicate no pattern and hence forecasting the future values of such a series is not possible.

2. Auto Regressive Model:

An AR model is one in which Y_t , depends only on its own past values Y_{t-1}, Y_{t-2}, Y_{t-3}, etc

Thus,

$$Y_t = f(Y_{t-1}, Y_{t-2}, Y_{t-3}, \ldots, \varepsilon_t)$$

A common representation of an autoregressive model where it depends on p of its past values called as AR(p) model is represented below:

$$Y_t = \beta_0 + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \beta_3 Y_{t-3} + .. + \beta_p Y_{t-p} + \varepsilon_t$$

3. Moving Average Model:

A moving average model is one when Y_t depends only on the random error terms which follow a white noise process

i.e.

$$Y_t = f(\varepsilon_t, \varepsilon_{t-1}, \varepsilon_{t-2}, \varepsilon_{t-3} \ldots.)$$

A common representation of a moving average model where it depends on q of its past values is called MA(q) model and is represented below:

$$Y = \beta_0 + \varepsilon_t + \varphi_1 \varepsilon_{t-1} + \varphi_2 \varepsilon_{t-2} + \varphi_3 \varepsilon_{t-3} + \ldots + \varphi_q \varepsilon_{t-q}$$

The error terms ε_t, are assumed to be white noise processes with mean zero and variance σ^2.

4. AutoRegressive Moving Average model:

There are situations where the time-series may be represented as a mix of both AR and MA models referred to as ARMA (p,q).

The general form of such a time-series model, which depends on p of its own past values and q past values of white noise disturbance, takes the form:

$$Y = \beta_0 + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \beta_3 Y_{t-3} + \ldots.. + \beta_p Y_{t-p} + \varepsilon_t + \varphi_1 \varepsilon_{t-1} + \varphi_2 \varepsilon_{t-2} + \varphi_3 \varepsilon_{t-3} + \ldots. + \varphi_q \varepsilon_{t-q}.$$

5. AutoRegressive Integrated Moving Average model:

The ARIMA model combines three basic methods:

- AutoRegression (AR) – In auto-regression the values of a given time series data are regressed on their own lagged values, which is indicated by the “p” value in the model.
- Differencing (I-for Integrated) – This involves differencing the time series data to remove the trend and convert a non-stationary time series to a stationary one. This is indicated by the “d” value in the model. If d = 1, it looks at the difference between

two time series entries, if d = 2 it looks at the differences of the differences obtained at d =1, and so forth.

- Moving Average (MA) – The moving average nature of the model is represented by the “q” value which is the number of lagged values of the error term.

Stationarity of a Time Series

A series is said to be "strictly stationary" if the marginal distribution of Y at time *t*[$p(Y_t)$] is the same as at any other point in time.
Therefore,

$p(Y_t) = p(Y_{t+k})$ and $p(Y_t, Y_{t+k})$ does not depend on t. (Here, $t \geq 1$ and k is any integer)

This implies that the mean, variance and covariance of the series Y_t are time invariant.

However, a series is said to be “weakly stationary” or covariance stationary” if the following conditions are met:

a) $E(Y_1) = E(Y_2) = E(Y_3) = \ldots\ldots = E(Y_t) = \mu$ (a constant)
b) $Var(Y_1) = Var(Y_2) = Var(Y_3) = \ldots\ldots = Var(Y_t) = \gamma_0$ (a constant)
c) $Cov(Y_1, Y_{1+k}) = Cov(Y_2, Y_{2+k}) = Cov(Y_3, Y_{3+k}) = \gamma_k$, depends on lag k.

A series which is non - stationary can be made stationary after differencing. A series which is stationary after being differentiated once is said to be **integrated by order 1 and is denoted by I(1).**
In general a series which is being differentiated d times is said to be integrated of order d, denoted by I(d).
Therefore, a series, which is stationary without differencing, is said to be I(0).

Necessity of assumption of stationarity

Why the Stationarity assumption?

(i) The results of classical econometric theory is derived under the assumption that variables of concern are stationary.
(ii) Standard techniques are largely invalid where data is non-stationary.
(iii) Sometimes, autocorrelation may result because the time series is non-stationary.
(iv) Non-stationary time series regressions may also result in spurious regressions, i.e cases when the regression equation shows significant relationship between two variables when actually there should not be any such relation.

The estimation and forecasting of univariate time-series models is carried out using the Box-Jenkins (B-J) methodology which has following three steps:

(1) Identification
(2) Estimation
(3) Diagnostic Checking

The B-J methodology is applicable only to stationary variables.

1. Identification

a) Autocorrelation function(ACF):
Autocorrelation refers to way the observations in a time series are related to each other an is measured by a simple correlation between current observation (Y_t) and the observation p periods from the current one (Y_{t-p}).

$$\rho_k = \text{Corr}(Y_t, Y_{t-p}) = \frac{Cov(Y_t, Y_{t-p})}{\sqrt{var(Y_t)}\sqrt{var(Y_{t-p})}} = \frac{\gamma_p}{\gamma_0}.$$

b) Partial Autocorrelation function(PACF):
Partial Autocorrelations are used to measure the degree of association between Y_t and Y_{t-p} when the effects at other time lags 1,2,3,..., (p-1) are removed.

c) Inference from ACF and PACF:
The theoretical ACFs and PACFs are available for various values of the lags of autoregressive and moving average components i.e p and q.
Therefore, a comparison of the correlograms (plot of sample ACFs versus lags) of the time series data with the theoretical ACFs and PACFfs leads to the selection of the appropriate ARIMA (p,q) model.

The general characteristics of the theoretical ACFs and PACFs are as follows:

MODEL	ACF	PACF
AR(p)	Spikes decay towards zero	Spikes cutoff to zero
MA(q)	Spikes cutoff to zero	Spikes decay towards zero
ARMA(p,q)	Spikes decay towards zero	Spikes decay towards zero

- The first step in modeling process is to check for the stationary of the time series as the estimation procedures are available only for the stationary series. A non-stationary process must be transformed and changed to a suitable stationary form.
- The next step is to find the initial values for the orders (p and q) of the AR and MA process. This is done by looking at the significant autocorrelation from the previous table.
- However, this is not a hard and fast rule and is only and indicative process. Therefore, the next two steps of the BJ methodology: Estimation and Diagnostic Checking should be performed cautiously.

2. Estimation:

Several methods are available for estimating the parameters of an ARMA model depending on the assumptions one makes on the error terms. They are

(a) Yule Walker procedure

(b) Maximum Likelihood method, etc....

Since no closed form estimates can be found analytically, one has to seek computer aided methods and most statistical and econometric packages facilitate the process of estimation.

3. Diagnostic Checking:

Different methods can be obtained for various combinations of AR and MA individually and collectively. The best model is obtained by following the diagnostic testing procedure.

The tests are given by:

a) Lowest value of AIC / BIC / SBIC - The model with the lowest value of the above criterion chosen as the best model.
b) Plot of the residual ACF :
On fitting the appropriate ARIMA model, the goodness of fit can be estimated by plotting the ACF of residuals of the fitted model. If most of the sample autocorrelation coefficients of the residuals lie within the limits $\left(\frac{-1.96}{\sqrt{N}}, \frac{+1.96}{\sqrt{N}}\right)$, where N is the number of observations, then the residuals are white noise indicating that the model fit is appropriate.

About our work:

This project focuses on using univariate time series forecasting methods for the stock market index, S&P 500 emphasizing on Box-Jenkins AutoRegressive Integrated Moving Average (ARIMA) modeling. The time series analysis was done using R studio to both predict and visualize our predictions. Along with the interactivity of plotly through the ggplot2 package we were able to create stunning visuals that help in understanding which value of p,q and d is most appropriate for our own time series analysis.

There are 2 files: App.R and global.R. App.R contains the main code of our app.

Key Libraries Used:

There are a number of packages available for time series analysis and forecasting.

Mentioned earlier in “LOADING DATA, R SCRIPTS AND PACKAGES TO THE APP” section. Key libraries are:

ggplot2, forecast, shinydashboard, zoo, &

DT : provides an R interface to the JavaScript library DataTables,

googleVis: provides an interface between R and Google's charts tools allowing users to create web pages with interactive charts based on R data frames. Charts are displayed locally via the R HTTP help server,

dplyr: for data manipulation,

TTR: A collection of over 50 technical indicators for creating technical trading rules,

quantmod: assist the quantitative trader in the development, testing, and deployment of statistically based trading models, and many other packages.

Getting Data, Cleaning Data & Loading Data:

Implemented (coded) mainly in global.R.

We collect our data. We want to use reliable sources of complete and accurate data. We collected *2 years* (2019-2020) of S&P 500 Stock Index data at a daily frequency (a total of 252 observations for each stock option) from Yahoo Finance.

global.R contains codes for importing these data from CSV files using read.csv() function so that our data set is included within our working R environment. We then process and manipulate it. We made sure that all fields contained data (i.e. no missing values) and that the headers (i.e. column names) were labeled appropriately.

Overview of our App:

In our apps interface there are two options:

1. To view raw data of the options.
2. Time Series analysis (containing various plots like plot of data and forecasting and also a table for accuracy of our forecasting plot).

Our app also provides options to select from a number of stocks belonging to different sectors.

Details about the code in app.R:

UI PART

We first create the dashboard for our app using the shinydashboard package. In the dashboard sidebar menu, we have two menu items **Data** and **TimeSeries**. We also have the option of selecting sectors of S&P 500 using a drop down menu and then the respective

stocks through a separate drop down menu. Following is the code snippet for the sidebar of our App's dashboard -

```
dashboardPage (
  skin = "green",
  dashboardHeader(title = 'Stock Price Predictor', titleWidth = 230),
  dashboardSidebar(
    sidebarUserPanel(h2("S&P500")
                     ),
    sidebarMenu(
      menuItem('Data', tabName = 'dt', icon = icon('database')),
      menuItem('TimeSeries', tabName = 'ts', icon = icon('line-chart'))
    ),

    selectizeInput('sector', h3('Sectors'),
                   choices = na.omit(unique(stocks_w_sec$Sector)),
                   selected = 'XLI'),
    selectizeInput('price1', h3('Select Stocks'),
                   choices = unique(stocks_w_spy$Name),
                   selected = 'MMM')
```

Next, we design the body of our dashboard. We have one body for the **Data** tab and another for **TimeSeries** tab. **Data** tab gives us a table of stocks and the **TimeSeries** has 3 boxes named Price,Correlation and Forecast. Following is the code snippet for the body of our App's dashboard -

```
tabItem(tabName = 'ts',
        fluidRow(
          box(title = 'ARIMA(p,d,q)', width = 3,
              solidHeader = T, status = 'warning',
              sliderInput('ar', 'Autoregression',
                          min = 0, max = 3, value = 0),
              sliderInput('diff', 'Differencing',
                          min = 0 , max = 3, value = 0),
              sliderInput('ma', 'Moving-average',
                          min = 0 , max = 3, value = 0)),
          tabBox(title = tagList(shiny::icon("line-chart"), "Time Series"), width = 9,
                 tabPanel('Price',
                          plotOutput('fit_stock')),
                 tabPanel('Correlation',
                          fluidRow(
                            box(h4(tags$b('ACF')), solidHeader = T, status = 'success',
                                'Auto correlation between time series and its own lag',
                                plotOutput('acf')),
                            box(h4(tags$b('PACF')), solidHeader = T, status = 'success',
                                'Conditional auto correlation between time seires and its own lag',
                                plotOutput('pacf'))
                          )),

                 tabPanel('Forecast',
                          fluidRow(
                            box('Forecast',
                                plotOutput('fore')),
                            box('Plot fitted data with price',
                                plotOutput('f_v_d')),
                            box('Accuracy',
                                tableOutput('accu'))
                          )))
        )
))
```

We have 3 slider inputs here for changing the values of p(Autoregression), d(Differencing) and q(Moving Average). We keep the values between 0 and 3 and the default value = 0. In the correlation tab, we show two types of auto correlation - ACF and PACF.

Server Part-

```
# Define server logic ----
server <- function(input, output, session) {
  output$table = DT::renderDataTable({
    datatable(stocks_w_sec, rownames = F)
  },striped = TRUE, bordered = TRUE)


  ## data fitting
  fit_data = reactive({
    stocks_w_spy %>%
      filter(., Name == input$price1) %>%
      select(., Date, Close) %>%
      na.omit(.) %>%
      column_to_rownames(.)
  })
  #model fitting
  fit_model = reactive({
    stocks_w_spy %>%
      filter(., Name == input$price1) %>%
      select(., Date, Close) %>%
      slice(., 1:200) %>%
      na.omit(.) %>%
      column_to_rownames(.) %>%
      as.ts(.) %>%
      Arima(., order = c(input$ar, input$diff, input$ma))
  })
```

output$table is used to render data from data frame stocks_w_sec. Function "renderDataTable" makes a reactive version of the given function that returns a data frame (or matrix), which will be rendered with the DataTables library. Paging, searching, filtering, and sorting can be done on the R side using Shiny as the server infrastructure.

In fit_model we have used ARIMA function which takes inputs p (variable name 'ar'),d (variable name 'diff') and q (variable name 'ma') and applies ARIMA on the timeseries data. The arima() method, found within the forecast package, yields the best model for a time-series based on Akaike-Information-Criterion (*AIC*). The *AIC* is a measurement of quality used across various models to find the best fit. After running our original and differenced data sets through the arima() method we confirmed that the *ARIMA(1, 1, 1)* is our best fit model, but we can also change the values of p,d and q to see how changing these values affect our forecast result (prediction result).

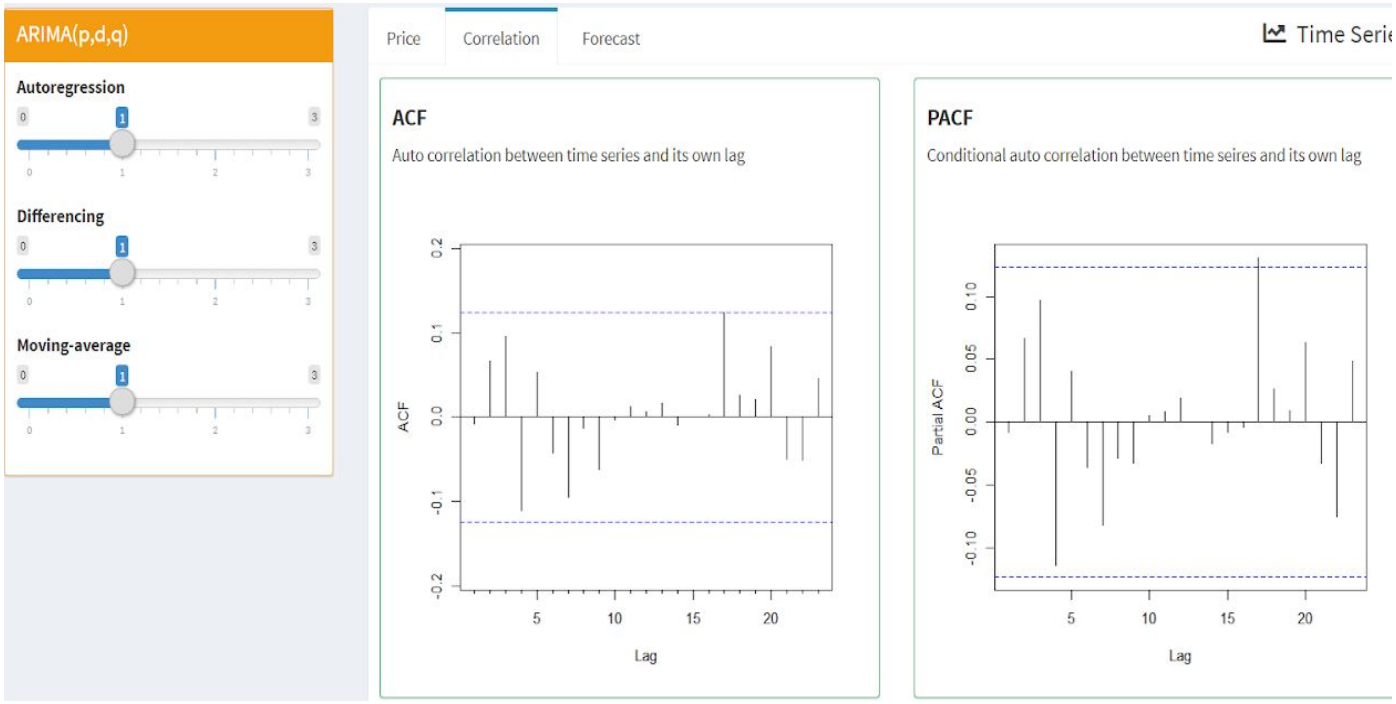
```
#### price of stocks
output$fit_stock = renderPlot({
  if (input$diff == 0) {
    ggplot(data = fit_data(), aes(x = 1:nrow(fit_data()), y = Close)) +
      geom_line() + xlab('Time Period') +
      ggtitle(input$price1) + theme_bw()
  } else {
    ggplot() + geom_line(aes(x = 1:length(diff(ts(fit_data()), differences = input$diff)),
                             y = diff(ts(fit_data()), differences = input$diff))) +
      xlab('Time Period') + ylab('After Differencing') +
      ggtitle(input$price1) + theme_bw()
  }
})

output$acf = renderPlot({
  if (input$diff == 0) {
    Acf(fit_data()[1:200,], main = '')}
  else {
    Acf(diff(ts(fit_data()), differences = input$diff), main = '')
  }
})

output$pacf = renderPlot({
  if (input$diff == 0) {
    pacf(fit_data()[1:200,], main = '')}
  else {
    pacf(diff(ts(fit_data()), differences = input$diff), main = '')
  }
})
```

In above we are returning three plots namely,

a. fit_stock: This is the returning plot of our time series data without forecasting. This helps in visualizing our data. This enables us to make inferences about important components of the time-series data, such as *trend*, *seasonality*, *heteroskedasticity*, and *stationarity.*
b. ACF plot & PACF plot: These plots help us confirm that we have stationarity and also helps us deduce which value of p and q will be the best to use in the ARIMA model. From the plots we obtain after running our app (shown below), we deduce that an *MA(1)* model (where *MA* stands for moving average) best fits our data because the *ACF* cuts off at one significant lag and the *PACF* shows geometric decay. Recall that we are examining the differenced time-series so we have to use the combined model *ARIMA* (Autoregressive integrated moving average), thus our model so far is *ARIMA(1, 1, 1)*. Here, we have used Acf() and pacf() functions. The function acf() computes (and by default plots) estimates of the autocovariance or autocorrelation function. Similarly, function pacf() is the function used for the partial autocorrelations.




```
143  #### following is the forecast graph code -
144    output$fore = renderPlot({
145      plot(forecast(fit_model(), h = 50))
146      lines(fit_data()[1:250,])
147      abline(h = forecast(fit_model(), h = 50)[[6]][50,1], lty = 'dashed', col = 'red') +
148        text(x = 200,
149             y = forecast(fit_model(), h = 50)[[6]][50,1] - 1,
150             labels = round(forecast(fit_model(), h = 50)[[6]][50,1], 2), cex = 0.8) +
151        text(x = 175,
152             y = forecast(fit_model(), h = 50)[[6]][50,1] - 1,
153             labels = '80%:', cex = 0.8)
154      abline(h = forecast(fit_model(), h = 50)[[6]][50,2], lty = 'dotdash', col = 'blue') +
155        text(x = 200,
156             y = forecast(fit_model(), h = 50)[[6]][50,2] - 1,
157             labels = round(forecast(fit_model(), h = 50)[[6]][50,2], 2), cex = 0.8) +
158        text(x = 175,
```

Above code returns the plot with our calculaed

We call the ARIMA function on the training dataset for which the order specified is (1, 1, 1). We use this fitted model to forecast the next data point by using the forecast Arima function.The function is set at 80% and 95% confidence level. One can use the confidence level argument to enhance the model. We will be using the forecasted point estimate from

the model. The “h” argument in the forecast function indicates the number of values that we want to forecast, in this case, the next day returns.

We can see that the model performs well and within the 80% and 95% confidence intervals. You can forecast values even further into the future by tuning the appropriate parameters.

```
  output$accu = renderTable({
    accuracy(forecast(fit_model(), h = 50), fit_data()[201:250,])
  })
  ## forecast output
  output$f_v_d = renderPlot({
    ggplot() +
      geom_line(aes(x = 1:200, y = fit_data()[1:200,])) +
      geom_line(aes(x = 1:length(fitted(fit_model())),
                    y = fitted(fit_model())), col = 'red',
                inherit.aes = F) +
      ggtitle(input$price1) + xlab('Time Period') + ylab('Stock Price') +
      theme_bw()
  })
}
```

In above, we are using an accuracy function to return the accuracy of our forecast which we are passing in the UI where we are printing the accuracy table.

The second part is for plotting fitted data with price i.e. our forecast predictions that we obtained after using the ARIMA model along with the price to see the comparison.