**Major – 2**

**1)Collect Fundamental Details**

from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.common.exceptions import NoSuchWindowException

import time

import json

# Set up the Chrome webdriver (you need to download chromedriver and specify its path)

driver = webdriver.Chrome()

def generate\_stock\_link(stock\_logo):

base\_url = 'https://www.screener.in/company/'

return base\_url + stock\_logo.upper() + '/consolidated'

def get\_profit\_and\_loss\_details(driver):

data = {}

# Find the profit and loss

profit\_and\_loss = driver.find\_element(By.ID, 'profit-loss')

# Find the tables inside the section

tables = profit\_and\_loss.find\_elements(By.CLASS\_NAME, 'ranges-table')

for table in tables:

# Extract the table title

table\_title = table.find\_element(By.TAG\_NAME, 'th').text.strip()

# Initialize an empty dictionary to store data for this table

table\_data = {}

# Find all rows in the table body

rows = table.find\_elements(By.TAG\_NAME, 'tr')

# Iterate over each row (skip the first row which contains the title)

for row in rows[1:]:

# Extract the cells in this row

cells = row.find\_elements(By.TAG\_NAME, 'td')

# Extract the time period (e.g., "10 Years") and the value (e.g., "66%")

time\_period = cells[0].text.strip()

value = cells[1].text.strip()

# Store the value in the table data dictionary

table\_data[time\_period] = value

# Store the table data in the main data dictionary with the table title as key

data[table\_title] = table\_data

return data

def get\_revenue\_and\_net\_profit\_details(driver, section\_class):

# Find the section

section = driver.find\_element(By.ID, section\_class)

# Find the content div inside the section

content\_div\_section = section.find\_element(By.CLASS\_NAME, 'fill-card-width')

# Find the table inside the content div

table\_section = content\_div\_section.find\_element(By.CLASS\_NAME, 'data-table')

# Find the table header (thead) element

thead = table\_section.find\_element(By.TAG\_NAME, 'thead')

# Find all th elements within the thead

th\_elements = thead.find\_elements(By.TAG\_NAME, 'th')

# Initialize an empty array to store the table header content

table\_header = []

# Iterate through each th element and extract its text content

for th in th\_elements[1:]:

table\_header.append(th.text.strip())

# Find the table body (tbody) element

tbody = table\_section.find\_element(By.TAG\_NAME, 'tbody')

# Find stripe rows

tr\_stripe\_elements = tbody.find\_elements(By.CLASS\_NAME, 'stripe')

tr\_strong\_elements = tbody.find\_elements(By.CLASS\_NAME, 'strong')

# Initialize an empty array to store the Revenue data

Revenue = []

Net\_Profit = []

# Iterate through stripe rows to find Revenue

for tr\_stripe in tr\_stripe\_elements:

td\_elements = tr\_stripe.find\_elements(By.TAG\_NAME, 'td')

td\_head = td\_elements[0]

if td\_head.text.strip() == "Revenue":

for td in td\_elements[1:]:

Revenue.append(td.text.strip())

# Iterate through strong rows to find Net Profit

for tr\_strong in tr\_strong\_elements:

try:

button\_element = tr\_strong.find\_element(By.TAG\_NAME, 'button')

if "Net Profit" in button\_element.text:

td\_elements = tr\_strong.find\_elements(By.TAG\_NAME, 'td')

for td in td\_elements[1:]:

Net\_Profit.append(td.text.strip())

except NoSuchElementException:

pass

return {"Headers" : table\_header, "Revenue" : Revenue, "Net Profit" : Net\_Profit}

stock\_logos = ["HDFCBANK", "ICICIBANK", "KOTAKBANK", "SBIN", "BAJFINANCE", "ITC", "HINDUNILVR", "TATAMOTORS", "BAJAJ-AUTO",

"MARUTI", "MRF", "INFY", "TCS", "WIPRO", "SUNPHARMA", "CIPLA", "RELIANCE", "LT", "ASIANPAINT", "TITAN"]

data = {}

for stock\_logo in stock\_logos:

stock\_link = generate\_stock\_link(stock\_logo)

# Delay for 2 seconds before navigating

time.sleep(2)

try:

# Navigate to the screener.com page for the specified stock

driver.get(stock\_link)

except NoSuchWindowException as e:

print("Error: No window available. Make sure the browser window is open.")

time.sleep(5)

data[stock\_logo] = {

"Quarters": get\_revenue\_and\_net\_profit\_details(driver, "quarters"),

"Annual": get\_revenue\_and\_net\_profit\_details(driver, "profit-loss"),

"ProfitAndLoss": get\_profit\_and\_loss\_details(driver)

}

# Save the data to a JSON file

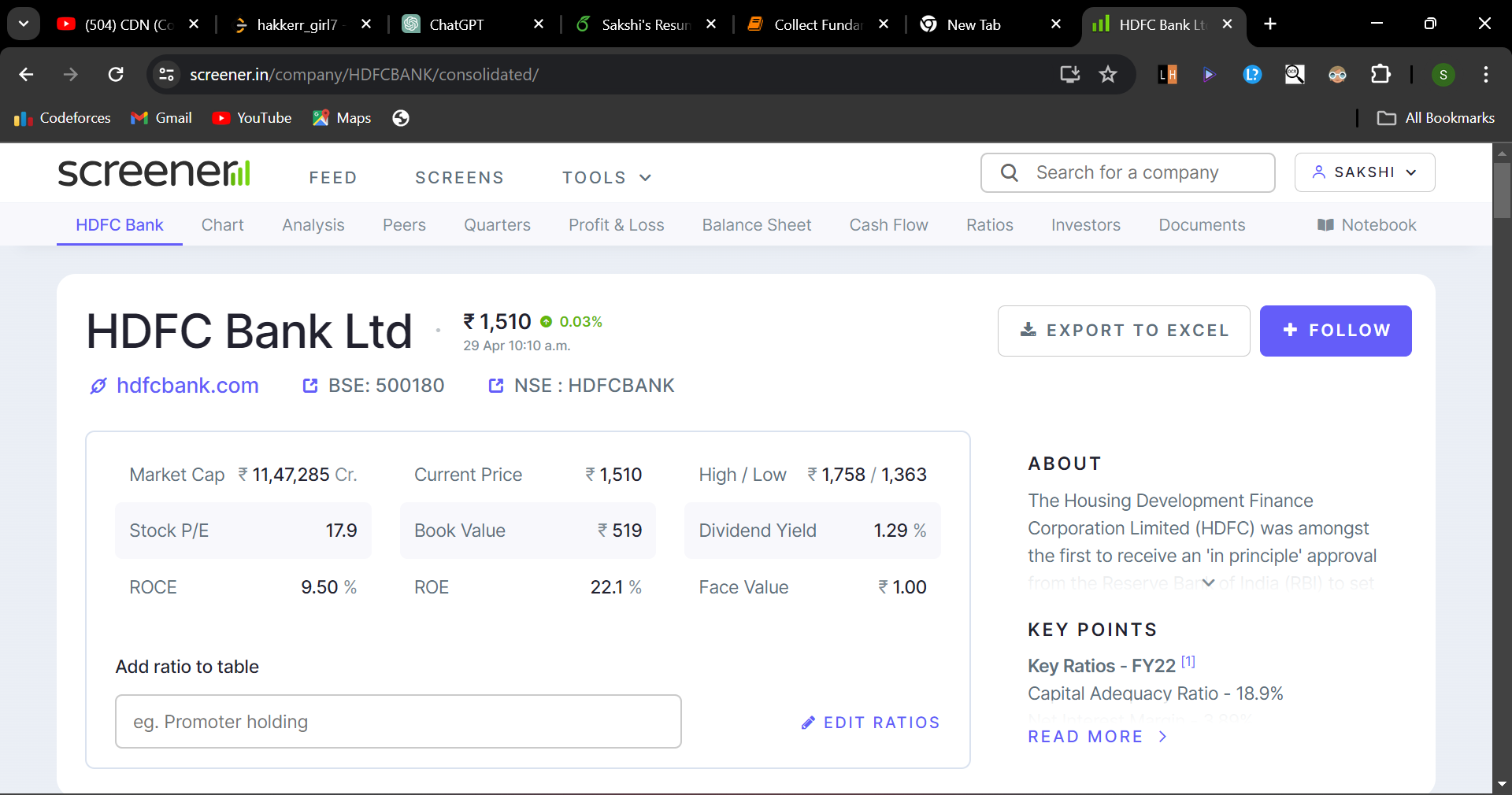
with open('stock\_data.json', 'w') as json\_file:

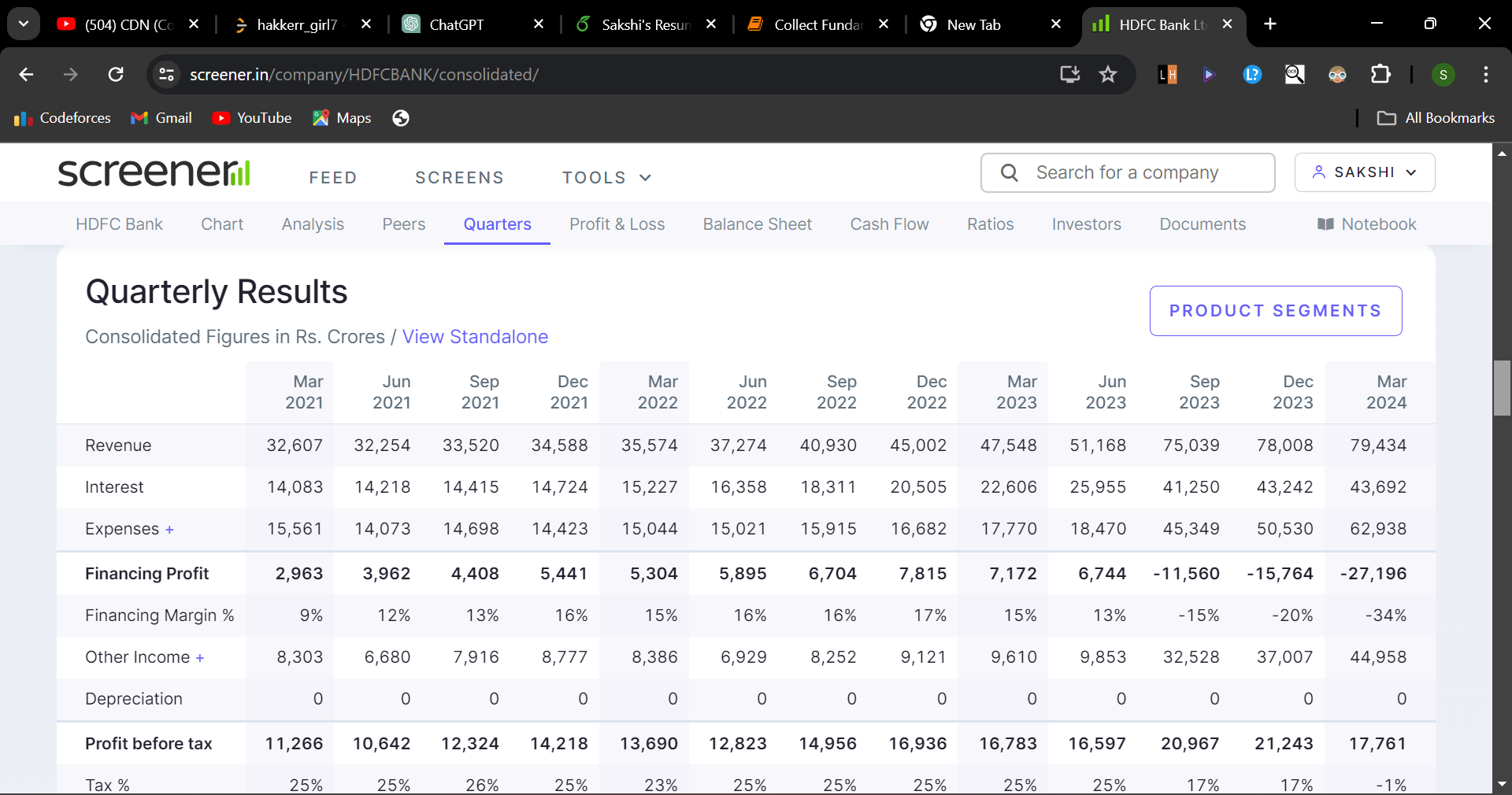
json.dump(data, json\_file)

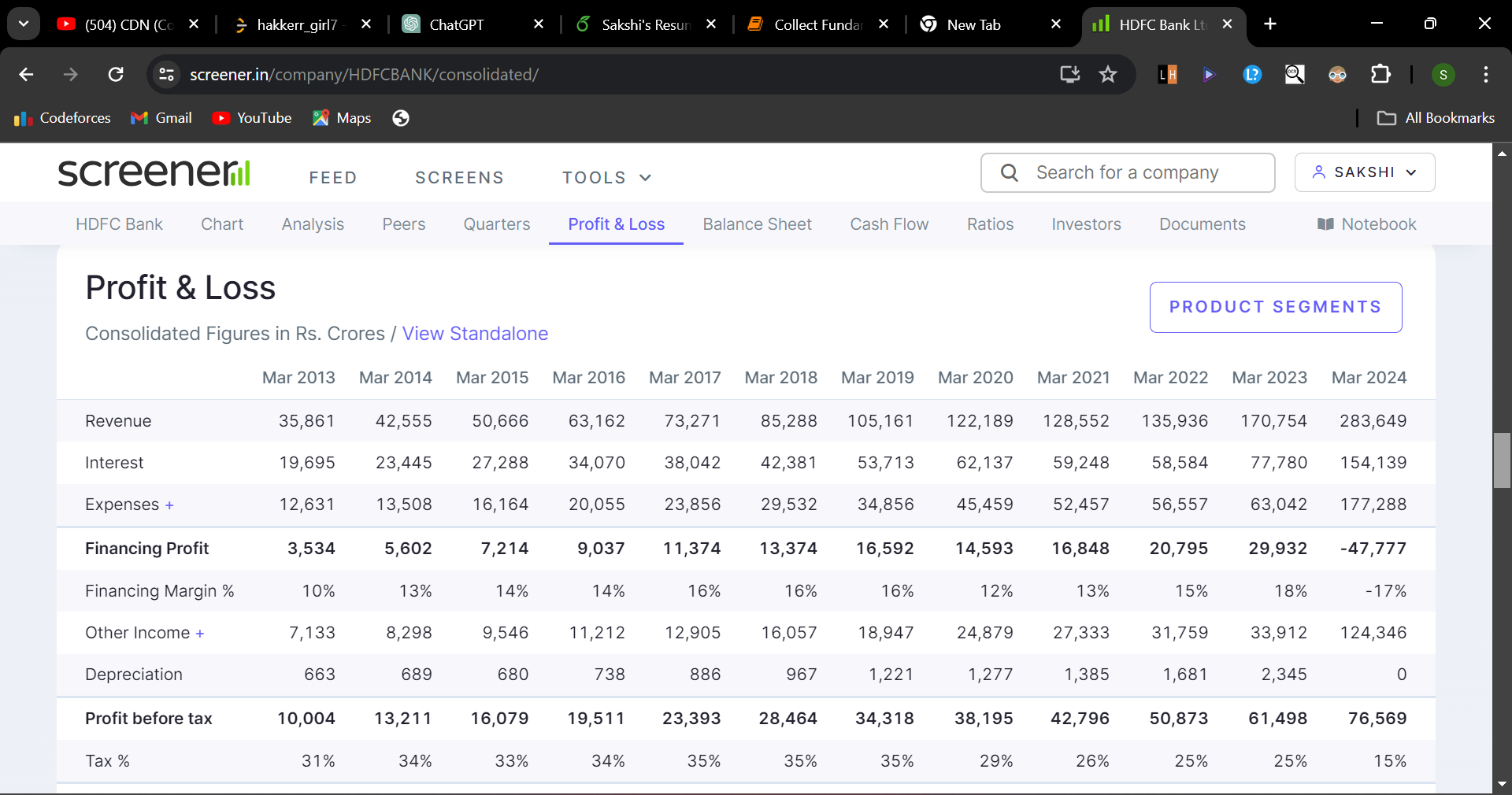
print("Data saved to 'stock\_data.json'")

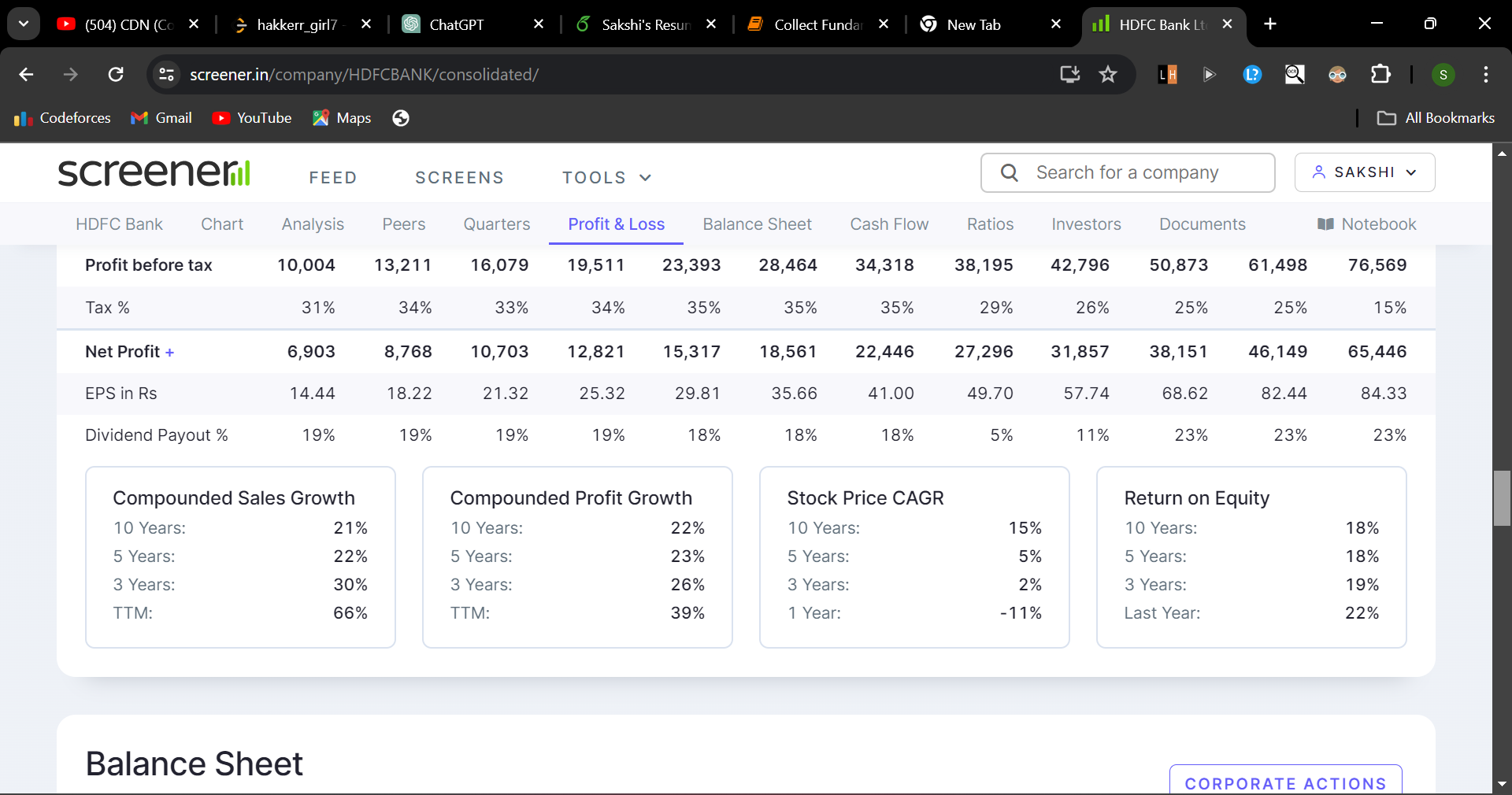
# Close the webdriver

driver.quit()

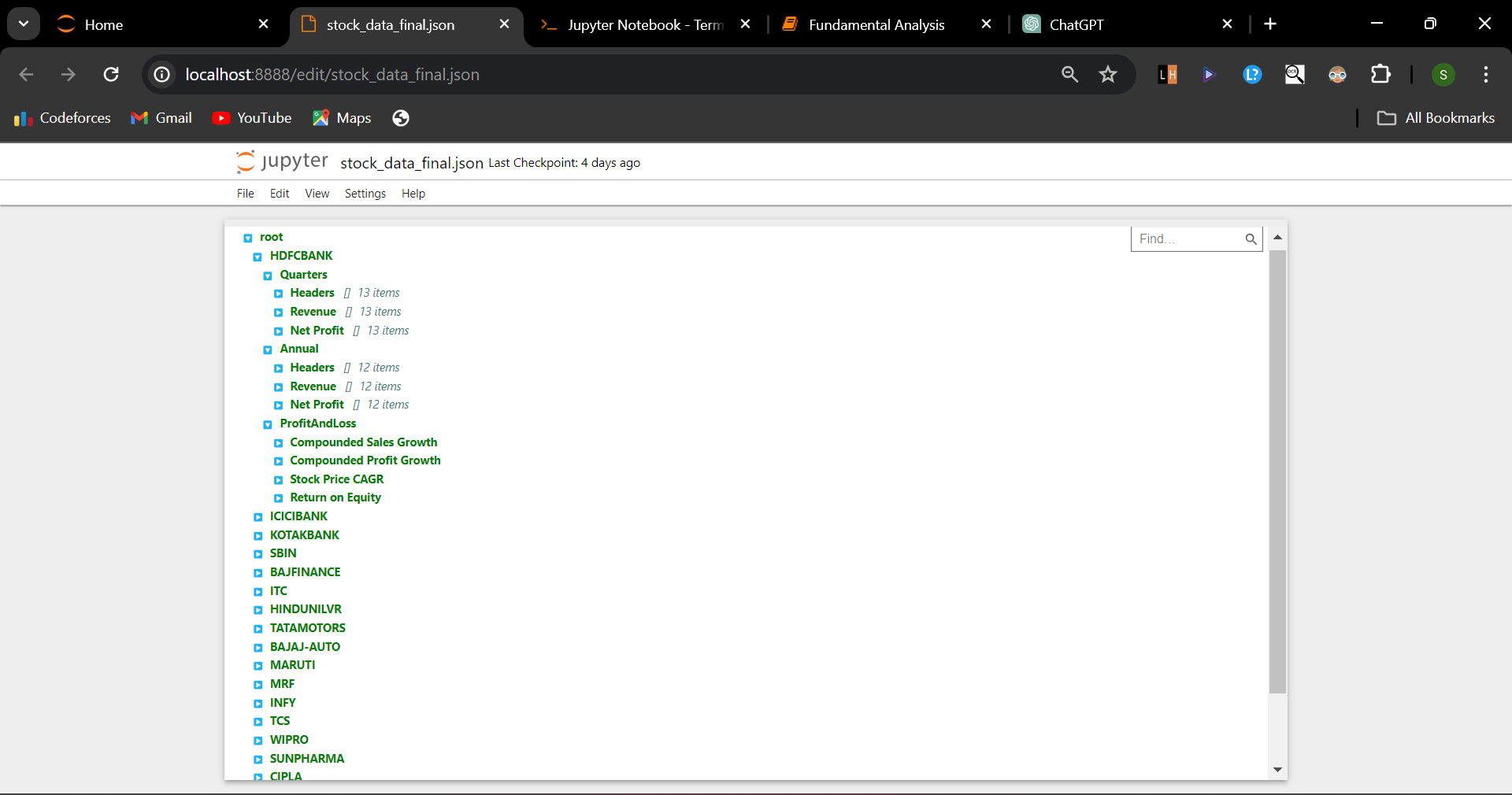


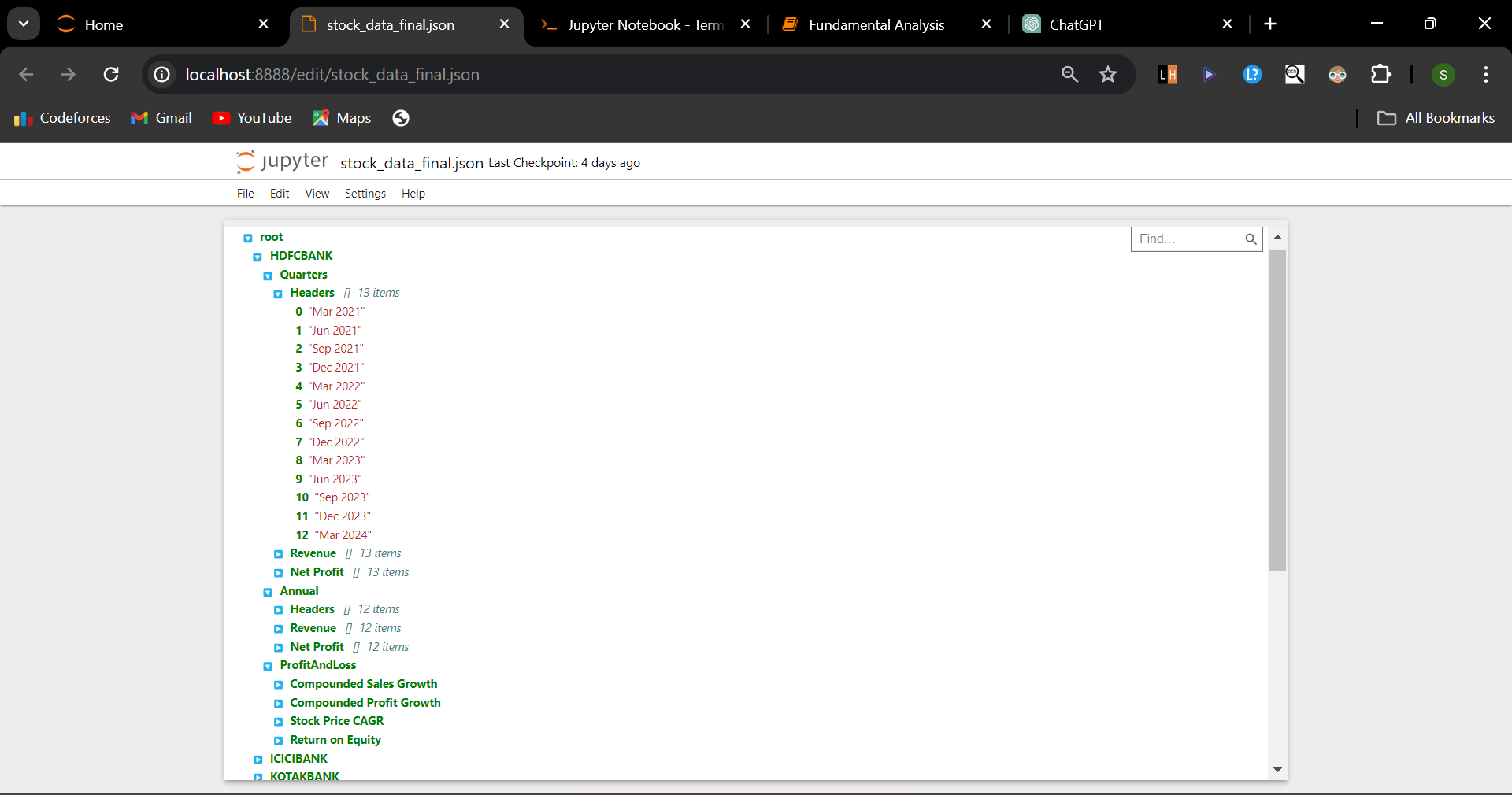


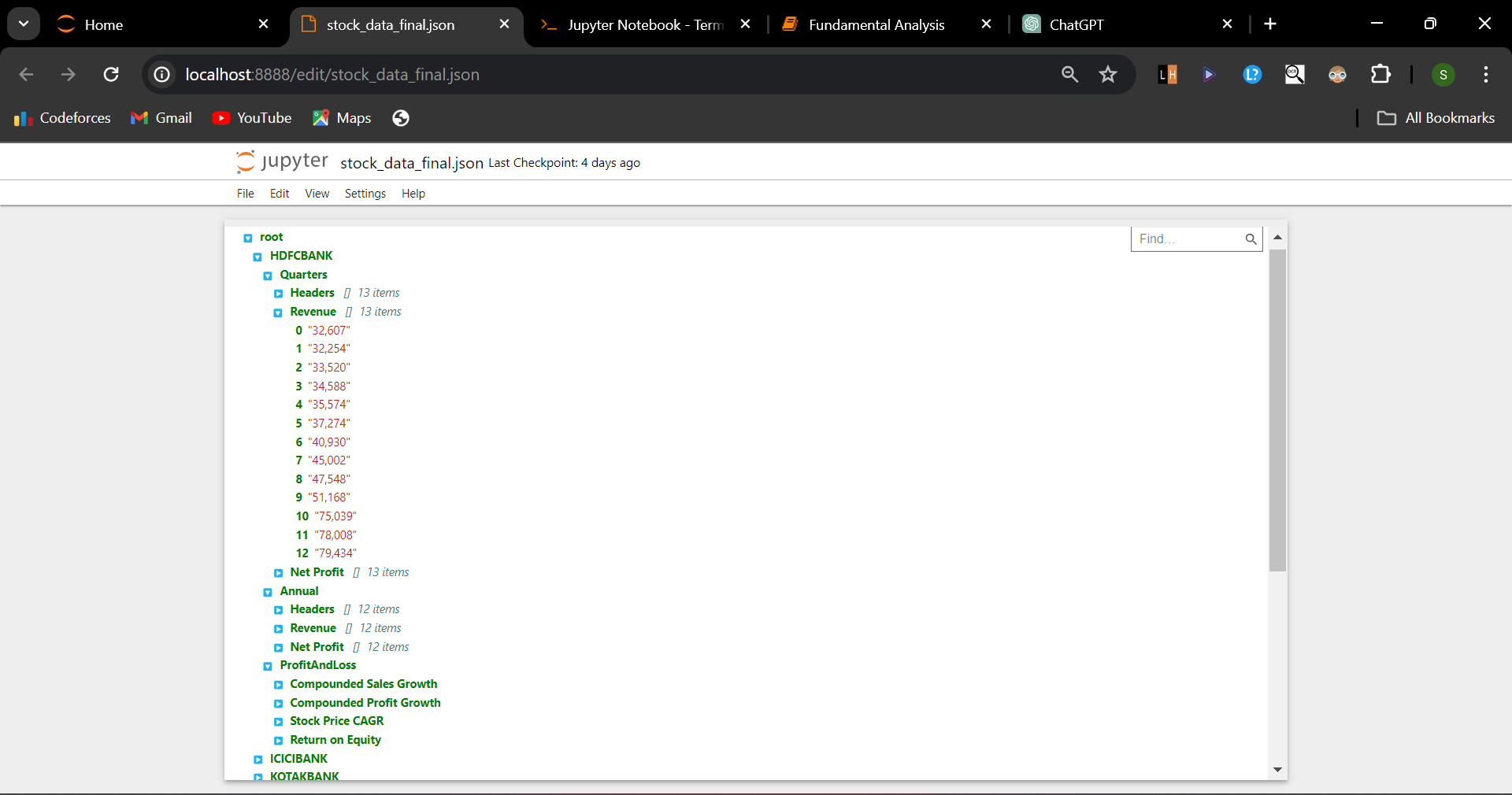


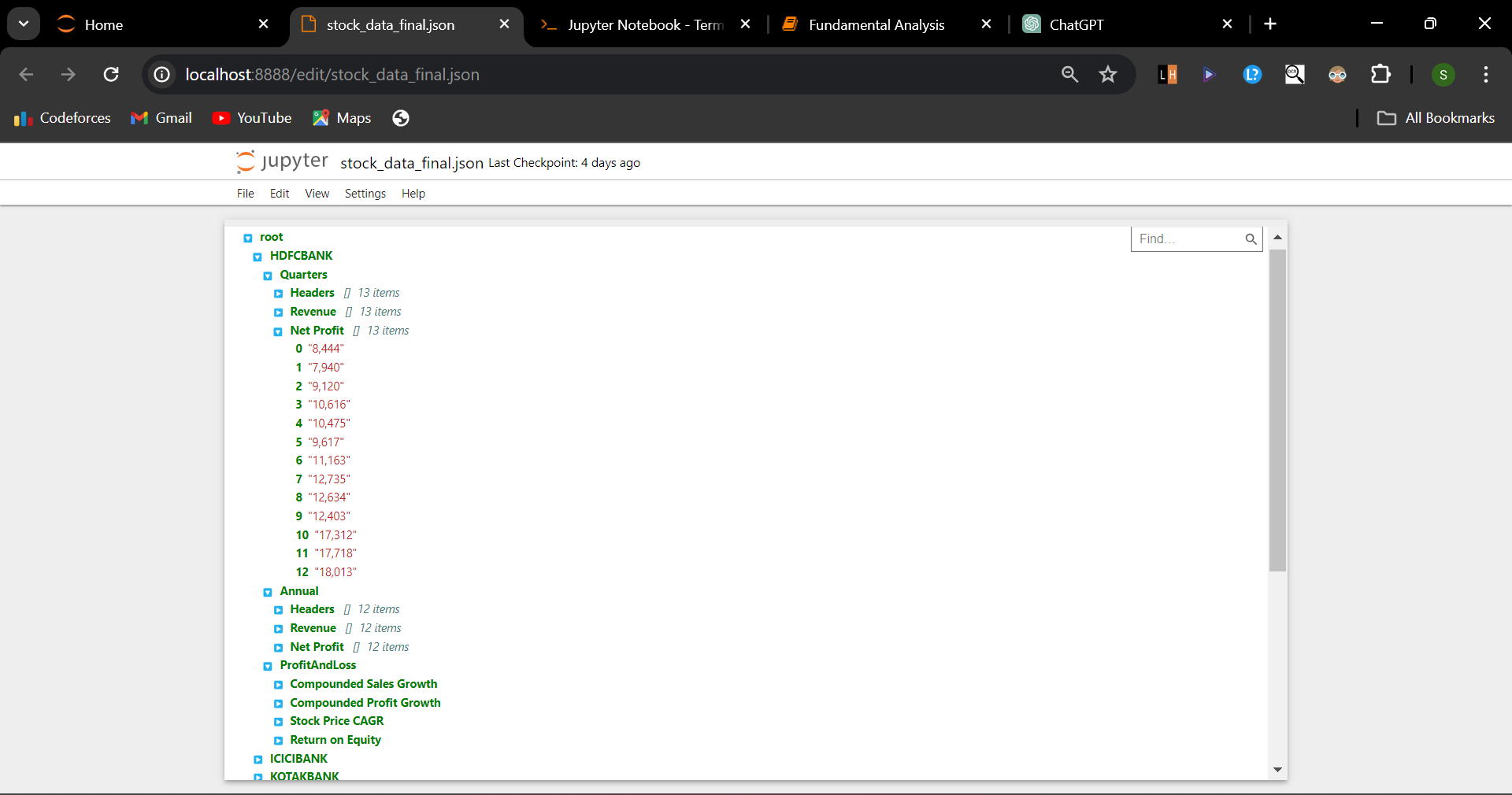


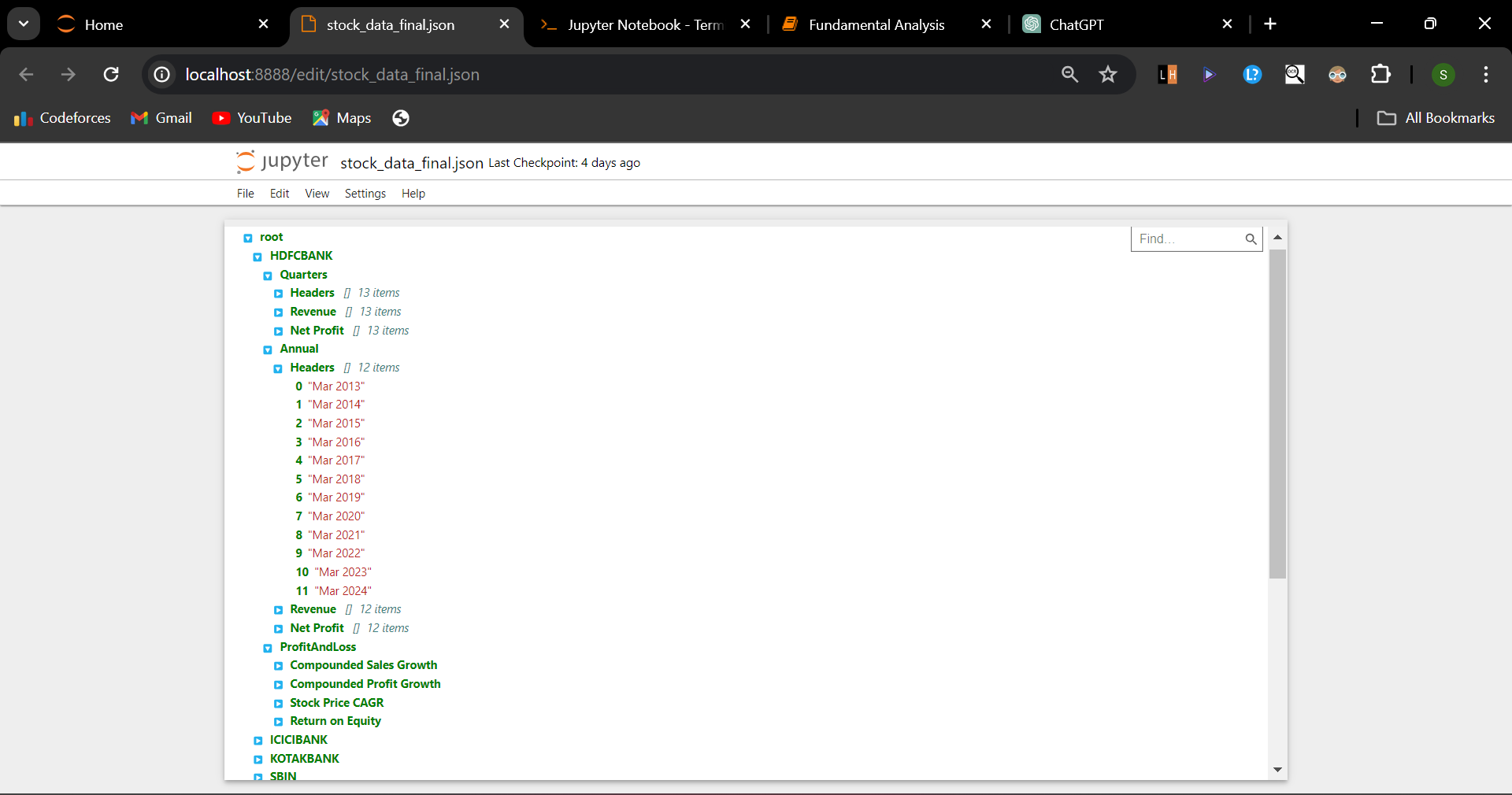


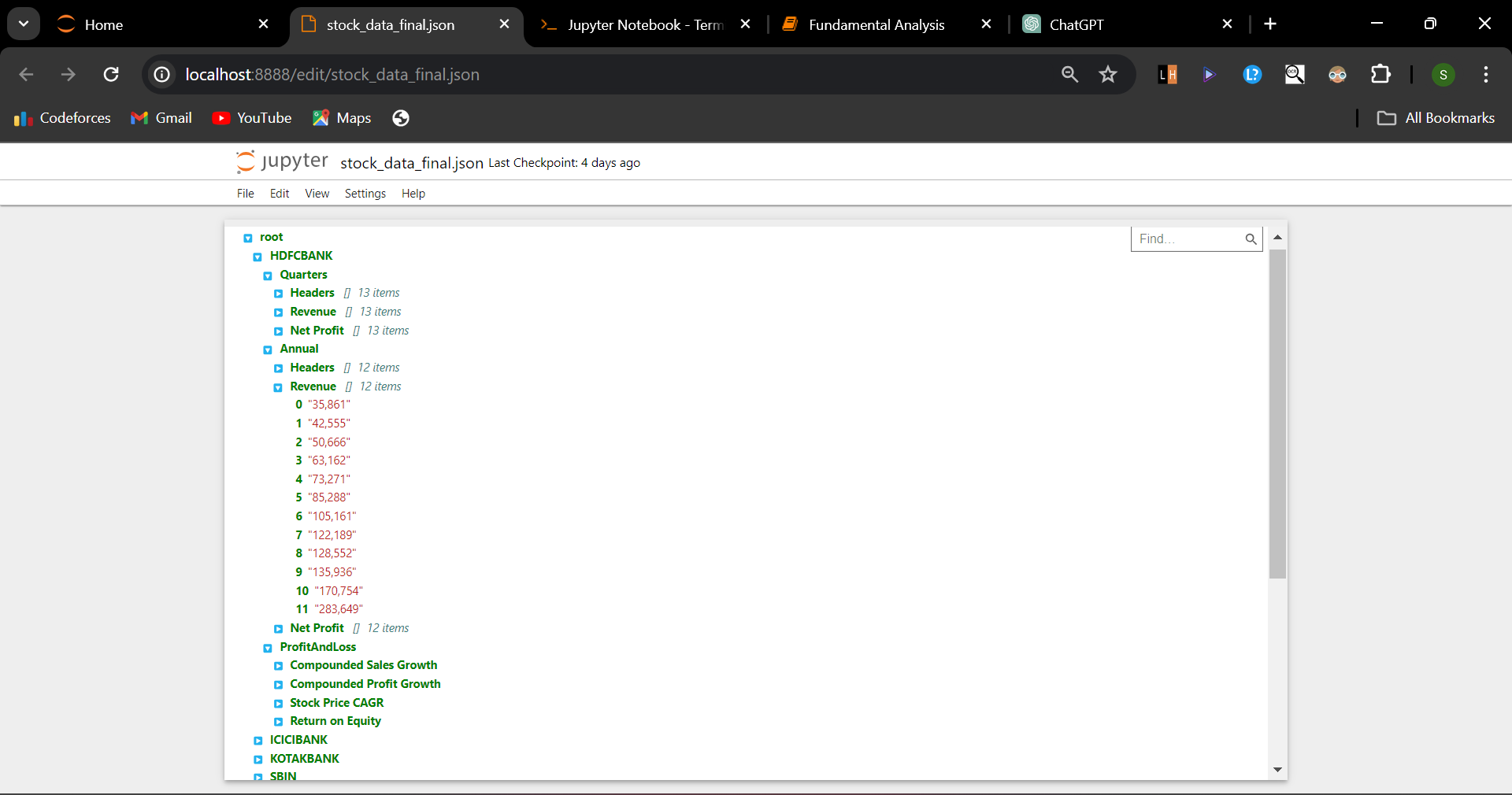


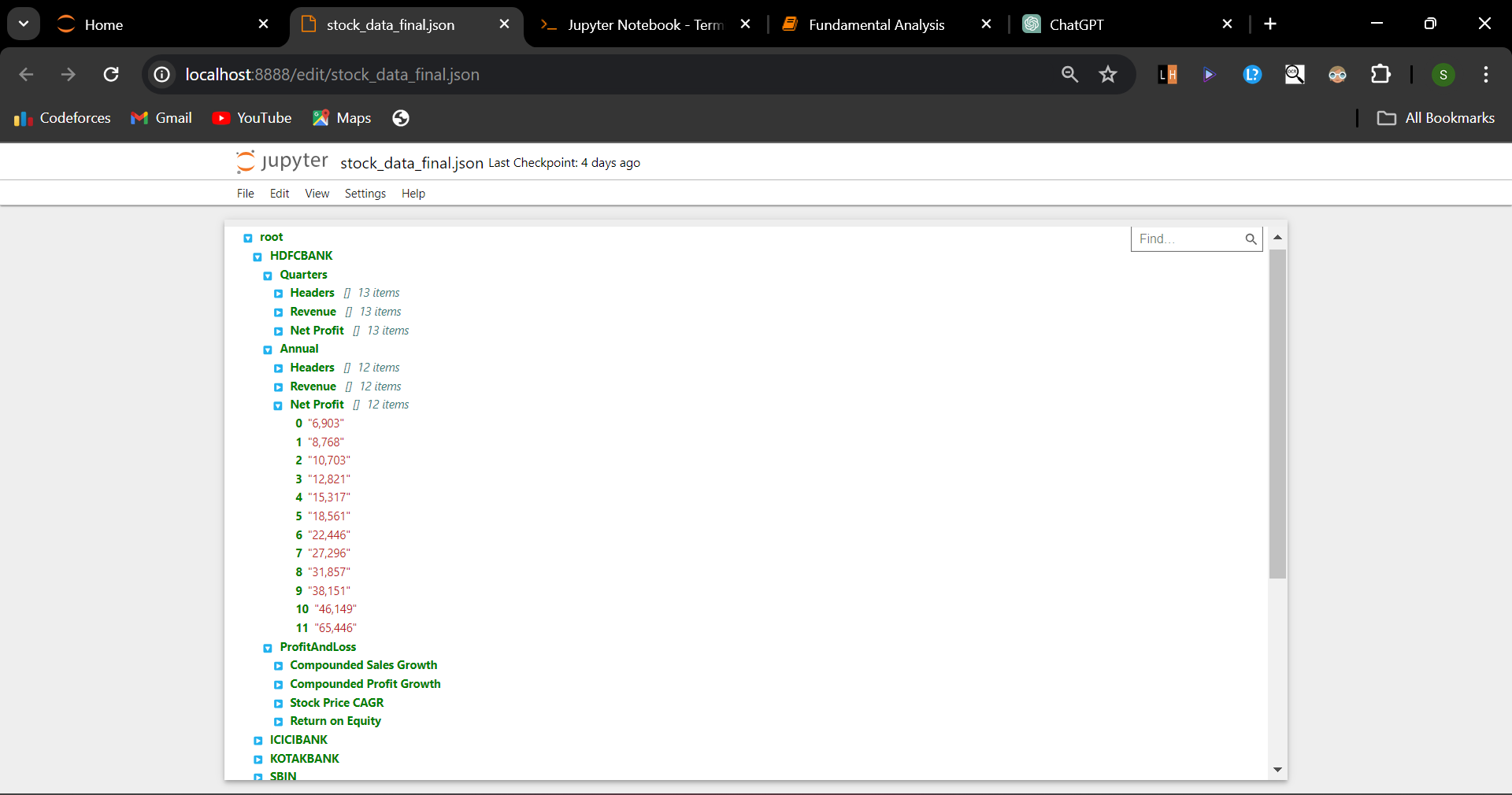


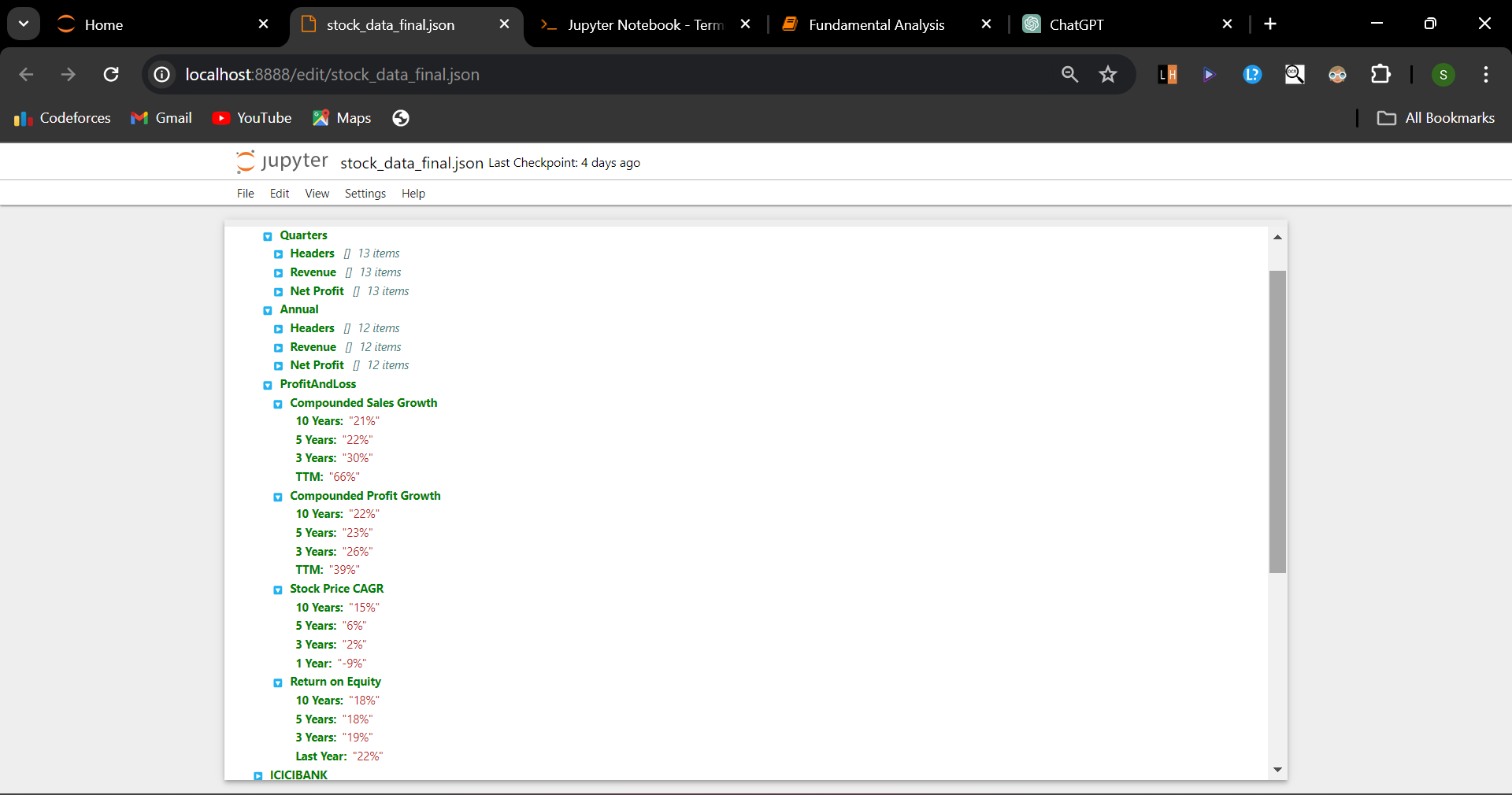












**2)Fundamental Analysis**

import json

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from statsmodels.tsa.arima.model import ARIMA

from statsmodels.tsa.statespace.sarimax import SARIMAX

from sklearn.linear\_model import LinearRegression

import warnings

import logging

warnings.filterwarnings('ignore')

logging.basicConfig(level=logging.ERROR)

# Load your JSON data

filename = 'stock\_data\_final.json'

with open(filename, 'r') as f:

data = json.load(f)

def forecast(stock, time\_period):

# Extract the revenue data for quarters and annual

timeline = data[stock][time\_period]['Headers']

revenue = data[stock][time\_period]['Revenue']

net\_profit = data[stock][time\_period]['Net Profit']

# Parse the string data to integers

revenue = [int(rev.replace(',', '')) for rev in revenue]

net\_profit = [int(rev.replace(',', '')) for rev in net\_profit]

n = 4 # number of future points to predict

forecast\_timeline = [timeline[-1]]

for i in range(n):

prev\_time = timeline[-1 - 3 + i]

month, year = prev\_time.split(' ')

upcoming\_year = int(year) + 1 if time\_period == 'Quarters' else int(year) + 4

next\_time = ' '.join([month, str(upcoming\_year)])

forecast\_timeline.append(next\_time)

# Forecasting for Revenue

model = ARIMA(revenue, order=(1,1,1))

model\_fit = model.fit()

forecast = model\_fit.get\_forecast(steps=n)

print(f"Forecasted {time\_period} Revenue for {stock}: ", forecast.predicted\_mean)

# Plot original data

plt.figure(figsize=(15, 5))

plt.plot(timeline, revenue, color='blue', label='Original data')

# Insert last original value to forcasted data

forecasted\_values = [revenue[-1]]

forecasted\_values.extend(forecast.predicted\_mean)

# Plot forecasted data with the last original point

plt.plot(forecast\_timeline, forecasted\_values, color='red', label='Forecasted data')

plt.title(f'{stock} - {time\_period} Revenue Forecast')

plt.xlabel('Time')

plt.ylabel('Revenue')

plt.legend()

plt.show()

# Forecasting for Net Profit

model = ARIMA(net\_profit, order=(1,1,1))

model\_fit = model.fit()

forecast = model\_fit.get\_forecast(steps=n)

print(f"Forecasted {time\_period} Net Profit for {stock}: ", forecast.predicted\_mean)

# Plot original data

plt.figure(figsize=(15, 5))

plt.plot(timeline, net\_profit, color='blue', label='Original data')

# Insert last original value to forcasted data

forecasted\_values = [net\_profit[-1]]

forecasted\_values.extend(forecast.predicted\_mean)

# Plot forecasted data with the last original point

plt.plot(forecast\_timeline, forecasted\_values, color='red', label='Forecasted data')

plt.title(f'{stock} - {time\_period} Net Profit Forecast')

plt.xlabel('Time')

plt.ylabel('Net Profit')

plt.legend()

plt.show()

for stock in data.keys():

print(f"Processing data for {stock}")

try:

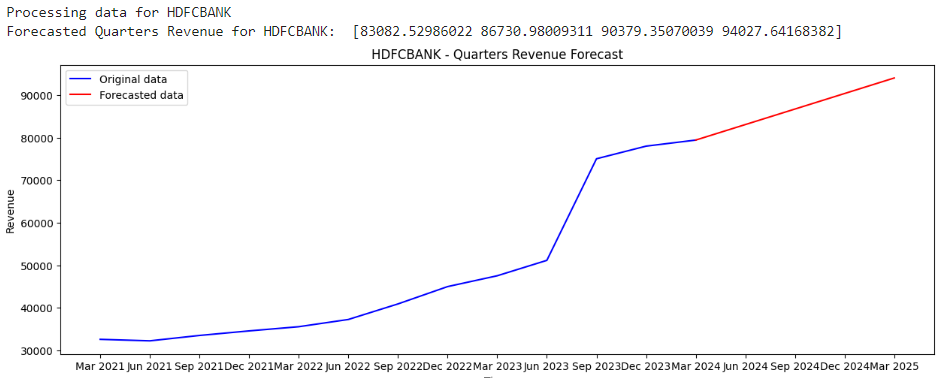
forecast(stock, 'Quarters')

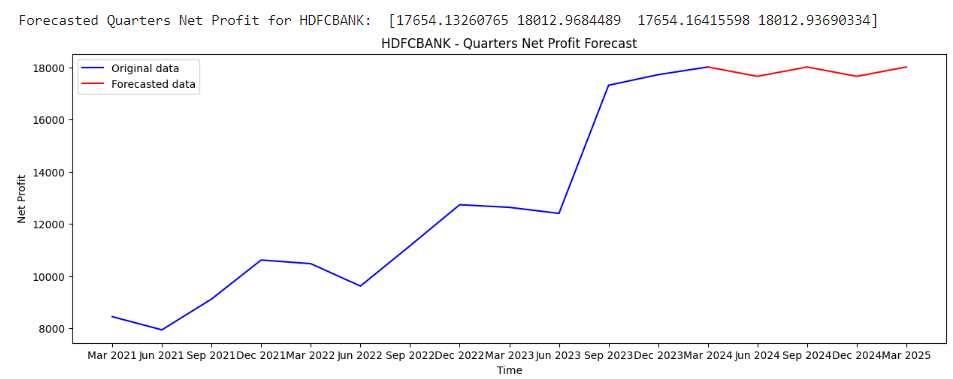
forecast(stock, 'Annual')

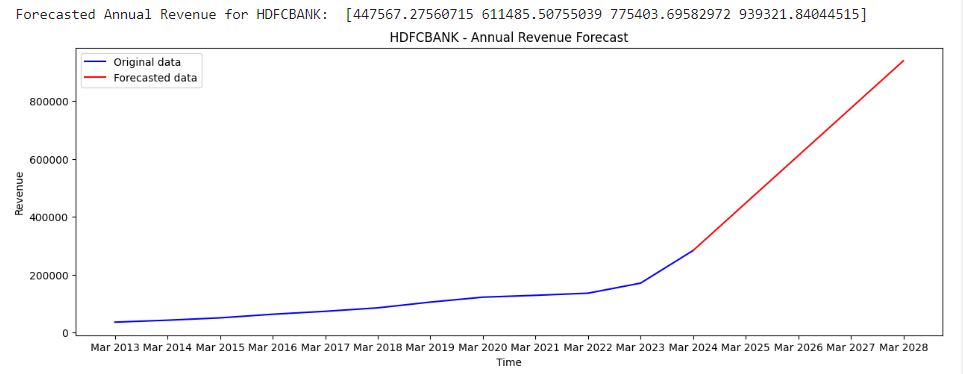
except:

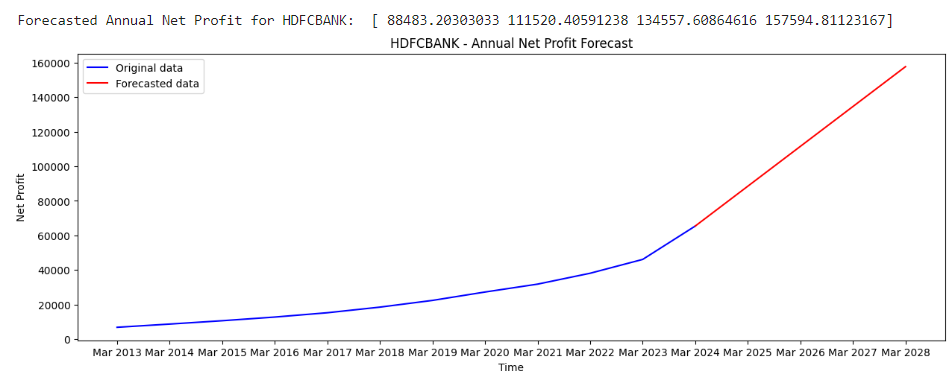
print(f"Could not process {stock} data completely")

break









# Prepare the data

quarters\_df = pd.DataFrame({

'ds': pd.to\_datetime(data['HDFCBANK']['Quarters']['Headers']),

'y': [int(rev.replace(',', '')) for rev in data['HDFCBANK']['Quarters']['Revenue']]

})

# Specify the order of the model (p, d, q)(P, D, Q, m)

model = SARIMAX(quarters\_df['y'], order=(1,1,1), seasonal\_order=(1,1,1,4))

# Fit the model

model\_fit = model.fit(disp=0)

# Make a future prediction

n = 3 # number of future points to predict

forecast = model\_fit.get\_forecast(steps=n)

# Create a time series for the forecasted data

forecasted\_time = pd.date\_range(start=quarters\_df['ds'].iloc[-1], periods=n+1, freq='Q')[1:] # start from the next quarter

# Create a series for the forecasted values

forecasted\_values = forecast.predicted\_mean

# Append the last actual value and time to the forecasted values and time

last\_actual\_time = quarters\_df['ds'].iloc[-1]

last\_actual\_value = quarters\_df['y'].iloc[-1]

forecasted\_time = pd.to\_datetime([last\_actual\_time] + list(forecasted\_time))

forecasted\_values = np.concatenate(([last\_actual\_value], forecasted\_values))

# Print the forecasted values

print('Forecasted Revenue: [', end = '')

for val in forecasted\_values[:-1]:

print(int(val), end=', ')

print(f'{int(forecasted\_values[-1])}]')

# Now, plot the graph again

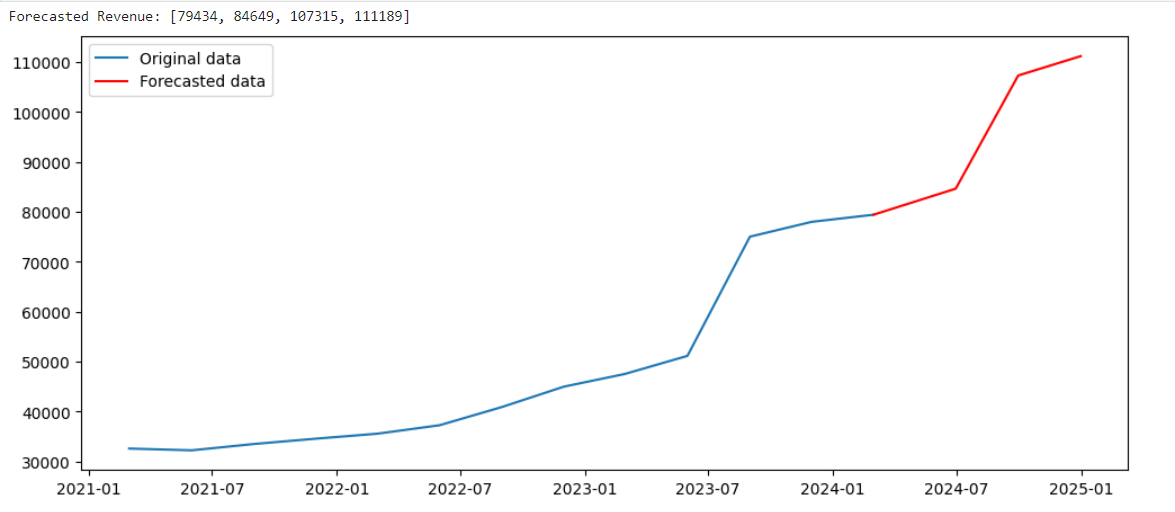
plt.figure(figsize=(12, 5))

plt.plot(quarters\_df['ds'], quarters\_df['y'], label='Original data')

plt.plot(forecasted\_time, forecasted\_values, color='red', label='Forecasted data')

plt.legend()

plt.show()



def forecast\_revenue(revenues, n, k):

model = LinearRegression()

for \_ in range(k):

X\_train = np.array(range(len(revenues[-n:]))).reshape(-1,1)

y\_train = np.array(revenues[-n:]).reshape(-1,1)

model.fit(X\_train, y\_train)

X\_test = np.array(len(revenues)).reshape(-1,1)

forecasted\_revenue = model.predict(X\_test)

# Append the forecasted revenue to the original list

revenues.append(int(forecasted\_revenue[0][0]))

return revenues

timeline = pd.to\_datetime(data['HDFCBANK']['Quarters']['Headers'])

revenues = [int(rev.replace(',', '')) for rev in data['HDFCBANK']['Quarters']['Revenue']]

n = len(revenues)

k = 3

forecasted\_revenues = forecast\_revenue(revenues, n, k)

print(forecasted\_revenues[n-1:])

# Extend timeline for forecasted quarters assuming each quarter is 3 months apart

forecast\_timeline = pd.date\_range(start=timeline[-1], periods=k+1, freq='Q')[1:]

timeline = timeline.append(forecast\_timeline)

# Plotting

plt.figure(figsize=(12,5))

plt.plot(timeline[:n], revenues[:n], label='Actual')

plt.plot(timeline[n-1:], forecasted\_revenues[n-1:], label='Forecasted') # n-1 to include the last actual point in forecast plot

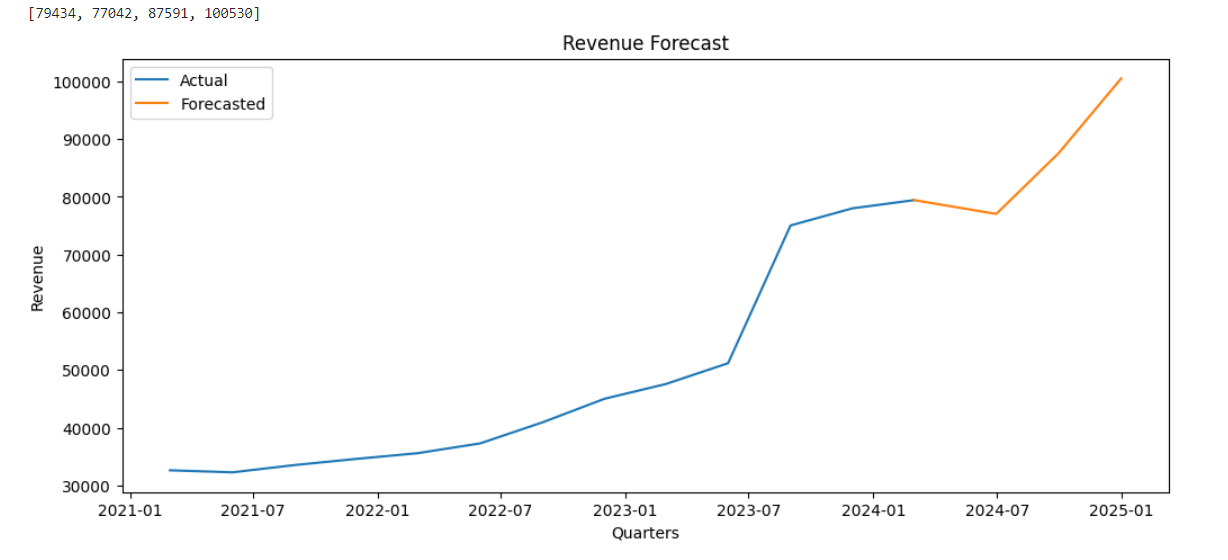
plt.xlabel('Quarters')

plt.ylabel('Revenue')

plt.title('Revenue Forecast')

plt.legend()

plt.show()



**3. Technical Analysis through RNN**

import numpy as np

import pandas as pd

from keras.models import Sequential

from keras.layers import LSTM, Dense, Dropout

from sklearn.preprocessing import MinMaxScaler

import matplotlib.pyplot as plt

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score

import math

import warnings

import logging

warnings.filterwarnings('ignore')

logging.basicConfig(level=logging.ERROR)

data = pd.read\_csv('Datasets/SCRIP/SBIN.csv')

data['Date'] = pd.to\_datetime(data['Date'])

startDate = '2018-01-01'

endDate = '2020-01-01'

data = data[(data['Date'] >= startDate) & (data['Date'] <= endDate)]

data['Date'] = (data['Date'] - data['Date'].min()).dt.days

data = data.dropna()

scaler = MinMaxScaler(feature\_range=(0,1))

scaled\_data = scaler.fit\_transform(data['Close'].values.reshape(-1,1))

def create\_dataset(dataset, look\_back=60):

X, Y = [], []

for i in range(look\_back, len(dataset)):

X.append(dataset[i-look\_back:i, 0])

Y.append(dataset[i, 0])

return np.array(X), np.array(Y)

X, Y = create\_dataset(scaled\_data)

train\_size = int(len(X) \* 0.7)

test\_size = len(X) - train\_size

X\_train, X\_test = X[0:train\_size,:], X[train\_size:len(X),:]

Y\_train, Y\_test = Y[0:train\_size], Y[train\_size:len(X)]

X\_train = np.reshape(X\_train, (X\_train.shape[0], X\_train.shape[1], 1))

X\_test = np.reshape(X\_test, (X\_test.shape[0], X\_test.shape[1], 1))

model = Sequential()

model.add(LSTM(units=50, return\_sequences=True, input\_shape=(X\_train.shape[1],1)))

model.add(Dropout(0.2))

model.add(LSTM(units=50, return\_sequences=False))

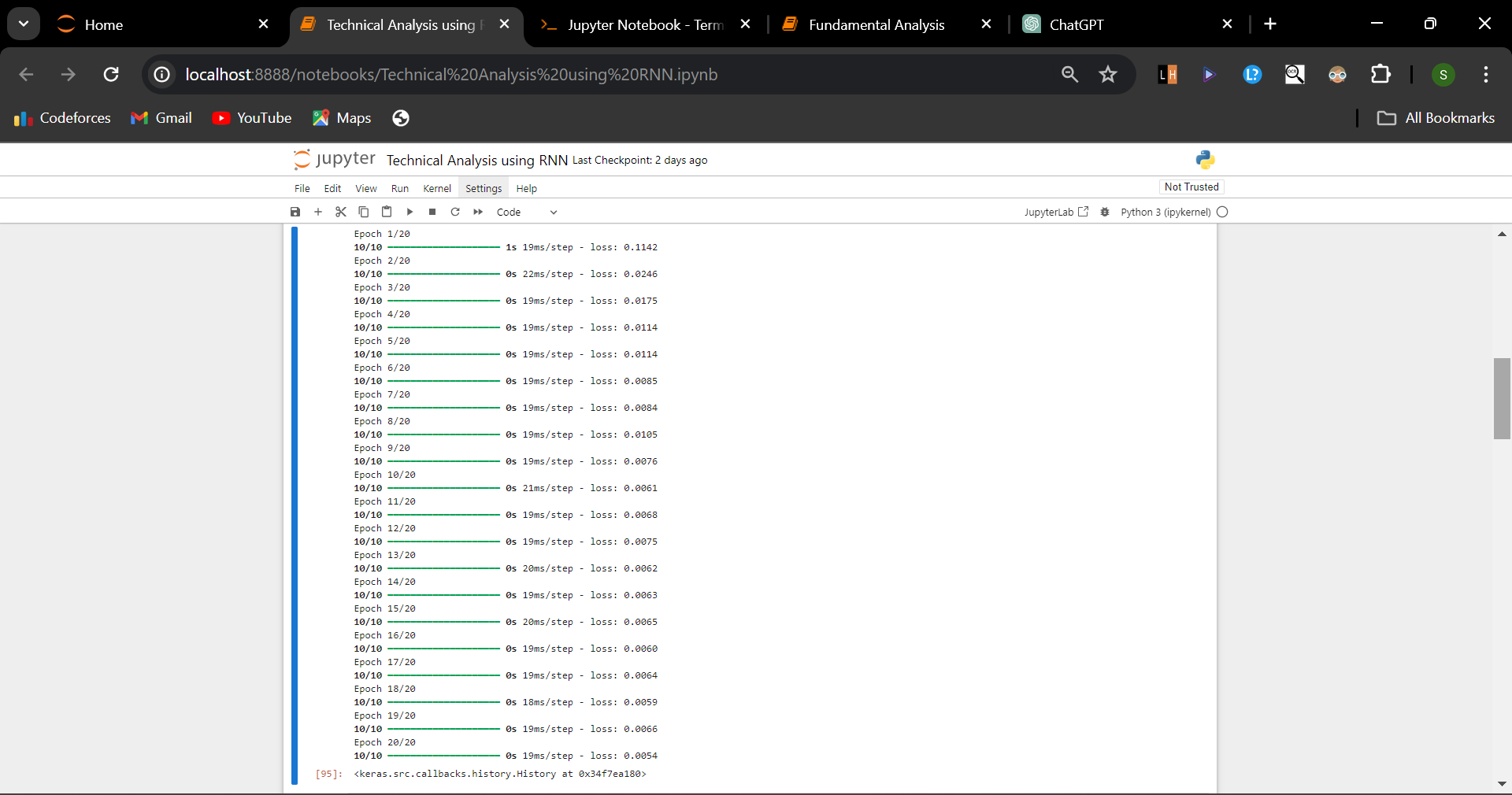
model.add(Dropout(0.2))

model.add(Dense(units=25))

model.add(Dense(units=1))

model.compile(optimizer='adam', loss='mean\_squared\_error')

model.fit(X\_train, Y\_train, epochs=20, batch\_size=32)



# Calculate the Mean Absolute Error

k = train\_size

mae = mean\_absolute\_error(overall\_actual[k:], overall\_predicted[k:])

print('Mean Absolute Error:', mae)

# Calculate the Mean Squared Error

mse = mean\_squared\_error(overall\_actual[k:], overall\_predicted[k:])

print('Mean Squared Error:', mse)

# Calculate the Root Mean Squared Error

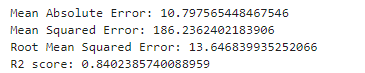
rmse = math.sqrt(mean\_squared\_error(overall\_actual[k:], overall\_predicted[k:]))

print('Root Mean Squared Error:', rmse)

# Calculate the R2 score

r2 = r2\_score(overall\_actual[k:], overall\_predicted[k:])

print(f'R2 score: {r2}')



plt.figure(figsize=(16,8))

plt.title('Model')

plt.xlabel('Date', fontsize=18)

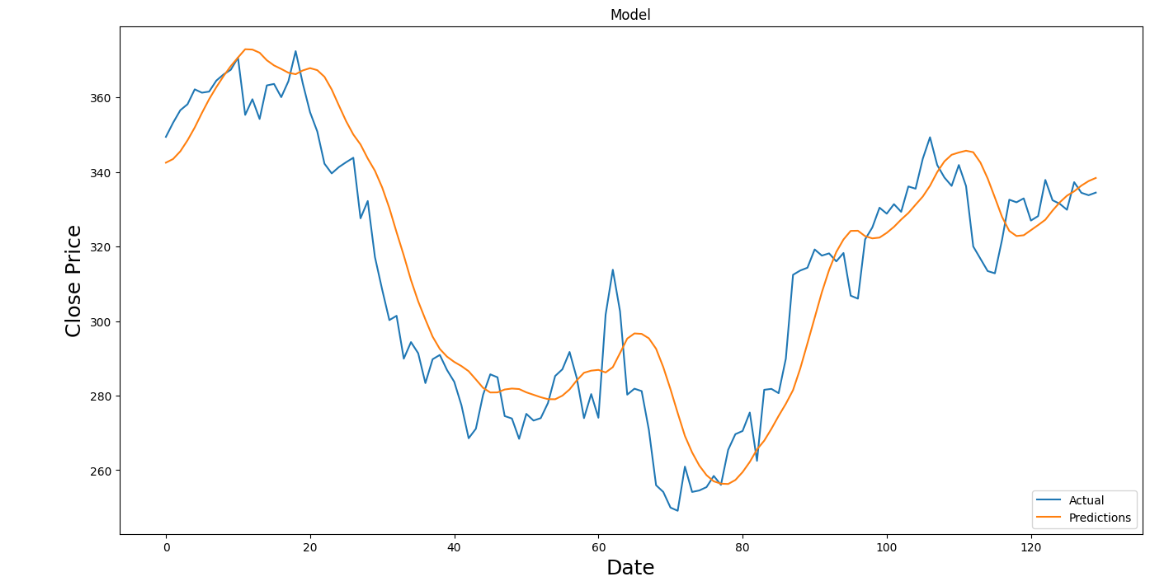
plt.ylabel('Close Price', fontsize=18)

plt.plot(actual)

plt.plot(predictions)

plt.legend(['Actual', 'Predictions'], loc='lower right')

plt.show()



plt.figure(figsize=(16,8))

plt.title('Model')

plt.xlabel('Date', fontsize=18)

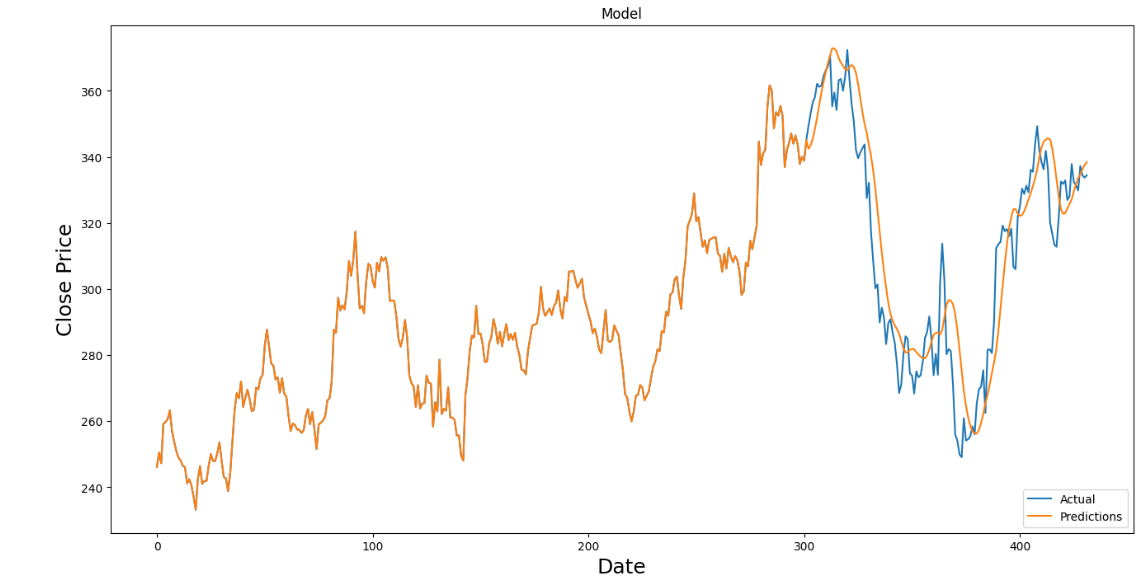
plt.ylabel('Close Price', fontsize=18)

plt.plot(overall\_actual)

plt.plot(overall\_predicted)

plt.legend(['Actual', 'Predictions'], loc='lower right')

plt.show()



**4. Support and Resistance through K Means**

import math

import random

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib.colors as mcolors

# Here cost function is simply the Manhattan distance (absolute diff) between the 2 points

def cost(p1, p2):

return np.abs(p1 - p2)

# Calculating total cost by summing up the costs in each cluster

def calculateTotalCost(clusters, means):

totalCost = 0

for i in range(len(clusters)):

for point in clusters[i]:

totalCost += cost(means[i], point) \*\*2

return totalCost \*\* 0.5

def kMeans(data, k, showClusters = False, maxIterations = 100, markerSize = 1):

n = len(data)

minVal, maxVal = np.min(data), np.max(data)

assignedClusterIds = np.zeros(n, dtype=int)

bestClusters = []

bestMeans = []

bestAssignedClusterIds = []

minTotalCost = n \* (maxVal - minVal) # This is the maximum possible cost (assume mean is at minVal & all the n points at maxVal)

# This outermost loop would really be useful when we randomize means instead of uniformly distributing

# So by default it will run only once

for iteration in range(maxIterations):

clusters = []

# Initialize means

means = [np.random.choice(data) for \_ in range(k)]

# Repeat the process of readjusting means 20 times

maxCycles = 50

# Cluster data by minimizing cost

for cycle in range(maxCycles):

clusters = [[] for \_ in range(k)]

for index, value in enumerate(data):

# Max diff between any 2 points

minCost = maxVal - minVal

minClusterId = 0

# Calculate the distance of the point from each mean

for i in range(k):

curCost = cost(means[i], value)

# Choose the cluster with closest mean

if (curCost < minCost):

minCost = curCost

minClusterId = i

# Assign the point to the cluster with the nearest mean value

clusters[minClusterId].append(value)

assignedClusterIds[index] = minClusterId

# Readjust mean values

for i in range(k):

numElements = len(clusters[i])

if numElements > 0:

means[i] = sum(clusters[i]) / numElements

else:

means[i] = np.random.choice(data)

# Find total cost for current clustering

totalCost = calculateTotalCost(clusters, means)

# Choose the overall best clustering

if (totalCost < minTotalCost):

minTotalCost = totalCost

bestClusters = clusters

bestMeans = means

bestAssignedClusterIds = assignedClusterIds

if (showClusters):

showClustering(range(n), data, assignedClusterIds, means, markerSize)

return bestClusters, bestMeans, bestAssignedClusterIds

# Plot data by color coding each cluster differently

def showClustering(x, y, assignedClusterIds, SNR, markerSize = 1):

numColors = 10

HSV\_tuples = [(i/numColors, 1.0, 1.0) for i in range(numColors)]

RGB\_tuples = list(map(lambda x: mcolors.hsv\_to\_rgb(x), HSV\_tuples))

plt.figure(figsize=(10,6))

for i in range(len(x)):

plt.scatter(x[i], y[i], color = RGB\_tuples[assignedClusterIds[i] % numColors], s = markerSize)

# Plot horizontal lines at the support & resistances

for snrValue in SNR:

plt.axhline(y=snrValue, color='black', linestyle='--')

plt.annotate(f'{snrValue}', xy=(1, snrValue), xycoords=('axes fraction', 'data'), textcoords='offset points', xytext=(5, 0), ha='left', va='center', color='black')

plt.show()

points = np.array([1,2,4,5,22,24,26,28,50,60,30,20])

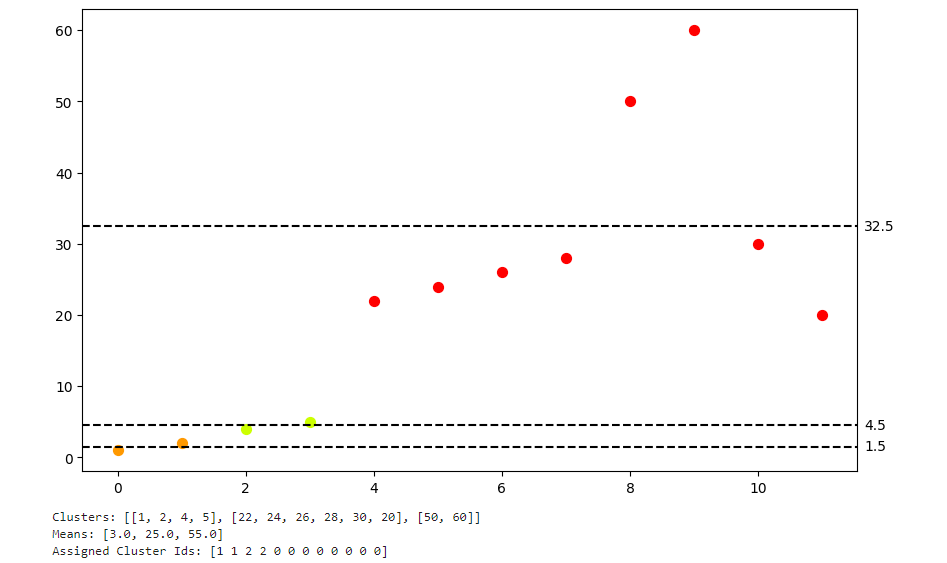
k = 3

clusters, means, assignedClusterIds = kMeans(points, k, True, 10, 50)

print(f'Clusters: {clusters}')

print(f'Means: {means}')

print(f'Assigned Cluster Ids: {assignedClusterIds}')



def autoKMeans(data, maxClusters = 10, showElbowCurve = False, showBestClustering = False):

totalCosts = [float('inf')] # for 0 clusters, cost is infinity

for k in range(1, min(len(data), maxClusters) + 1):

clusters, means, \_assignedClusterIds = kMeans(data, k)

totalCosts.append(calculateTotalCost(clusters, means))

# Find the elbow point - value of k for most optimal clustering (highest reduction in cost on increasing total clusters by 1)

bestK = 1

maxAngleDiff = 0

for k in range(2, len(totalCosts) - 1):

x1, y1 = 10 \* (k - 1), totalCosts[k - 1]

x2, y2 = 10 \* k, totalCosts[k]

x3, y3 = 10 \* (k + 1), totalCosts[k + 1]

prevAngle = math.degrees(math.atan(abs((y2 - y1) / (x2 - x1))))

curAngle = math.degrees(math.atan(abs((y3 - y2) / (x3 - x2))))

angleDiff = prevAngle - curAngle

# print(prevAngle, curAngle, angleDiff)

if (angleDiff > maxAngleDiff):

maxAngleDiff = angleDiff

bestK = k

# Plotting the totalCosts for each k

if (showElbowCurve):

plt.figure(figsize=(10,6))

plt.plot(range(1, len(totalCosts)), totalCosts[1:], marker='o')

plt.plot(bestK, totalCosts[bestK], marker='o', color='orange')

plt.annotate('Elbow point', (bestK, totalCosts[bestK]), textcoords="offset points", xytext=(40,10), ha='center')

plt.xlabel('Number of clusters (k)')

plt.ylabel('Total Cost')

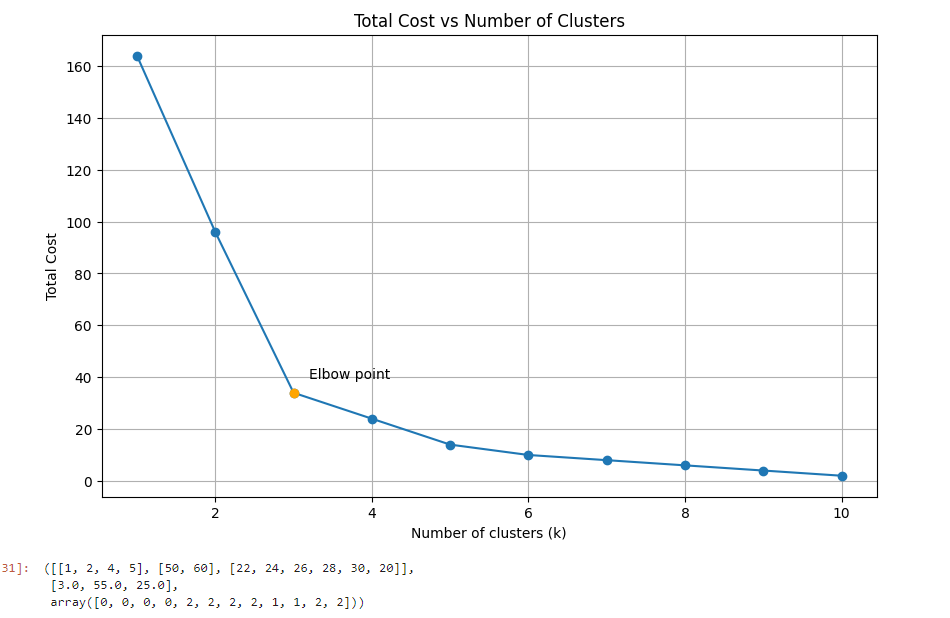
plt.title('Total Cost vs Number of Clusters')

plt.grid(True)

plt.show()

return kMeans(data, bestK, showBestClustering)

autoKMeans(points, 10, True)



# Main

# Load the stock prices of ITC using pandas

dataframe = pd.read\_csv('Datasets/SCRIP/ITC.csv')

# Select the relevant attributes and rename them

data = dataframe[['Date', 'Close']].rename(columns = {'Close': 'Price'})

# Convert the price from string to float convert coerce mode converts non-numeric strings to NaN

data['Price'] = pd.to\_numeric(data['Price'], errors = 'coerce')

# Convert dates to datetime format

data['Date'] = pd.to\_datetime(data['Date'])

# Slice the time range

startDate = '2007-01-01'

endDate = '2024-01-01'

data = data[(data['Date'] >= startDate) & (data['Date'] <= endDate)]

# Subtract the initial date from all dates and convert it into days (we only need these indexes to plot)

data['Date'] = (data['Date'] - data['Date'].min()).dt.days

# Drop rows that have NaN values

data = data.dropna()

# Convert single columns into numpy arrays

prices = data['Price'].values

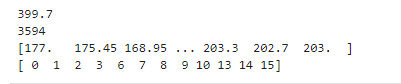
dates = data['Date'].values

print(max(prices))

print(len(prices))

print(prices)

print(dates[0:12]) # Notice that Saturday-Sunday markets are closed



# This is an expensive process. Will take some time depending on your machine.

clusters, means, assignedClusterIds = autoKMeans(prices)

# Calculating all the Support & Resistance lines (SNR)

supports = []

resistances = []

for cluster in clusters:

supports.append(math.floor(min(cluster)))

resistances.append(math.floor(max(cluster)))

supports.sort()

resistances.sort()

SNR = [supports[0]] # bottom line

# Middle lines

for i in range(len(clusters) - 1):

SNR.append((resistances[i] + supports[i + 1]) / 2)

SNR.append(resistances[len(clusters) - 1]) # top line

showClustering(dates, prices, assignedClusterIds, SNR)

