FOSSEE Semester Long Internship Report

(Part-time)

on

FLOSS – R

submitted by

Ayush Kumar Nayak (National Institute of Technology, Rourkela)

under the guidance of

Prof. Radhendushka Srivastava

Mathematics Department

IIT Bombay

Prof. Kannan M. Moudgalya

Chemical Engineering Department,

IIT Bombay

and supervision of

Debatosh Chakraborty

Project Research Assistant,

R Team, FOSSEE,

IIT Bombay

July 18, 2024

Acknowledgement

I thank Prof. Radhendushka Srivastava, Department of Mathematics, IIT Bombay, for his guidance and suggestions throughout the project. I also express my sincere gratitude to Prof. Kannan M. Moudgalya, Department of Chemical Engineering, IIT Bombay, for creating the FOSSEE Semester Long Internship program and providing students from all over India an opportunity to participate in it. I equally respect the guidance provided by Mr. Debatosh Chakraborty throughout the project.

Contents

Chapter 1

Introduction

In this report, I shall present my contributions made to the Free and Open Source Software (FLOSS) community during the Semester Long Internship 2024. I worked online, in part time mode starting from 1st March 2024 to 30th June 2024. Contributions were made using ‘R’, a FLOSS language and environment for statistical computing and graphics. This internship was organised by the FOSSEE Project based at IIT Bombay. The thrust area of the project is promoting and creating open-source software equivalent to proprietary software. This project is a part of the National Mission on Education through ICT funded by the Ministry of Education, GoI. My contributions included the creation of an algorithm to digitize line plots, & implementation of the algorithm in R and the creation of an R TBC.

Chapter 2

Contribution to the TBC Project

As a part of the selection procedure for the FOSSEE Semester Long Internship, an applicant is required to select a standard textbook related to Probability/Statistics etc., with at least 80 solved examples to submit a TBC proposal for the R TBC project. I was asked to complete a book on time series and code not only the examples but also code the plots for the datasets given in the book. My proposal got approved and during the fellowship period, I contributed to the R TBC project by creating an R textbook companion for the below-mentioned textbook:

Textbook name	**Author**	**Edition**
Introduction to time series and forecasting	Peter J. Brockwell Richard A. Davis	3rd

My submitted TBC shall be available for public use on the R TBC Completed Books webpage upon approval.

Chapter 3

Plotting time series data

About the book:

This book, “Introduction to time series and forecasting” is aimed at the reader who wishes to gain a working knowledge of time series and forecasting methods. It contains a total of 11 chapters and 5 appendices. Starting with the introduction to time series, the book delves deep into time series data and its analysis. Most examples in the book include graphs and several calculations to understand the behaviour of time series data. Based on the datasets given in the book, I have recreated all the plots which has helped me to understand the nuances of time series data and its analysis.

Dataset link:

http://extras.springer.com

Example:

An example from Page 84 was recreated, the code and output of which are shown below.

```
oshorts<- read.csv("OSHORTS.TSM", header =FALSE)
colnames(oshorts)[1] <- "overshorts"
oshorts$days <- seq(1,nrow(oshorts))
# Figure 3-5
plot(oshorts$days,oshorts$overshorts, xlab = "Days", ylab = "Overshorts",
     type = 'o', col = "blue")

abline(h=0)

# Figure 3-6
acf_result <- acf(oshorts$overshorts, plot = FALSE)
n <- length(oshorts)
bounds <- 1.96 * ((1 + 2 * acf_result$acf[2]^2)^(1/2)) / sqrt(n)
plot(acf_result, main = "Sample ACF with Bounds")

print(mean(oshorts$overshorts))

## [1] -4.035088

acvf<-acf(oshorts$overshorts, plot= FALSE, type = 'covariance')
print(acvf$acf[1])
```

```
## [1] 3415.718

print(acvf$acf[2])

## [1] -1719.956
```

The plots are given below:

Time series of the **overshorts**.

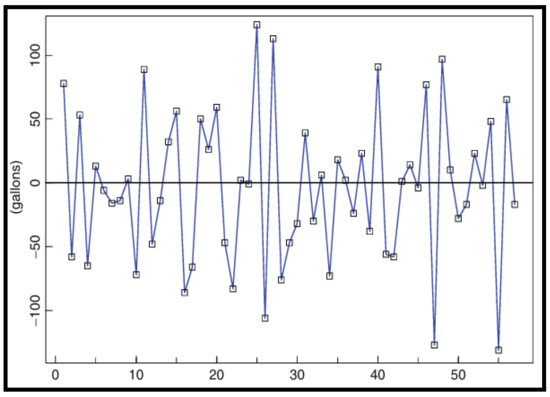


Figure: Timeseries plots of the data

The sample ACF of the overshorts data, showing the given bounds

$\pm 1.96 n^{0.5} - \frac{1}{2}(1 + 2\rho^{2}(1))^{0.5}$ assuming an MA(1) model for the data.

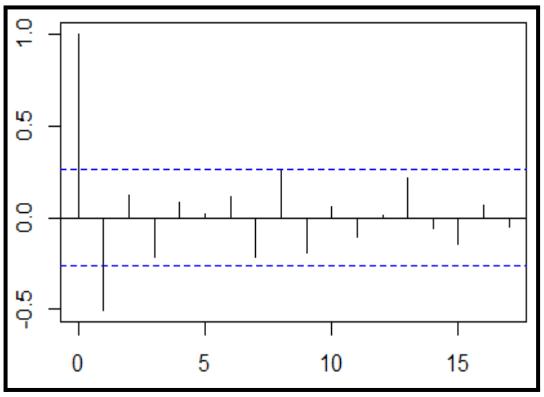


Figure: Recreated ACF plot

Chapter 4

Digitization

1. Introduction

Digitization is the process of converting information into a digital format that can be easily stored and analysed. In the context of plots, digitization involves transforming graphical representations of data, such as charts and graphs into digital data points. This process allows for efficient data analysis to uncover meaningful insights. Digitizing plots is essential because many plots are still only available as static images.

2. Digitization in MATLAB

Digitizing plots has become more accessible thanks to various available tools and software. One of the popular options include MATLAB. MATLAB is renowned for its robust mathematical and engineering capabilities. Extraction of datapoints can be performed using available open-source software files. In the procedure demonstrated below, I have used GRABIT Version 1.0.0.1 by Jiro Doke.

3. About GRABIT

GRABIT starts a GUI program for extracting data from an image file. It is capable of reading in BMP, JPG, TIF, GIF, and PNG files (anything that is readable by IMREAD). Multiple data sets can be extracted from a single image file, and the data is saved as an n-by-2 matrix variable in the workspace. It can also be renamed and saved as a MAT file.

4. Procedure

To demonstrate the implementation, I first generated a line plot in R and saved it as an image.

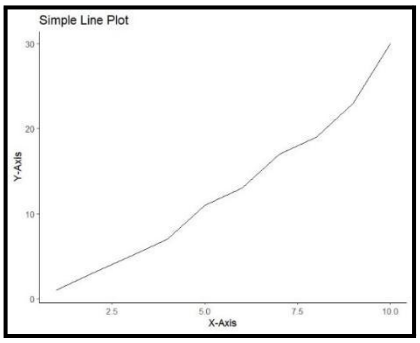


Figure: Line plot stored as an image

Next, Following steps were taken for digitizing the plot:

- The image file was loaded in the interface.

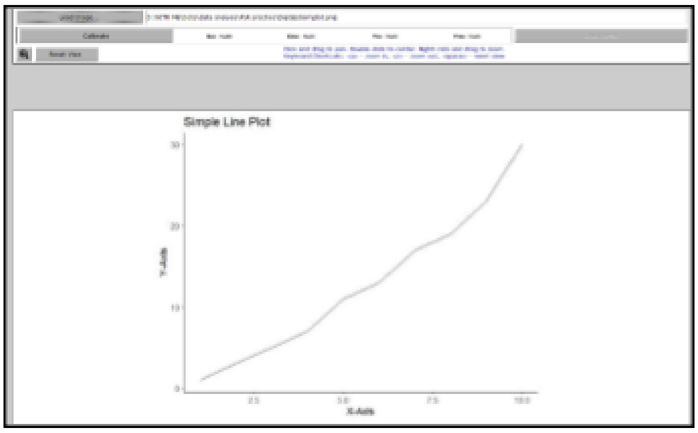
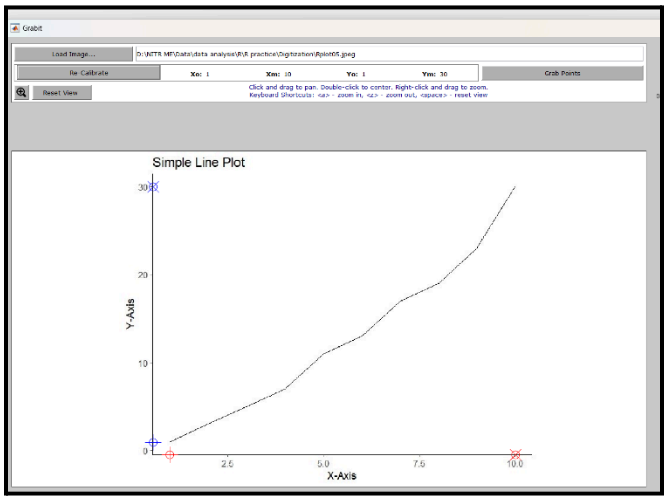


Figure: Loaded image in interface

- Calibrate axes dimensions. User will be prompted to select 4 points on the image.


Grabit

D:\NITR ME\Data\data analysis\R\R practice\Digitization\Rplot05.jpeg

Xo: 1 Xm: 10 Yo: 1 Ym: 30

Click and drag to pan. Double-click to center. Right-click and drag to zoom.
Keyboard Shortcuts: <a> - zoom in, <z> - zoom out, <space> - reset view

Figure: Calibration of axes

- The datapoints were grabbed by clicking on points on the line. Right-click to delete a point. Image can be zoomed during this stage.

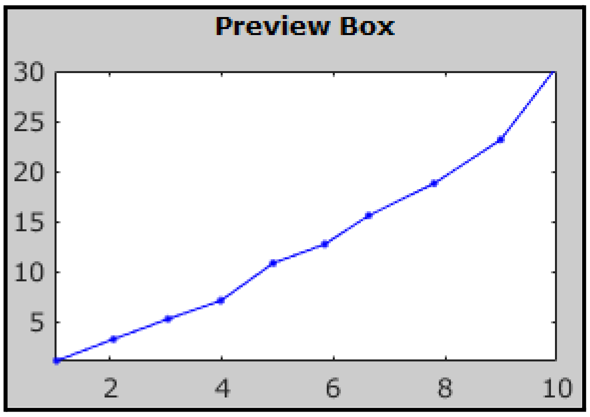
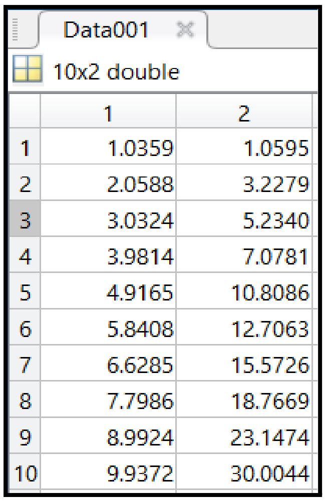


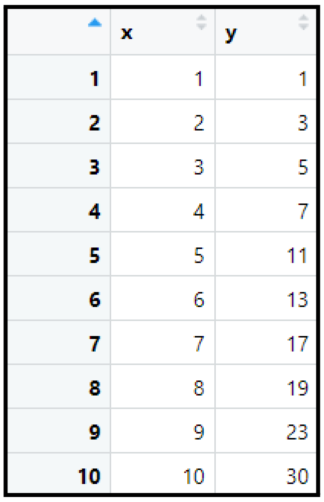
Figure: Preview of recreated plot

- Multiple data sets will remain in memory so long as the GUI is open. Variables can be renamed, saved to file, or edited in Array Editor.
- Shown below is a comparison of obtained and real data.


Data001
10x2 double

	1	2
1	1.0359	1.0595
2	2.0588	3.2279
3	3.0324	5.2340
4	3.9814	7.0781
5	4.9165	10.8086
6	5.8408	12.7063
7	6.6285	15.5726
8	7.7986	18.7669
9	8.9924	23.1474
10	9.9372	30.0044

Figure: Digitized data



	x	y
1	1	1
2	2	3
3	3	5
4	4	7
5	5	11
6	6	13
7	7	17
8	8	19
9	9	23
10	10	30

Figure: Real data

5. Digitization procedure using R

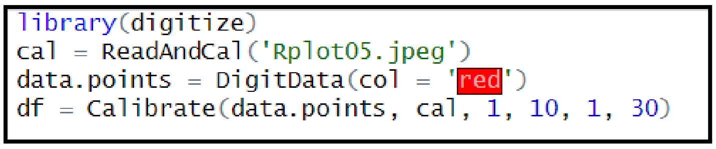
The package **digitize** facilitates digitization in R. It requires user input for calibration as well as mapping of datapoints. This might prove to be a handy tool for retrieving data points from figures in old articles for which the raw data is not available.

- Step 1: A simple plot is generated and saved as an image. For comparison of results, the same line plot is used for digitization in both R and the GRABIT tool in MATLAB.

- Step 2: First the image is read using the **ReadAndCal** function. The lowest and highest x and y values were selected.

- Step 3: To retrieve values for specific data points, simply each points of interest are simple clicked. The function **DigitData** will mark each selected point with the specified colour and save the corresponding raw x and y coordinates to the **data.points** list.

- Step 4: Finally, raw x and y coordinates were converted to the original graph's scale by calling the **Calibrate** function. The **data.points** list, the **cal** list containing the four control points, and four numeric values representing the original points clicked on the x and y axes, in the figure's original scale were specified for the function.



```
library(digitize)
cal = ReadAndCal('Rplot05.jpeg')
data.points = DigitData(col = 'red')
df = Calibrate(data.points, cal, 1, 10, 1, 30)
```

Figure: R code snippet for digitization

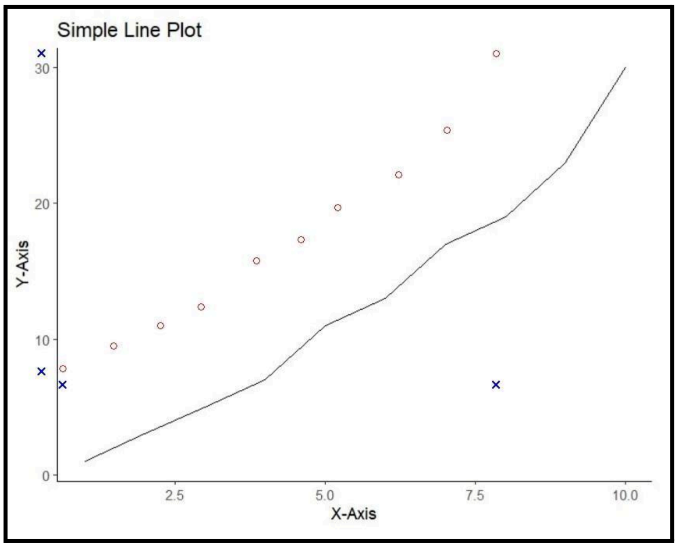
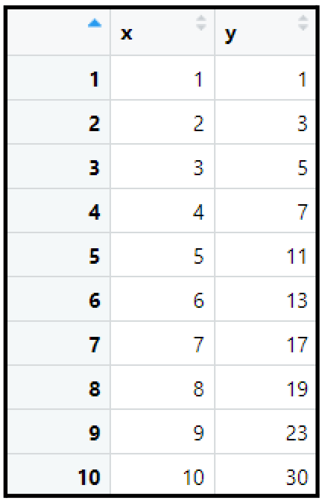


Figure: Calibration of axes and datapoints

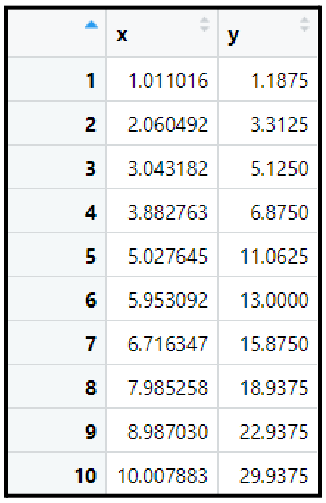
6. Results:

The following images show the comparison of the extracted data with the real data. It is observed that neither the x nor the y values match with the real data.



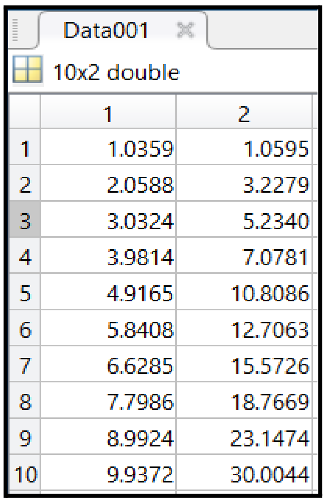
	x	y
1	1	1
2	2	3
3	3	5
4	4	7
5	5	11
6	6	13
7	7	17
8	8	19
9	9	23
10	10	30

Figure: Real data



	x	y
1	1.011016	1.1875
2	2.060492	3.3125
3	3.043182	5.1250
4	3.882763	6.8750
5	5.027645	11.0625
6	5.953092	13.0000
7	6.716347	15.8750
8	7.985258	18.9375
9	8.987030	22.9375
10	10.007883	29.9375

Figure: Digitized data in R



Data001

10x2 double

	1	2
1	1.0359	1.0595
2	2.0588	3.2279
3	3.0324	5.2340
4	3.9814	7.0781
5	4.9165	10.8086
6	5.8408	12.7063
7	6.6285	15.5726
8	7.7986	18.7669
9	8.9924	23.1474
10	9.9372	30.0044

Figure: Digitized data in MATLAB

7. Challenges

- **Errors in axis detection:** Requires manually clicking on the plot to calibrate the axes. Involves selecting control points, which can introduce errors if not accurately clicked.
- **Manual data extraction:** Data points must be clicked on individually for extraction. This process is time-consuming and prone to inaccuracies.
- **Limited tick marks:** If the plot has few tick marks, estimating the exact values becomes challenging. It increases the probability of inaccuracies due to guessing.
- **Lack of reproducibility:** Manually digitized data can vary between different users. Lack of standardization can affect the reproducibility of results.
- **Limited time efficiency:** Large datasets can be particularly cumbersome to handle manually. Also causes visual fatigue to users.

8. Conclusion

Digitization of plots is a crucial process for transforming graphical data into a digital format for enhanced analysis and reproducibility. As shown in the results, manual digitization presents challenges such as human error, time consumption, and limited accuracy. In the next chapter, I will explore automating the data extraction process, addressing the need for efficiency and accuracy in digitizing plots.

Chapter 5

Extraction of data from line plots

1. Introduction

In data analysis and research, charts and graphs are essential for sharing information. However, it's often hard to get the data these visuals are representing. This problem comes up frequently when analyzing government data and stock information which are often, only available as images. Therefore, I was given the task of extracting data from the image of a line plot using R programming language. To address this challenge, it was suggested to analyze plots depicting stock prices versus dates.

My project, Plot Digitization involves identifying the key components of the plot and accurately mapping the pixel data back to the original coordinates. I have used contour detection, optical character recognition and regression for this project. The process involved conducting an exhaustive literature survey, the creation of an algorithm to solve the problem, and implementing it in the R programming language.

2. Problem statement

The project aims to address the challenge of extracting data and recreating a line plot from an image of a stock prices plot for AAPL (Apple Inc.) over the past month. By digitizing the plot from its image, we seek to extract data from the image of a line plot. This solution is based on image processing and regression techniques while ensuring precision in recreating the line plot from its image.

3. Literature survey

I performed an exhaustive literature survey to find any existing solutions for plot digitization. Prior research has explored various approaches to automate the extraction of data from different types of charts, focusing on techniques such as image processing, deep learning, and optical character recognition (OCR). I searched for research papers, various R packages for each step of the process, and various published notebooks and publications on plot digitization. Following is a list containing all search results:

3.1 Research papers and articles

Title	Authors	Link	Description
Scatteract: Automated Extraction of Data from Scatter Plots	Mathieu Cliché, David Rosenberg, Dhruv Madeka, Connie Yee.	DOI:10.1007/978-3-319-71249-9_9	Illustrates an algorithm that leverages deep learning techniques and OCR techniques to retrieve chart coordinates. Lastly it uses robust regression to retrieve real values.
LINEEX: Data Extraction from Scientific Line Charts	Shivasankaran V P, Muhammad Yusuf Hassan, Mayank Singh	DOI:10.1109/WACV56688.2023.00615	Involves creation of a large line chart dataset, detection of various chart elements by detection transformer and detection of text by OCR. Employs OKS for keypoint extraction.
Automatic Extraction of Data from 2-D Plots in Documents	Xiaonan Lu, J.Z. Wang, Prasenjit Mitra, C.L. Giles	DOI:10.1109/ICDAR.2007.4378701	A binarized image is taken as an input. Axis detection is performed using a customised hough transform. Next, an image thinning algorithm is applied and the curves are identified.
Automatic Identification and Data Extraction from2-Dimensional Plots in Digital Documents	William J. Brouwer, Saurabh Kataria, Sujatha Das, Prasenjit Mitra, C. L Giles	https://www.researchgate.net/publication/220486631_Automatic_Identification_and_Data_Extraction_from_2-Dimensional_Plots_in_Digital_Documents	A binarized image is taken as an input. Axis detection is performed using a customised hough transform. Next, an image thinning algorithm is applied and the curves are identified.
Automatic Extraction of Data Points and Text Blocks from 2-Dimensional Plots in Digital Documents	Saurabh Kataria, William Browuer, Prasenjit Mitra, C. L Giles	https://www.researchgate.net/publication/221605706_Automatic_Extraction_of_Data_Points_and_Text_Blocks_from_2-Dimensional_Plots_in_Digital_Documents	A binarized image is taken as an input. Axis and plot region is segregated using SVM algorithm. Ticks are detected using OCR while simulated annealing is applied for datapoints disambiguation.
Segregating and extracting overlapping data points in two-dimensional plots	William J. Brouwer, Saurabh Kataria, Sujatha Das, Prasenjit Mitra, C. L Giles	DOI:10.1145/1378889.1378936	

Line Graphics Digitization: A Step Towards Full Automation	Omar Moured, Jiaming Zhang, Alina Roitberg, Thorsten Schwarz, Rainer Stiefelhagen	DOI:10.1007/978-3-031-41734-4_27	Creation of a line graph dataset of mathematical graphics and supports both semantic segmentation and object detection. Segmentation tasks are performed in pytorch.
Auto-Digitizer for Fast Graph-to-Data Conversion	Deepti Sanjay Mahajan, Sarah Pao Radzihovsky, Ching-Hua (Fiona) Wang	https://web.stanford.edu/class/ee368/Project_Winter_1718/Reports/Mahajan_Radzihovsky_Wang.pdf	Sobel filter is applied to images dataset for separation of axis and plot region. Label recognition is performed using OCR and data is extrapolated to the axis ranges.
Program for Automatic Numerical Conversion of a Line Graph	Michiko Yoshitake, Takashi Kono, Takuya Kadohira	https://doi.org/10.2477/jccj.2020-0002	OCR performed on an image containing many plots. Tensorflow and Keras used for deep learning on the backend. DBSCAN implemented to separate each plot image.

3.2 Available tools for digitization

Sl no.	Title	Link
1.	WebPlotDigitizer	automeris.io: AI assisted data extraction from charts using WebPlotDigitizer
2.	PlotDigitizer	PlotDigitizer — Extract Data from Graph Image Online
3.	GraphReader	graphreader.com - Online tool for reading graph image values and save as CSV / JSON
4.	Engauge Digitizer	Engauge Digitizer (markummitchell.github.io)
5.	DigitizeIt	DigitizeIt - Plot Digitizer Software. Digitize graphs, charts and math data.

4. Algorithm:

The proposed algorithm for extraction of data from line plots consists of three major parts. Image processing is the initial part of this algorithm. The main objective here, is to isolate the pixels of the image denoting the line plot from the other pixels that form the image of the plot. The implementation of this part may differ for some images based on specific requirements. Next, the optical character recognition detects the text in the image. The image of a line plot has labels depicting the values on each axis. The pixels values for the same, can be detected using bounding boxes. Finally, a model is trained to understand the relationship between pixel values and real values, and using regression, the real values are predicted. Each step of the algorithm is presented in detail as follows:

4.1 Plot generation

For generating a line plot, stock prices data for Apple Inc. for June 2024 has been used. This line plot will be recreated, and data will be extracted from it.

- Step 1: Retrieve stock data for last month. First, we obtain the stock data available in open source. Then, we filter out the stock prices and dates from the beginning of the previous month till the current date.
- Step 2: Plot the stock prices versus date data.
- Step 3: Save the plot as an image. This image will be used to extract data and recreation of the plot.

4.2 Image processing

The image of any plot is considered to have three basic parts: the x axis, the y axis, and the line plot area. Several image processing techniques can be used to segregate these parts of the plot. In the context of plot digitization, the **main objective of image processing is to extract the pixel values depicting the line plot**.

- Step 1: Load the image. Images can be read using different R packages.
- Step 2: Image processing for line plot isolation.
  Various functions can be used to isolate the line plot from the rest of the image but the techniques may vary for different images. Here, we are using smooth unsupervised contour detection to detect the pixels denoting the line plot.
- Step 3: Isolation of axes in the image. This step ensures accurate optical character recognition, an essential process further in the algorithm. Image processing techniques like image composition can be used to isolate the axes in images.

4.2.1 Contour detection on the image

Contour detection is a popular technique in computer vision useful in identification and segmentation on an image. Contours represent the boundary or outline of an object, consisting of pixels along the object's boundary. **Contour detection is utilized for the segregation of the pixel values of the line plot from the rest of the image.**

- Step 1: Convert image to the required format. There are different packages in R for contour detection. So, we must read the image in required format to use the contour detector functions.
- Step 2: Detect and plot the contourlines.
- Step 3: Filter out those contourlines which represent the line plot.

4.3 Optical Character Recognition for tick detection

Optical Character Recognition (OCR) is a technology used to convert different types of documents into editable and searchable data. Tesseract is one of the most popular open-source OCR engines, developed by Google. It supports multiple languages and provides highly accurate text recognition. **OCR was used to detect the ticks and their positions on the plot.**

- Step 1: Apply OCR on the image. I have used the English OCR engine of tesseract to extract the text data in the image.
- Step 2: Extract the recognized tick labels/values. I have used regular expressions to extract each tick value for both the axes.

4.4 Bounding boxes calculation

Bounding boxes are imaginary rectangles that enclose key objects within an image, defining their boundaries. When an OCR system recognizes text, it provides both the recognized words and their position in pixel values. **Each detected word or text block is enclosed by a bounding box, allowing precise localization.**

- Step 1: Y coordinates calculation. The mean of the bounding box limits are considered to denote the pixel values for the tick positions in the Y axis.
- Step 2: X coordinates calculation. Similarly, we can extract the position of X tick values in pixels. Since, X axis denotes the dates, I have converted it into numeric values for further process.

4.5 Regression and recreation of plot

Regression analysis is a statistical method used to model and analyze the relationships between a dependent variable and one or more independent variables. **In plot recreation, overfitting ensures that the recreated plot closely adheres to the intricate details of the original data.**

- Step 1: Apply a linear regression model based on tick pixel values versus tick real values. This models the relationship between tick pixel values and tick real values.
- Step 2: Predict x values and y values for the pixels data denoting the line plot. Due to the small training sample size and the large number of predictions, the linear model (lm) tends to overfit, thus ensuring accuracy in plot recreation
- Step 3: Recreate the plot with the predicted values.
- Step 4: Comparison of predicted prices with real prices.
- Step 5: Update the predictions due to the discrete nature of data in stock price dates.

5. Implementation in R:

5.1 Workflow

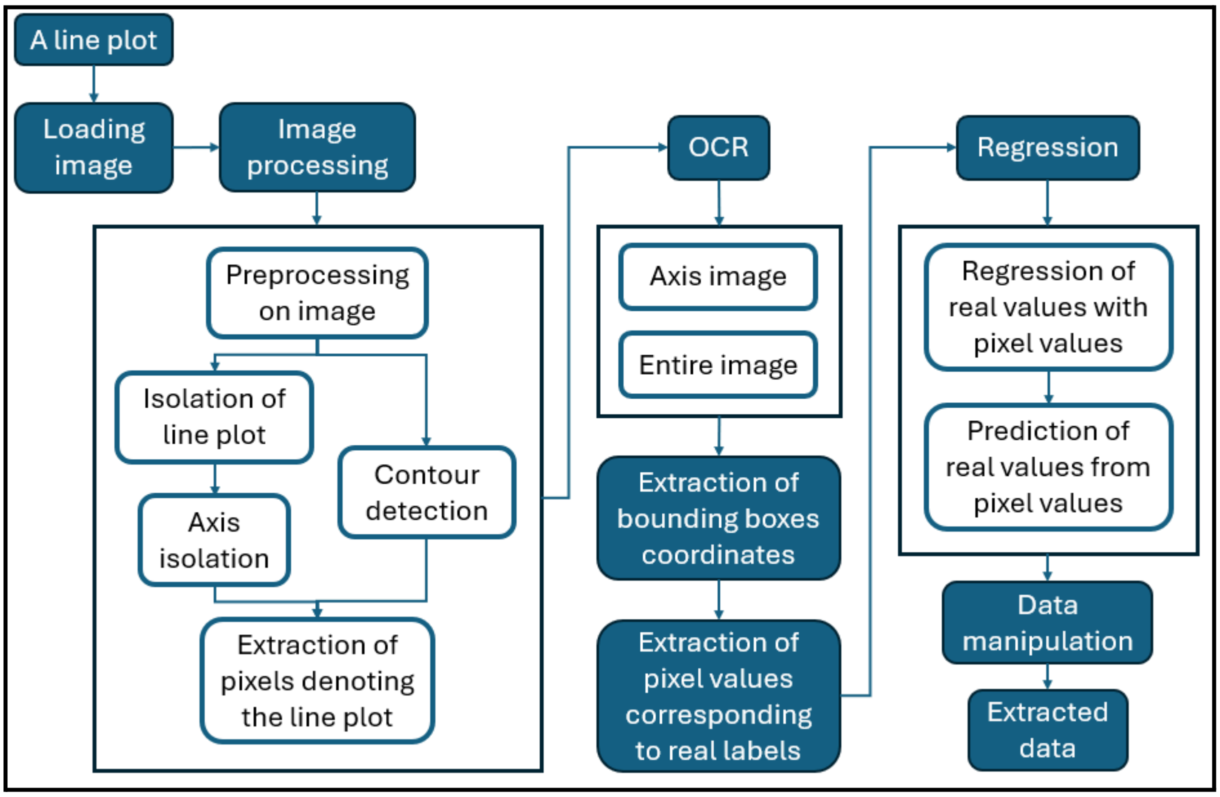

A line plot
Loading image
Image processing
Preprocessing on image
Isolation of line plot
Contour detection
Axis isolation
Extraction of pixels denoting the line plot
OCR
Axis image
Entire image
Extraction of bounding boxes coordinates
Extraction of pixel values corresponding to real labels
Regression
Regression of real values with pixel values
Prediction of real values from pixel values
Data manipulation
Extracted data

Figure: Proposed algorithm

5.2 Packages used

- **imager**: For image processing tasks, such as edge detection and segmentation.
- **magick**: For advanced image manipulation and transformations.
- tesseract: For optical character recognition (OCR) to extract textual data from images.
- **dplyr**: For data analysis and manipulation, facilitating efficient data processing workflows.
- **stringr**: For string operations and data cleaning.
- ggplot2: For visualizing the digitized data, enabling the creation of accurate and aesthetically pleasing plots.
- **quantmod**: For stock data analysis, providing tools to model and analyze the stock prices of AAPL.
- **tesseract**: For optical character recognition to get the precise pixel value of the ticks.
- **image.ContourDetector**: For detection of contours on the image.
- **Lubridate**: For data manipulation and analysis.

5.3 Procedure

5.3.1 Plot generation

- **Step 1**: Retrieve stock data for last month

  a) A variable **ticker** is defined with the value "**AAPL**", which is the ticker symbol for Apple Inc. Ticker symbols are unique identifiers for traded stocks.

  b) The **getSymbols** function from the **quantmod** package is used to download the stock data for the specified ticker symbol. By default, this function retrieves historical stock data from Yahoo Finance.

  c) To extract prices and dates from the stock data of Apple Inc., I utilized the following:

     I. **get(ticker)$AAPL.Adjusted** accesses the adjusted closing prices from the retrieved data. Closing prices are adjusted for actions like dividends and splits.

     II. **data.frame(Date = ..., Price = ...)** creates a data frame with two columns: Date and Price.

  d) Here are the functions and methods used to retrieve the stock data for the previous month:

I. **format(as.Date(current_date), "%Y-%m-01")**: Converts the **current_date** to a string representing the first day of the current month in the format "YYYY-MM-01".

II. Then, 31 days are subtracted from the first day of the current month to get the variable **last_month_start**.

III. **stock_data_last_month** filters out and stores that part of **stock_data** which is within the **current_date** and the **last_month_start.**

- **Step 2:** Plot the stock data and save it as an image.

  a) The stock data was plotted using the **ggplot** function from **ggplot2** package and the theme was set to **theme_classic**.

  b) This plot was saved using the **ggsave()** function.

- **R Code implementation for Plot generation:**

```r
# 1. Plot generation

# Step 1: Retrieve stock data for last month
# Step 1(a) Name of the stock
ticker <- "AAPL"
# Step 1(b) Retrieving the stock data
getSymbols(ticker)

# Step 1(c) Stock data to dataframe
stock_data <- data.frame(Date = index(get(ticker)), Price = as.numeric(get
(ticker)$AAPL.Adjusted))
current_date <- Sys.Date()
# Step 1(d) Stock data for the previous month
last_month_start <- as.Date(format(as.Date(current_date), "%Y-%m-01")) -30
stock_data_last_month <- stock_data[stock_data$Date >= last_month_start &
stock_data$Date <= current_date,]

# Step 2: Plot the stock data and save as an image.
# Step 2(a) Plotting stock prices versus dates

real_plot<-ggplot(stock_data_last_month, aes(x = Date, y = Price)) +
  geom_line() +
  theme_classic()

print(real_plot)
# Step 2(b) Saving the plot as an image
ggsave("stock_plot.jpeg",real_plot)
```

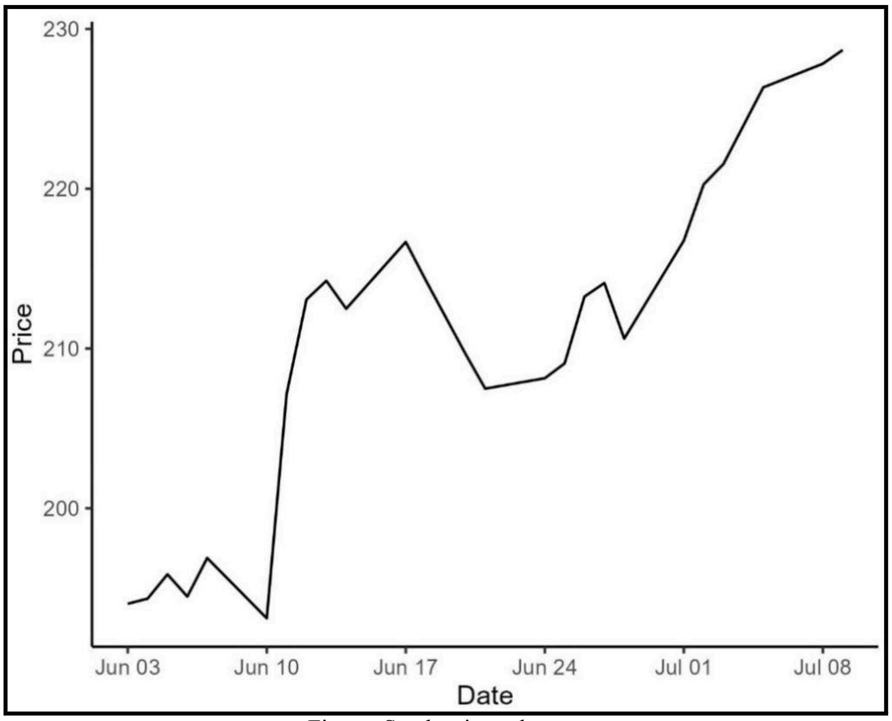


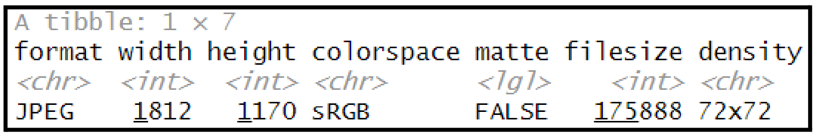
Figure: Stock prices plot

5.3.2 Contour detection on the image

- **Step 1:** Loading the image

  a) Image is loaded using **image_read** function in **magick**.

  b) Image data is shown below



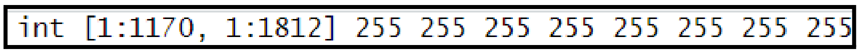
```
A tibble: 1 × 7
format width height colorspace matte filesize density
<chr>  <int>  <int> <chr>      <lgl>    <int> <chr>
JPEG    1812   1170 sRGB       FALSE   175888 72x72
```

- **Step 2:** Converting image to required format

  a) We are using the package **image.ContourDetector** which uses unsupervised smooth contour detection.

  b) It accepts the image formats in only "gray" and "bitmap" channels. Thus, **image_data** function is used and channel is set to "grey".

  c) The data is stored in variable **img_data** which is a matrix. The following code is adapted from the CRAN **image.contourdetection** documentation.



```
int [1:1170, 1:1812] 255 255 255 255 255 255 255 255
```

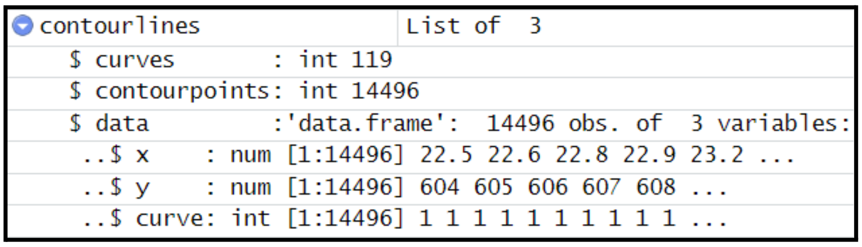
Figure: variable **img_data**

- **Step 3: Detect and plot the detected contour lines.**

  a) The function **image_contour_detector** is used to detect the contour lines.

     I. Data is stored in the variable **contourlines**.
     II. **mat** is the matrix of the image pixel values in the 0-255 range.
     III. **Q** is numeric value with the pixel quantization step. Lower Q leads to more accuracy in detection but higher complexity.

  b) The **contourlines** data is shown below:



```
contourlines              List of  3
   $ curves        : int 119
   $ contourpoints: int 14496
   $ data          :'data.frame':  14496 obs. of  3 variables:
    ..$ x     : num [1:14496] 22.5 22.6 22.8 22.9 23.2 ...
    ..$ y     : num [1:14496] 604 605 606 607 608 ...
    ..$ curve: int [1:14496] 1 1 1 1 1 1 1 1 1 1 ...
```

Figure: Contourlines data

     I. **contourlines** is a list with 2 integer arrays namely **curves** and **contourpoints** and a dataframe called **data** with 3 columns(**x**, **y**, **curve**).
     II. The **curves** array contains the number of curves detected while the **contourpoints** array detects the number of contourpoints detected.
     III. The data that is of our interest here is **contourlines$data**. Here **x** and **y** columns denote the pixel coordinates and **curve** denotes the particular contour detected.

  c) Our objective is to find the contour points pertaining to the line plot so that we can digitize the plot. Therefore, visualization of the **contourlines$data** by color is necessary so that each detected curve is represented in a different color.

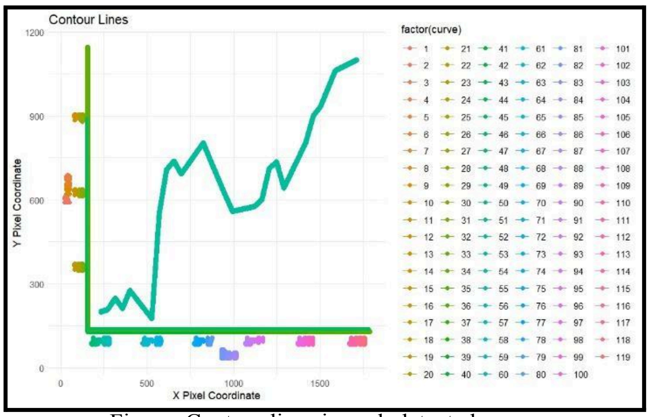


Figure: Contour lines in each detected curve

d) Next, the suitable range for curves is visualized for identification of the curve denoting the line plot.

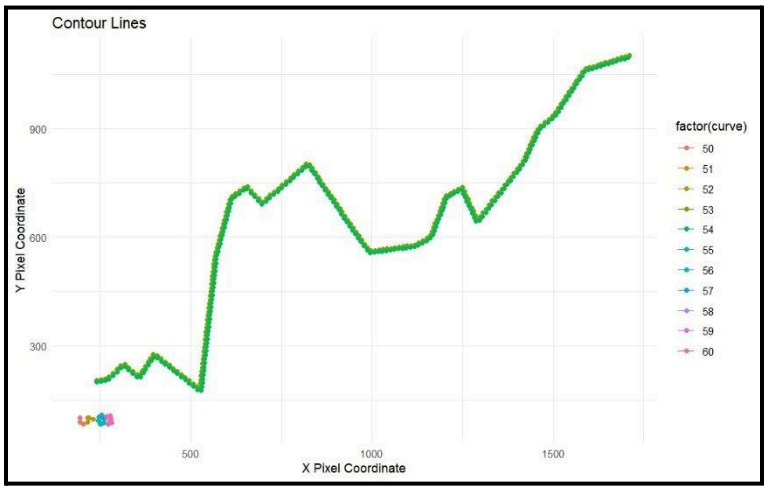


Figure: Contour lines denoting line plot

- **Step 4:** Filter out and plot the line data

a) Filter out the contours of the line plot using **filter** function.

b) Plot the filtered line data.

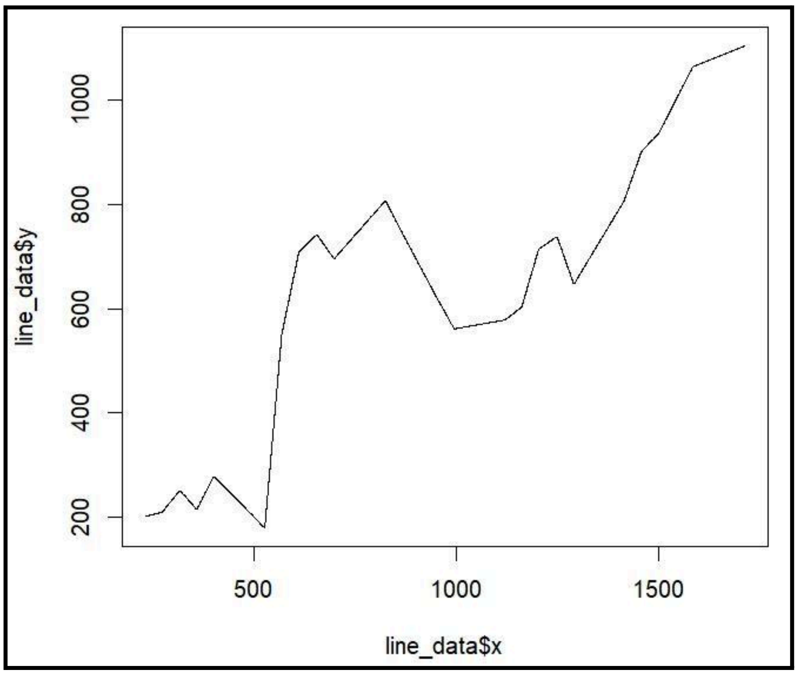


Figure: Line plot in pixel values

- **R implementation for Contour detection:**

```r
# 2. Contour detection

# Step 1: Loading the image

# Step 1(a): Image Loaded using magick package
img <- image_read("stock_plot.jpeg")

# Step 1(b): Print image
print(img)

# Step 2: Convert image to required format

# Step 2(a): From image to grayscale (unreadable element)
img_grey <- image_data(img, channels = "gray")

# Step 2(b): Conversion to integer will convert it to colour:
# values for each pixel: 0 for black 255 for white
img_grey_t <- as.integer(mat_gray, transpose = TRUE)

# Dropping the colour channel as it is not required for greyscale
img_gray_t <- drop(img_data)


# Step 3: Detect and plot the contourlines

# Step 3(a): Detect the contourlines
contourlines <- image_contour_detector(img_grey_t, Q=0.001)

# Step 3(c): Plot the contourlines data by each curve
ggplot(contourlines$data, aes(x = x, y = y, color = factor(curve))) +
  geom_point() +
  geom_line() +
  labs(title = "Contour Lines", x = "X Pixel Coordinate",
y = "Y Pixel Coordinate") +
  theme_minimal()

# Step 3(d): Subset the data to filter out the contourline needed
ggplot(subset(contourlines$data, curve = 50:60),
aes(x = x, y = y, color = factor(curve))) +
  geom_point() +
  geom_line() +
  labs(title = "Contour Lines", x = "X Pixel Coordinate",
y = "Y Pixel Coordinate") +
  theme_minimal()

# Step 4: Filter out and plot the line data.

# Step 4(a): Filter out the contours of the line plot
line_data <- contourlines$data %>% filter(curve == 53)
# Step 4(b): Plot the line data
plot(line_data$x, line_data$y, type ='l')
```

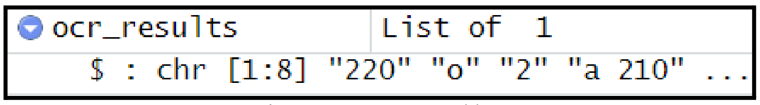
5.3.3 Optical character recognition for tick detection

- **Step 1: Extract data from image using OCR:**

    a) Next, we extract data from the image as shown below.

    I. **"ocr"** function performs OCR on text using the Tesseract engine for English.
    II. **"str_split"** splits the OCR result into lines and stores them in **"ocr_results"** variable.

    b) Shown below, is the data extracted by OCR:



```
ocr_results          List of  1
   $ : chr [1:8] "220" "o" "2" "a 210" ...
```

Figure: Data extracted by OCR

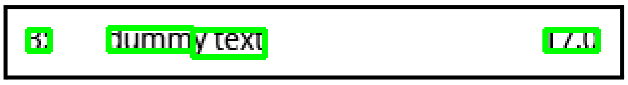


Figure: Bounding boxes

- **Step 2: Extract the recognized tick values**

    a) **str_extract_all** function extracts all the string values in **ocr_results** which was obtained by applying OCR engine previously.

    b) Regular expression **\\b\\d{3}\\b** is used for tick values in y axis and **[A-Za-z]{3} \\d{2}** is used for the tick values in x axis.
    I. **\\b** denotes word boundary.
    II. **\\d** denotes a digit and d{3} means we are extracting 3 digits.
    III. **[A-Za-z]** denotes all the alphabet characters.

    c) **unlist** function is used to convert the recognized and detected tick values to a vector. The following are the results for y tick values and x tick values, respectively.

- **R implementation for OCR:**

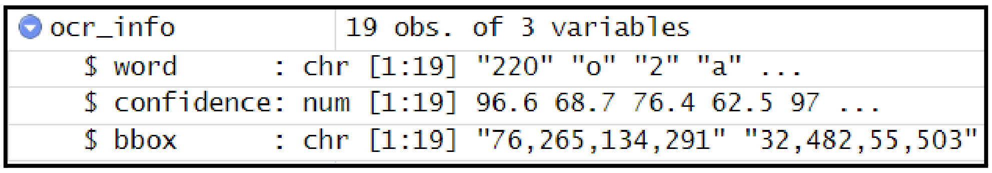
```r
# 4. OCR for tick detection

# Step 1: Apply OCR and extract data from the image
# Step 1(a) Extract data using OCR
ocr_results <- ocr(img, engine = tesseract("eng"))%>% str_split("\n")
```

```r
# Step 2: Extract the recognized tick values
y_labels <- unlist(str_extract_all(ocr_results, "\\b\\d{3}\\b"))
x_labels <- unlist(str_extract_all(ocr_results, "[A-Za-z]{3} \\d{2}"))
```

5.3.4 Bounding boxes calculation

- **Step 1: Extracted data from OCR in a dataframe**



```
ocr_info              19 obs. of 3 variables
   $ word      : chr [1:19] "220" "o" "2" "a" ...
   $ confidence: num [1:19] 96.6 68.7 76.4 62.5 97 ...
   $ bbox      : chr [1:19] "76,265,134,291" "32,482,55,503"
```

Figure: OCR data

I. The **ocr_info** dataframe has 3 columns: **word**, **confidence** and **bbox**.
II. **word** denotes the recognized characters in the image.
III. **confidence** denotes the confidence level with which the OCR engine detects the text.
IV. **bbox** defines the positions of the ticks. For example, **76, 265, 134, 291** are the bounding dimensions, where the first and third elements denote the pixel value range in x axis and the other elements denote the pixel value range in y axis.

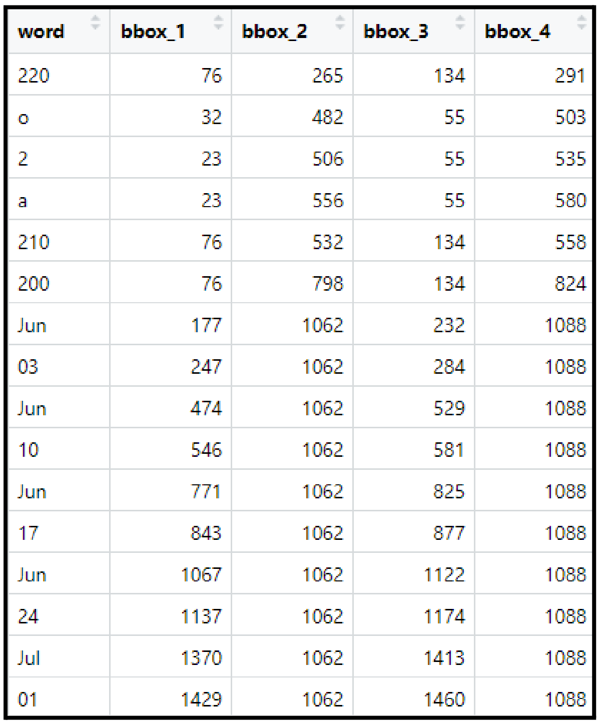
- **Step 2: Y coordinates calculation**

a) Find the bounding box dimensions:

I. **as.character(ocr_info$bbox)** extracts the **bbox** column from the **ocr_info** dataframe and converts it to a character vector.
II. **strsplit(..., ",")** splits each string in the **bbox** column by the comma delimiter, resulting in a list of vectors while **do.call(rbind, ...)** combines these vectors into a dataframe by row-binding.
III. **seq_len(...)** creates a sequence from 1 to length of the dataframe and **paste("bbox",sep = "_")** constructs the column names by concatenating **bbox**, the column number, and an underscore.
IV. **bounding_coord** is finally obtained by column binding the words and their bounding box dimensions.

b) Therefore, **bbox_1** and **bbox_3** store the range of pixel values in x direction and **bbox_2** and **bbox_4** store the range of pixel values in the y direction.

c) The resulting **bounding_coord**, formed is presented below.



word	bbox_1	bbox_2	bbox_3	bbox_4
220	76	265	134	291
o	32	482	55	503
2	23	506	55	535
a	23	556	55	580
210	76	532	134	558
200	76	798	134	824
Jun	177	1062	232	1088
03	247	1062	284	1088
Jun	474	1062	529	1088
10	546	1062	581	1088
Jun	771	1062	825	1088
17	843	1062	877	1088
Jun	1067	1062	1122	1088
24	1137	1062	1174	1088
Jul	1370	1062	1413	1088
01	1429	1062	1460	1088

Figure: Bounding boxes coordinates

d) This dataframe is then filtered such that we get the data only for the y axis ticks.

e) It is estimated that the mean of the maximum and minimum bounding box pixel values represents the real tick in pixel values.

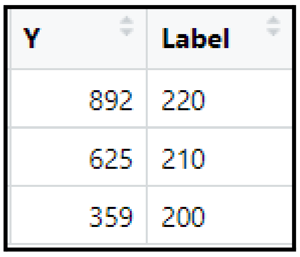
f) After this calculation the extracted tick pixel values are stored in a column named **Y** in another dataframe **y_coord** while the real tick values detected in OCR are stored in a column named **Label**. **y_coord** is demonstrated below.

- **Step 3: X coordinates calculation**

a) Filter the OCR results of the x-labels similar to the extraction of y-labels.

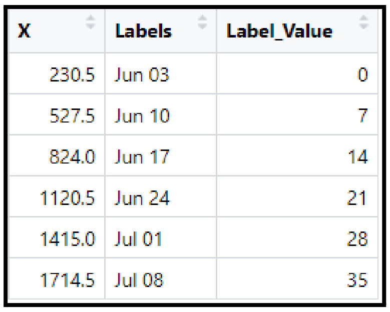
b) Calculation of x-coordinates of the label ticks. Compute the X-coordinates of labels by averaging the left and right bounding box coordinates in an alternate sequence. This is done by iteration using a **for** loop.

c) Convert the dates into numeric values by **as.Date** function and the **Label_Value** is found out by creating a sequence upto the difference between the detected dates. This is stored in the variable, **x_coord**.



Y	Label
892	220
625	210
359	200

Figure: Y ticks pixel and real values



X	Labels	Label_Value
230.5	Jun 03	0
527.5	Jun 10	7
824.0	Jun 17	14
1120.5	Jun 24	21
1415.0	Jul 01	28
1714.5	Jul 08	35

Figure: X ticks real and pixel values

- **R implementation for bounding box calculations:**

```
# 4. Bounding Box Calculation

# Step 1: Extract the bounding box dimensions
ocr_info <- ocr_data(img, engine = tesseract("eng"))%>% str_split("\n")

# Step 2: Y coordinates calculation
# Step 2(a): Find the bounding box dimensions
bounding_coord <-data.frame(do.call(rbind,
strsplit(as.character(ocr_info$bbox), ",")))
colnames(bounding_coord) <- paste("bbox",
seq_len(ncol(bounding_coord)),sep="_")
bounding_coord <- apply(bounding_coord, 2, as.numeric)
bounding_coord <- cbind(ocr_info[,"word"], bounding_coord)


# Step 2(d): Filtering the words corresponding to Y-labels
y_label_boun_box<-bounding_coord
%>%filter(bounding_coord$word)%in%y_labels)


# Step 2(f): Calibrating the Y-coordinates
img_height <- as.numeric(image_info(img)[1, 3])
y_label_boun_box[, c(3, 5)] <- img_height -  y_label_boun_box[, c(3, 5)]
y_coord <- data.frame(rowMeans(y_label_boun_box[, c(3, 5)]))
colnames(y_coord) <- "Y"
y_coord$Labels <-(y_labels)

# Step 3: X coordinates calculation
# Step 3(a): Filter the OCR result for x-labels
x_label_boun_box <- bounding_coord %>% filter(bounding_coord$word) %in%
unlist(str_split(x_labels, ," ")))


# Step 3(b): Calculation of x-coordinates of the label ticks
x_coord<- c()
for(i in seq(1,nrow(x_label_boun_box),by=2)){
  left_x <- as.numeric(x_label_boun_box [i,"bbox_1"])
  right_x <- as.numeric(x_label_boun_box [i+1,"bbox_3"])
  x_coord <- append(x_coord,((left_x + right_x)/2))}
x_coord <- data.frame(X = x_coord, Labels = x_labels)
```

```
# Step 3(c): Conversion of dates to numeric values
date_values <- as.Date(x_values, format = "%b %d")
x_coord$Label_Value <- seq(0, as.numeric(difftime(max(date_values),
min(date_values), units = "days")), length.out = length(x_labels))
```

5.3.5 Regression and recreation of plot

- **Step 1**: **Fit a model to learn from the contour values and given labels:**
  lm function was used to create a linear model based on the detected tick values and their positions in pixel values for both the axes.

- **Step 2**: **Predict x values and y values for the contourlines data:**
  Values in x and y axis were predicted using the **predict** function on the **line_data** containing the contourlines data for the line plot.

- **Step 3: Recreate the plot:**
  The **plot** function was used to recreate the plot as shown below. The x axis denotes the number of days starting from the start of the previous month.

- **Step 4: Convert continuous x values to discrete corresponding to dates:**

  I. **x_discrete_convert** function finds the Y value in **predicted_data** closest to a given X value (day).
  II. **x_close** subsets **predicted_data** to rows where X rounded to the nearest integer equals day.
  III. **x_close$diff** calculates the absolute difference between X and day and then the row with the smallest difference is selected

- **Step 5: Final recreated plot for the discrete x values:**

  I. **x_updated** creates a sequence of X values from 0 to the maximum of x_predicted, rounded to the nearest integer.
  II. **y_updated** applies **x_discrete_convert** function to each value in **x_updated** to get the corresponding Y values.
  III. **predicted_data** is a dataframe that stores the updated X and Y values and this data is the plotted.
- **R implementation for regression and recreation of plot:**

```
# 5. Regression and recreation of plot
```

```r
# Step 1: Fit a model to learn from the contour values and given labels
ymodel <- lm(Value ~ Y, data = y_coord)
xmodel <- lm(Value ~ X, data = x_coord)

# Step 2: Predict the label values for all the contour values
y_predicted <-predict(ymodel, newdata = data.frame(Y=line_data$y))
x_predicted <- predict(xmodel, newdata = data.frame(X = line_data$x))

# Step 3: Plot the recreated data
plot(x_predicted, y_predicted, type = 'l')
predicted_data <- data.frame (X= x_predicted,Y= y_predicted)

# Step 4: Convert continuous x values to discrete corresponding to dates
x_discrete_convert <- function(day){
# Find X-value rounded off to integer and find their differences
x_close <- subset(predicted_data, round(X) == day)
x_close$diff <- abs(x_close$X - day) # Identify closest Y value
x_value <- x_close[x_close$diff == min(x_close$diff),]
return(x_value$Y)
}

# Step 5: Final predicted plot
x_updated <- 0:round(max(x_predicted))
y_updated <- sapply(x_updated, x_discrete_convert)
predicted_data <- data.frame(X= x_updated, Y = y_updated)
plot(predicted_data, type = 'l')
```

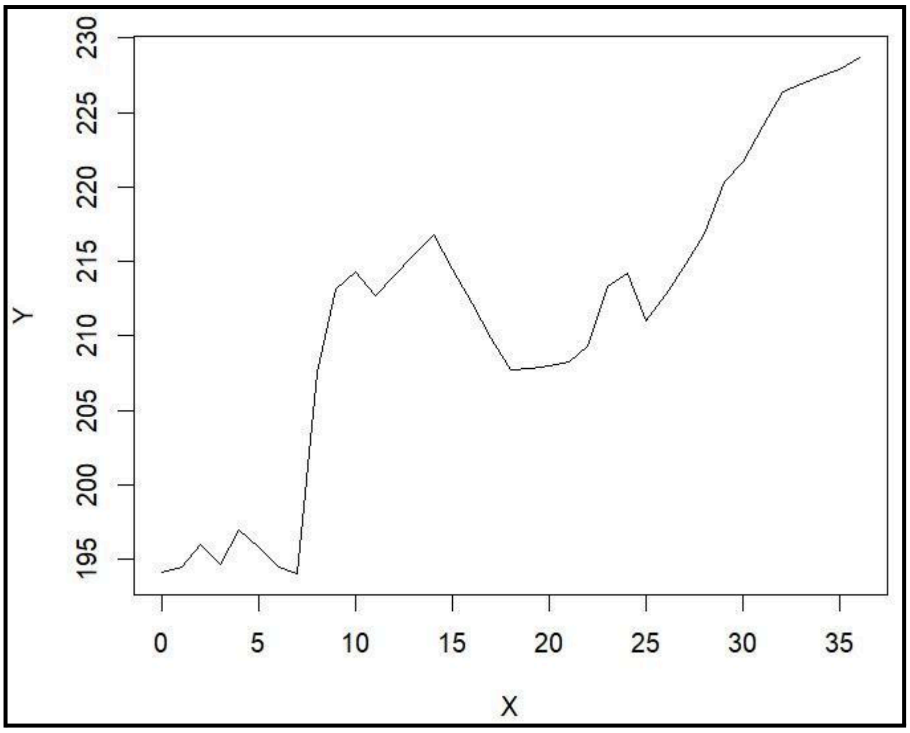


Figure : Recreated plot

5.3.6 Comparison of original and recreated plot:

- **Step 1: Conversion of dates in original data to numbers for comparison:** Treating each date as a unique day, **x_scaled** is created for each date. A function converting date into numeric by calculating the number of days is used. This function is applied to all dates in stock prices data to create the **x_scaled** array.

- **Step 2: Filtering the prediction data to predict the price for each date:** The **predicted_data** dataframe is filtered such that the x values in **predicted_data** match with **x_scaled** array. This provides us with the relevant data for dates and prices which is then used to compare the real stock prices with the predicted stock prices.

- **Step 3: Comparison of original and recreated plots:** The **ggplot** function is used to visualise the recreated and the original plots.

- **Step 4: Mean squared error and mean percentage difference is calculated.**

```
# Step 1: Realtime conversion of the dates to numbers

x_dat <- as.Date(stock_data_last_month$Date)
x_scaled <- sapply(x_dat, function(x)
{as.numeric(difftime(x, x_dat[1],units = "days"))})

# Step 2: Filtering the prediction data to predict the price for each date
price_predict <- predicted_data[predicted_data$X %in% x_scaled, "Y"]

# Step 3: Comparison of original and recreated plots.
ggplot(stock_data_last_month, aes(x = Date)) +
  geom_line(aes(y = Price, color = "Real"),size=2) +
  geom_line(aes(y = price_predict, color = "Predicted"),size=2) +
  labs(title = "Two Line Plots on the Same Graph",
       x = "X-axis",
       y = "Y-axis",
       color = "Legend") +
  theme_minimal()
```

```
stock_data_last_month$predicted_prices <- price_predict
stock_data_last_month$percentage_diff <-
((stock_data_last_month$Price - stock_data_last_month$predicted_price)/
stock_data_last_month$Price) * 100
mean_percentage_diff<-mean(stock_data_last_month$percentage_diff,
na.rm = TRUE)
print(mean_percentage_diff)
## [1] -0.08093236

mse<-mean((stock_data_last_month$Price -
stock_data_last_month$predicted_price)^2)
```

```
print(mse)
## [1] 0.04220163
```

6. Results

The discussed algorithm is a comprehensive approach to plot digitization and is applicable to various images of different size and formats. Consequently, it is also important that the implementation of this algorithm may vary for specific charts.
The implementation of this algorithm in R language demonstrates the versatility of the algorithm. We have obtained an **MSE** of **0.042** and a **mean percentage difference** of **0.08%** between the real stock prices and the predicted stock prices.

Comparison between the real data and the predicted data for stock prices versus dates:

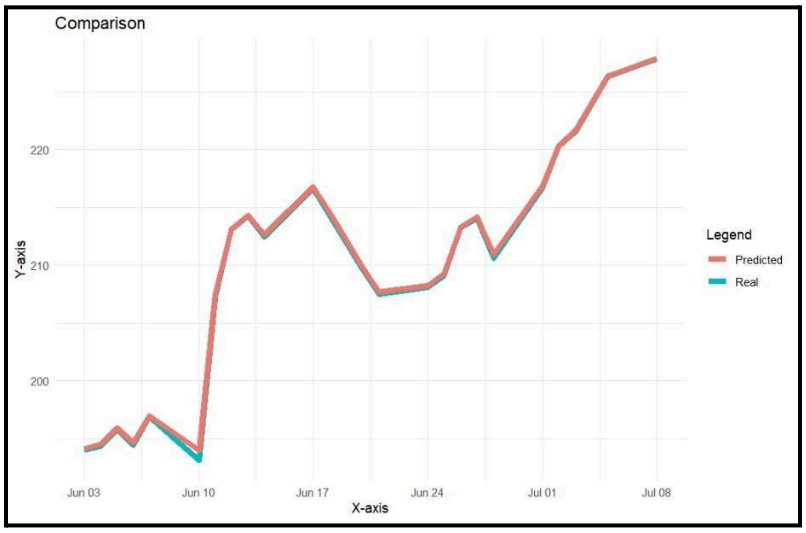
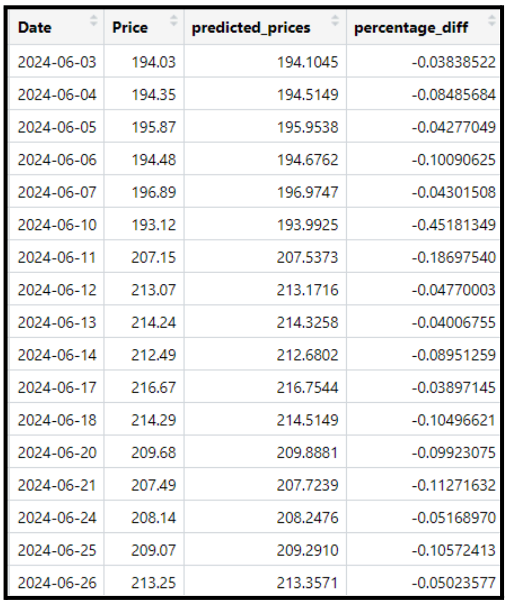


Figure: Line plots for real and predicted prices



Date	Price	predicted_prices	percentage_diff
2024-06-03	194.03	194.1045	-0.03838522
2024-06-04	194.35	194.5149	-0.08485684
2024-06-05	195.87	195.9538	-0.04277049
2024-06-06	194.48	194.6762	-0.10090625
2024-06-07	196.89	196.9747	-0.04301508
2024-06-10	193.12	193.9925	-0.45181349
2024-06-11	207.15	207.5373	-0.18697540
2024-06-12	213.07	213.1716	-0.04770003
2024-06-13	214.24	214.3258	-0.04006755
2024-06-14	212.49	212.6802	-0.08951259
2024-06-17	216.67	216.7544	-0.03897145
2024-06-18	214.29	214.5149	-0.10496621
2024-06-20	209.68	209.8881	-0.09923075
2024-06-21	207.49	207.7239	-0.11271632
2024-06-24	208.14	208.2476	-0.05168970
2024-06-25	209.07	209.2910	-0.10572413
2024-06-26	213.25	213.3571	-0.05023577

Table: Stock prices, predicted prices and percentage difference on various dates

Chapter 6

Conclusion

The projects completed during the FOSSEE Semester Long fellowship contributed towards the increment in usability and awareness of open-source software, i.e., R. Completed R TBCs are made available to the general public to be used as a companion to the associated standard textbooks in mathematics and sciences. The plot digitization project demonstrates R's capabilities in transforming visual data into a digital format that can be analyzed and interpreted quantitatively. This process aids users in extracting data from graphical representations.

Overall, it was a great learning experience. I gained new skills and knowledge. I also learned the different facets of working within an organization. In a nutshell, the fellowship taught me work ethics, commitment, and the importance of contributing back to society, besides technical skills.