Capstone Project 2: Stock price forecasting

Abstract

We present the results of the Capstone 2 Project, focusing on using machine learning algorithms for learning stock price distributions and forecasting. The project focuses on two distinct stocks: Coca-Cola (KO) and Tesla (TSLA), representing different spectrums of the stock market: the former is a solid company that has demonstrated consistent growth over six decades, whereas the latter is a younger company characterized by a high volatility. The documentation is systematically structured into four integral sections, each detailing a crucial phase of the machine learning workflow applied to stock price prediction.

1. Data Wrangling: The project begins with the process of collecting historical stock prices and company earnings data. The objective in this phase is to prepare a clean and comprehensive dataset that forms the bedrock for all subsequent analyses and modeling efforts. Emphasis is placed on accuracy and relevancy of the data to ensure robustness in the predictive models.

2. Exploratory Data Analysis (EDA): This phase focuses on identifying patterns, trends, and volatilities in the stocks of KO and TSLA, along with examining correlations between various features. The insights gained from this EDA are crucial for understanding the behavior of these stocks and for informing the direction of feature engineering and model development.

3. Feature Engineering: The original dataset is augmented with calculated financial indicators derived from stock prices and earnings per share, designed to enrich the dataset and enhance the predictive models' capabilities. Additionally, this section includes critical steps such as rescaling of numerical values and strategic splitting of the dataset into training and testing sets, taking into account the chronological nature of stock price data.

4. Model Development: The final phase of the project is centered around the development and evaluation of two advanced machine learning models: Random Forests and LSTM (Long Short-Term Memory) Neural Networks. These models are tailored to predict the closing prices of stocks, and their performance is thoroughly evaluated. The section also explores how the size of the test set influences the effectiveness of these models, thereby providing insights into their applicability in real-world stock price forecasting scenarios.

We end this report with a conclusions section summarizing the milestones reached, the limitations of our models and potential areas of improvement.

Section 1: Data wrangling

In this section we focus on the data wrangling part of the project. We will collect data about historical stock prices and company earnings using Yahoo Finance and Alpha Vantage. The aim is to create a dataframe containing all the relevant information to train our models for stock price forecasting in future steps. We will focus on two stocks: Coca-Cola, a stock with a long history of solid growth, and Tesla, a newer and more volatile one.

1. Coca-Cola stock

In [1]:
```
import sys
import requests
import json
import numpy as np
import pandas as pd
```

Data collection

We shall import the historical stock price data from Yahoo Finance. Let's start with the Coca-Cola stock, whose ticker is `KO`. Obviously the start date of the collected data will vary for the different stocks. Regarding the end date, we shall fix it as 31st December 2022.

In [2]:
```
import yfinance as yf

# Define the ticker symbol and end date
ticker = "KO"
end_date = "2022-12-31"

# Fetch the historical data using yfinance with start=None
data = yf.download(ticker, start=None, end=end_date)
```

```
[**********100%**********]  1 of 1 completed
```

In [3]:
```
# Make sure we have a dataframe type
type(data)
```

Out[3]:
```
pandas.core.frame.DataFrame
```

Now we shall print out the head of our dataset `data`:

In [4]:
```
# Print out the head of the dataframe
data.head()
```

Out[4]:

Date	Open	High	Low	Close	Adj Close	Volume
1962-01-02	0.263021	0.270182	0.263021	0.263021	0.048913	806400
1962-01-03	0.259115	0.259115	0.253255	0.257161	0.047823	1574400
1962-01-04	0.257813	0.261068	0.257813	0.259115	0.048187	844800
1962-01-05	0.259115	0.262370	0.252604	0.253255	0.047097	1420800
1962-01-08	0.251302	0.251302	0.245768	0.250651	0.046613	2035200

Let's check for missing values and NaN values:

In [5]:
```
# Check for missing values in the DataFrame
missing_values = data.isnull().sum()

# Print the missing values count
print(missing_values)
```

```
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

In [6]:
```
# Check for NaN values in the DataFrame
nan_values = data.isna().sum()

# Print the NaN values count
print(nan_values)
```

```
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

We have no missing or NaN values in this original dataset for stock prices. However, in future steps we will find NaN values for certain rows when computing financial indicators, and these will need to be dropped.

With this information about stock prices we can compute financial indicators, which will be used as features. However, we would like to have additional features that are not directly related to the stock price, such as the earnings per share (EPS), which enables us to compute the P/E ratio --a commonly used metric to value a publicly traded company. Here we shall import this information from the Alpha Vantage API.

In [8]:
```python
# Obtaining earnings per share (EPS)

import requests
import pandas as pd

API_KEY = "AKOOJR5JCFPFDX02"

BASE_URL = "https://www.alphavantage.co/query?"

def earnings_history_api(api_key, symbol):
    assert symbol is not None
    symbol = symbol.strip().upper()

    url = f"{BASE_URL}function=EARNINGS&symbol={symbol}&apikey={api_key}"

    response = requests.get(url)
    data = response.json()

    if 'quarterlyEarnings' in data:
        earnings_data = data['quarterlyEarnings']
        df = pd.DataFrame(earnings_data)
        return df
    else:
        print("No earnings data found for the specified symbol.")
        return None

df_earnings = earnings_history_api(API_KEY, "KO")

# Print dataframe with EPS
df_earnings
```

Out[8]:

	fiscalDateEnding	reportedDate	reportedEPS	estimatedEPS	surprise	surprisePercenta
0	2023-03-31	2023-04-24	0.68	0.65	0.03	4.61
1	2022-12-31	2023-02-14	0.45	0.45	0	
2	2022-09-30	2022-10-25	0.69	0.64	0.05	7.81
3	2022-06-30	2022-07-26	0.7	0.67	0.03	4.47
4	2022-03-31	2022-04-25	0.64	0.58	0.06	10.34
...	...	...	...	...	...	
104	1997-03-31	1997-04-14	0.2	0.2	0	
105	1996-12-31	1997-01-31	0.16	0.16	0	
106	1996-09-30	1996-10-15	0.2	0.19	0.01	5.26
107	1996-06-30	1996-07-15	0.21	0.21	0	
108	1996-03-31	1996-04-17	0.14	0.14	0	

109 rows × 6 columns

We see that the first row in this dataframe dates from 1996-03-01, which is way more recent that the first data within the historical stock prices dataframe. Still, the EPS dataframe provides us with 26 years of data, which in combination with the stock prices data restricted to this period should be sufficient to effectively train our models.

In the dataframes of Alpha Vantage missing values appear as "None". We have to check that there are no None values in `df_earnings`:

In [9]:
```
(df_earnings == 'None').sum().sum()
```

Out[9]:
```
0
```

Data organization and data cleaning

We now would like to incorporate the `reportedEPS` column in our dataframe for stock prices `data`, in such a way that every row is assigned the reported EPS that corresponds to the row's date. In order to do this, first we note that the data type of the `reportedDate` column is different from the data type of the index in our main dataframe: the former is an Object type whereas the latter is a datetime64 type.

In [10]:
```
print(df_earnings['reportedDate'].dtype)
```

```
object
```

In [11]:
```
print(data.index.dtype)
```

```
datetime64[ns]
```

So first we need to convert the data type of the `reportedDate` column to a datetime64 type.

In [12]:
```
# Convert the reportedDate column to a datetime64 type
df_earnings['reportedDate'] = pd.to_datetime(df_earnings['reportedDate'])

# Check data type
print(df_earnings['reportedDate'].dtype)
```

```
datetime64[ns]
```

Then we promote the `reportedDate` column to being the index:

In [13]:
```
# Set the `reportedDate' column as the index
df_earnings = df_earnings.set_index('reportedDate')

# Print the dataframe and check
df_earnings.head()
```

Out[13]:

reportedDate	fiscalDateEnding	reportedEPS	estimatedEPS	surprise	surprisePercentage
2023-04-24	2023-03-31	0.68	0.65	0.03	4.6154
2023-02-14	2022-12-31	0.45	0.45	0	0
2022-10-25	2022-09-30	0.69	0.64	0.05	7.8125
2022-07-26	2022-06-30	0.7	0.67	0.03	4.4776
2022-04-25	2022-03-31	0.64	0.58	0.06	10.3448

Let's sort the dataframe in ascending order:

In [14]:
```
df_earnings.sort_index(ascending=True, inplace=True)
```

In [15]:
```
df_earnings.head()
```

Out[15]:

reportedDate	fiscalDateEnding	reportedEPS	estimatedEPS	surprise	surprisePercentage
1996-04-17	1996-03-31	0.14	0.14	0	0
1996-07-15	1996-06-30	0.21	0.21	0	0
1996-10-15	1996-09-30	0.2	0.19	0.01	5.2632
1997-01-31	1996-12-31	0.16	0.16	0	0
1997-04-14	1997-03-31	0.2	0.2	0	0

We are only interested in adding the `reportedEPS` column, so we shall drop the others.

In [16]:
```
df_earnings.drop(["fiscalDateEnding", "estimatedEPS", "surprise", "surprisePercenta
```

Let's also change the name of `reportedEPS` column to `Reported EPS`:

In [17]:
```
df_earnings.rename(columns={'reportedEPS': 'Reported EPS'}, inplace=True)
```

Finally we merge the two dataframes on their index and forward fill the missing values in the `Reported EPS` column

In [18]:
```
# Merge 'data' and 'df_earnings' based on their index
merged_df = data.join(df_earnings, how="left")

# Forward fill the missing values in the "Reported EPS" column
merged_df["Reported EPS"].fillna(method="ffill", inplace=True)
```

Let's have a look at the head of our merged dataframe:

In [19]:
```
merged_df.head()
```

Out[19]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
1962-01-02	0.263021	0.270182	0.263021	0.263021	0.048913	806400	NaN
1962-01-03	0.259115	0.259115	0.253255	0.257161	0.047823	1574400	NaN
1962-01-04	0.257813	0.261068	0.257813	0.259115	0.048187	844800	NaN
1962-01-05	0.259115	0.262370	0.252604	0.253255	0.047097	1420800	NaN
1962-01-08	0.251302	0.251302	0.245768	0.250651	0.046613	2035200	NaN

As we mentioned earlier, the dataframe containing information about EPS only has entries from 1996-04-07, so we will drop the rows up to that point. This will yield a dataframe with sufficent information to fit our models reasonably well.

In [20]:
```
# Drop rows with NaN values
df = merged_df.dropna()

# Print the final form of our dataframe
df
```

Out[20]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
1996-04-17	20.281250	20.281250	19.843750	20.031250	10.141947	8906000	0.14
1996-04-18	20.031250	20.156250	19.843750	19.875000	10.062837	9608000	0.14
1996-04-19	19.875000	20.062500	19.562500	19.687500	9.967900	13010400	0.14
1996-04-22	19.875000	20.187500	19.875000	20.156250	10.205229	7160800	0.14
1996-04-23	20.156250	20.281250	20.062500	20.250000	10.252701	6218800	0.14
...	...	...	...	...	...	...	...
2022-12-23	63.500000	63.869999	63.200001	63.820000	62.855492	6463300	0.69
2022-12-27	63.930000	64.290001	63.709999	64.209999	63.239597	7320700	0.69
2022-12-28	64.459999	64.650002	63.490002	63.570000	62.609272	7159400	0.69
2022-12-29	63.799999	64.150002	63.700001	63.950001	62.983532	7169300	0.69
2022-12-30	63.919998	63.919998	63.169998	63.610001	62.648666	7650200	0.69

6724 rows × 7 columns

Let's now check that the `Reported EPS` column has been filled correctly when the values are updated. For instance, let's print the transition from 2022-07-25 to 2022-07-26:

In [21]:
```
# Print our final dataframe on the selected transition date
df.tail(112)
```

Out[21]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2022-07-25	61.549999	62.299999	61.310001	62.189999	60.378578	14941700	0.64
2022-07-26	62.750000	63.799999	62.529999	63.209999	61.368870	20118000	0.7
2022-07-27	62.880001	63.189999	61.790001	63.009998	61.174698	12142900	0.7
2022-07-28	62.919998	64.250000	62.889999	64.059998	62.194111	10837000	0.7
2022-07-29	63.709999	64.290001	63.630001	64.169998	62.300907	13734200	0.7
...	...	...	...	...	...	...	...
2022-12-23	63.500000	63.869999	63.200001	63.820000	62.855492	6463300	0.69
2022-12-27	63.930000	64.290001	63.709999	64.209999	63.239597	7320700	0.69
2022-12-28	64.459999	64.650002	63.490002	63.570000	62.609272	7159400	0.69
2022-12-29	63.799999	64.150002	63.700001	63.950001	62.983532	7169300	0.69
2022-12-30	63.919998	63.919998	63.169998	63.610001	62.648666	7650200	0.69

112 rows × 7 columns

We see that the `Reported EPS` column is correctly updated on the correspoding date accoridng to the information in the dataframe `df_earnings`.

Last, we export our dataframe to a csv file so we can access it from other notebooks:

In [22]:
```
df.to_csv('KO.csv')
```

2. Tesla stock

We shall repeat the same steps we took in the previous section but focusing this time on the TSLA ticker.

Data collection

In [23]:
```
data2 = yf.download('TSLA', start=None, end=end_date)
data2
```

```
[**********100%**********]  1 of 1 completed
```

Out[23]:

Date	Open	High	Low	Close	Adj Close	Volume
2010-06-29	1.266667	1.666667	1.169333	1.592667	1.592667	281494500
2010-06-30	1.719333	2.028000	1.553333	1.588667	1.588667	257806500
2010-07-01	1.666667	1.728000	1.351333	1.464000	1.464000	123282000
2010-07-02	1.533333	1.540000	1.247333	1.280000	1.280000	77097000
2010-07-06	1.333333	1.333333	1.055333	1.074000	1.074000	103003500
...	...	...	...	...	...	...
2022-12-23	126.370003	128.619995	121.019997	123.150002	123.150002	166989700
2022-12-27	117.500000	119.669998	108.760002	109.099998	109.099998	208643400
2022-12-28	110.349998	116.269997	108.239998	112.709999	112.709999	221070500
2022-12-29	120.389999	123.570000	117.500000	121.820000	121.820000	221923300
2022-12-30	119.949997	124.480003	119.750000	123.180000	123.180000	157777300

3150 rows × 6 columns

In [24]:
```
# Check for missing values in the DataFrame
missing_values = data2.isnull().sum()

# Print the missing values count
print(missing_values)
```

```
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

In [25]:
```
# Check for NaN values in the DataFrame
nan_values = data2.isna().sum()

# Print the NaN values count
print(nan_values)
```

```
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

In [26]:
```
df_earnings2 = earnings_history_api(API_KEY, "TSLA")
df_earnings2
```

Out[26]:

	fiscalDateEnding	reportedDate	reportedEPS	estimatedEPS	surprise	surprisePercentag
0	2023-03-31	2023-04-19	0.85	0.85	0	
1	2022-12-31	2023-01-25	1.19	1.13	0.06	5.309
2	2022-09-30	2022-10-19	1.05	0.99	0.06	6.060
3	2022-06-30	2022-07-20	0.76	0.6	0.16	26.666
4	2022-03-31	2022-04-20	1.07	0.75	0.32	42.666
5	2021-12-31	2022-01-26	0.85	0.79	0.06	7.594
6	2021-09-30	2021-10-20	0.62	0.53	0.09	16.981
7	2021-06-30	2021-07-26	0.48	0.33	0.15	45.454
8	2021-03-31	2021-04-26	0.31	0.26	0.05	19.230
9	2020-12-31	2021-01-27	0.27	0.34	-0.07	-20.588
10	2020-09-30	2020-10-21	0.25	0.19	0.06	31.578
11	2020-06-30	2020-07-22	0.15	-0.01	0.16	160
12	2020-03-31	2020-04-29	0.08	-0.02	0.1	50
13	2019-12-31	2020-01-29	0.14	0.11	0.03	27.272
14	2019-09-30	2019-10-23	0.12	-0.03	0.15	50
15	2019-06-30	2019-07-24	-0.07	-0.02	-0.05	-25
16	2019-03-31	2019-04-24	-0.12	-0.05	-0.07	-14
17	2018-12-31	2019-01-30	0.13	0.08	0.05	62
18	2018-09-30	2018-10-24	0.19	0.03	0.16	533.333
19	2018-06-30	2018-08-01	-0.16	-0.2	0.04	2
20	2018-03-31	2018-05-02	-0.22	-0.24	0.02	8.333
21	2017-12-31	2018-02-07	-0.2	-0.21	0.01	4.761
22	2017-09-30	2017-11-01	-0.19	-0.15	-0.04	-26.666
23	2017-06-30	2017-08-02	-0.09	-0.12	0.03	2
24	2017-03-31	2017-05-03	-0.09	-0.09	0	
25	2016-12-31	2017-02-22	-0.14	-0.21	0.07	33.333
26	2016-09-30	2016-10-26	0.14	-0.08	0.22	27
27	2016-06-30	2016-08-03	-0.07	-0.04	-0.03	-7
28	2016-03-31	2016-05-04	-0.04	-0.05	0.01	2
29	2015-12-31	2016-02-10	-0.87	0.1	-0.97	-97

	fiscalDateEnding	reportedDate	reportedEPS	estimatedEPS	surprise	surprisePercentag
30	2015-09-30	2015-11-03	-0.04	-0.04	0	
31	2015-06-30	2015-08-05	-0.03	-0.6	0.57	9
32	2015-03-31	2015-05-06	-0.36	-0.5	0.14	2
33	2014-12-31	2015-02-11	-0.01	0.02	-0.03	-15
34	2014-09-30	2014-11-05	0.02	-0.01	0.03	30
35	2014-06-30	2014-07-31	0.01	0.04	-0.03	-7
36	2014-03-31	2014-05-07	0.01	0.1	-0.09	-9
37	2013-12-31	2014-02-19	0.02	0.01	0.01	10
38	2013-09-30	2013-11-05	0.01	0.01	0	
39	2013-06-30	2013-08-07	0.01	-0.01	0.02	20
40	2013-03-31	2013-05-08	0.01	0.04	-0.03	-7
41	2012-12-31	2013-02-20	-0.04	-0.04	0	
42	2012-09-30	2012-11-05	-0.06	-0.06	0	
43	2012-06-30	2012-07-25	-0.06	-0.06	0	
44	2012-03-31	2012-05-09	-0.05	-0.05	0	
45	2011-12-31	2012-02-15	-0.05	-0.04	-0.01	-2
46	2011-09-30	2011-11-02	-0.04	-0.04	0	
47	2011-06-30	2011-08-03	-0.04	-0.03	-0.01	-33.333
48	2011-03-31	2011-05-04	-0.03	-0.03	0	
49	2010-12-31	2011-02-15	-0.03	-0.03	0	
50	2010-09-30	2010-11-09	-0.02	-0.03	0.01	33.333
51	2010-06-30	2010-08-05	-0.0271	None	None	Nor

Now we check for None values in `df_earnings2`. We see that there are 3 entries with a `None` value:

In [27]:
```
# Check for missing values in 'df_earnings2'
(df_earnings2 == 'None').sum().sum()
```

Out[27]: 3

However, these are the first entries for `estimatedEPS`, `surprise` and `surprisePErcentage`, which obviously must be undefined as there is no possibility of prior estimation. In any case, these `None` values do not affect our analysis, as we are just picking the `reportedEPS` column.

Data organization and cleaning

Let's incorporate the `reportedEPS` column into the dataframe `data2` containing stock price data. We follow the same steps as in the previous case.

In [28]:
```
# Convert the reportedDate column to a datetime64 type
df_earnings2['reportedDate'] = pd.to_datetime(df_earnings2['reportedDate'])

# Check data type
print(df_earnings2['reportedDate'].dtype)
```

```
datetime64[ns]
```

In [29]:
```
# Set the `reportedDate' column as the index
df_earnings2 = df_earnings2.set_index('reportedDate')

# Sort in ascending order
df_earnings2.sort_index(ascending=True, inplace=True)

# Print the dataframe and check
df_earnings2.head()
```

Out[29]:

reportedDate	fiscalDateEnding	reportedEPS	estimatedEPS	surprise	surprisePercentage
2010-08-05	2010-06-30	-0.0271	None	None	None
2010-11-09	2010-09-30	-0.02	-0.03	0.01	33.3333
2011-02-15	2010-12-31	-0.03	-0.03	0	0
2011-05-04	2011-03-31	-0.03	-0.03	0	0
2011-08-03	2011-06-30	-0.04	-0.03	-0.01	-33.3333

In [30]:
```
# Drop columns different from "reportedEPS" and rename to "Reported EPS"
df_earnings2.drop(["fiscalDateEnding", "estimatedEPS", "surprise", "surprisePercent
df_earnings2.rename(columns={'reportedEPS': 'Reported EPS'}, inplace=True)
```

In [31]:
```
# Merge 'data' and 'df_earnings' based on their index
merged_df2 = data2.join(df_earnings2, how="left")

# Forward fill the missing values in the "Reported EPS" column
merged_df2["Reported EPS"].fillna(method="ffill", inplace=True)
```

In [32]:
```
merged_df2.head()
```

Out[32]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2010-06-29	1.266667	1.666667	1.169333	1.592667	1.592667	281494500	NaN
2010-06-30	1.719333	2.028000	1.553333	1.588667	1.588667	257806500	NaN
2010-07-01	1.666667	1.728000	1.351333	1.464000	1.464000	123282000	NaN
2010-07-02	1.533333	1.540000	1.247333	1.280000	1.280000	77097000	NaN
2010-07-06	1.333333	1.333333	1.055333	1.074000	1.074000	103003500	NaN

We observe that the first entries of `merged_df2` have NaN values for `Reported EPS`, as the first reported EPS happened after the stock became publicly traded. So we will drop these rows.

In [33]:
```
# Drop rows with NaN values
df2 = merged_df2.dropna()

# Print the final form of our dataframe
df2
```

Out[33]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2010-08-05	1.436000	1.436667	1.336667	1.363333	1.363333	11943000	-0.0271
2010-08-06	1.340000	1.344000	1.301333	1.306000	1.306000	11128500	-0.0271
2010-08-09	1.326667	1.332000	1.296667	1.306667	1.306667	12190500	-0.0271
2010-08-10	1.310000	1.310000	1.254667	1.268667	1.268667	19219500	-0.0271
2010-08-11	1.246000	1.258667	1.190000	1.193333	1.193333	11964000	-0.0271
...	...	...	...	...	...	...	...
2022-12-23	126.370003	128.619995	121.019997	123.150002	123.150002	166989700	1.05
2022-12-27	117.500000	119.669998	108.760002	109.099998	109.099998	208643400	1.05
2022-12-28	110.349998	116.269997	108.239998	112.709999	112.709999	221070500	1.05
2022-12-29	120.389999	123.570000	117.500000	121.820000	121.820000	221923300	1.05
2022-12-30	119.949997	124.480003	119.750000	123.180000	123.180000	157777300	1.05

3124 rows × 7 columns

We also export this dataframe to a csv file:

In [34]:
```
df2.to_csv('TSLA.csv')
```

Section 2: Exploratory Data Analysis

This section is dedicated to the Exploratory Data Analysis (EDA) step of our project. Since we are dealing with stock price time series, the EDA should investigate trends and underlying patterns in our datasets, as well as the volatility of the stocks we are studying and correlations between the different features. Here we outline the aspects that we will be studying in this section for both the KO and TSLA stocks. These companies represent large players in their respective industries and analyzing their stock price behavior will provide us with insightful knowledge into these differing markets.

Firstly, we will commence by visually inspecting the stock price data, aiming to identify discernible trends, patterns, and potential outliers. The plots of stock prices over time for both KO and TSLA will be critical to initially assess the overall movement and volatility of the stock prices.

Following this, we will perform a time series decomposition to separate the underlying trend, seasonality, and noise components. This decomposition is crucial in helping us discern the inherent characteristics of the time series and make informed decisions when modeling.

Next, we will undertake volatility analysis. In financial markets, volatility is a vital measure of the price variations. Understanding volatility patterns can help in the decision-making process by providing a measure of risk associated with the stocks.

Subsequently, we will conduct a correlation analysis to uncover potential relationships between our features, namely stock prices and earnigns per share. Discovering such relationships can aid in predicting price movements and creating a more robust forecasting model.

Lastly, we will perform a lag analysis, which will help us in identifying the temporal structure and dependencies in the data. This analysis could further reveal any autocorrelation in the series.

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
```

1. Coca-Cola stock

We start by importing the data we stored in a csv file:

In [2]:
```
# Read the csv file containing the data for the KO stock.
# index_col=0 tells pandas to use the first column as the index.
```

```
# parse_dates=True tells pandas to interpret the index as a DateTimeIndex.
df_KO = pd.read_csv('KO.csv', parse_dates=True, index_col=0)
```

Let us recall what columns this dataframe has:

In [3]:
```
df_KO.head()
```

Out[3]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
1996-04-17	20.28125	20.28125	19.84375	20.03125	10.141947	8906000	0.14
1996-04-18	20.03125	20.15625	19.84375	19.87500	10.062837	9608000	0.14
1996-04-19	19.87500	20.06250	19.56250	19.68750	9.967900	13010400	0.14
1996-04-22	19.87500	20.18750	19.87500	20.15625	10.205229	7160800	0.14
1996-04-23	20.15625	20.28125	20.06250	20.25000	10.252701	6218800	0.14

The `Open` and `Close` columns display the stock prices at opening and closing time of stock market on a given date, respectively. The columns `High` and `Low` contain the highest and lowest prices of the stock on a given date.

The `Adj Close` refers to the "Adjusted Closing Price." This is a stock's closing price on any given day of trading that has been amended to include any corporate actions that occurred at any time before the next day's open. For instance, dividends and stock splits can affect the stock price.

`Volume` refers to the number of shares traded on a given day. This is a significant metric for traders because it provides an indication of the strength or intensity behind price movements. High volume levels often signify a lot of trader interest or excitement about a stock, while low volume levels might indicate a lack of interest or awareness.

`Reported EPS` contains the last reported earnings per share (EPS). EPS is a financial metric representing the portion of a company's profit allocated to each share of a stock. It is an indicator of a company's profitability and is often considered to be one of the most important variables in determining a share's price.

Let's recheck that our dataframe has no null values:

In [4]:
```
df_KO.info()
```

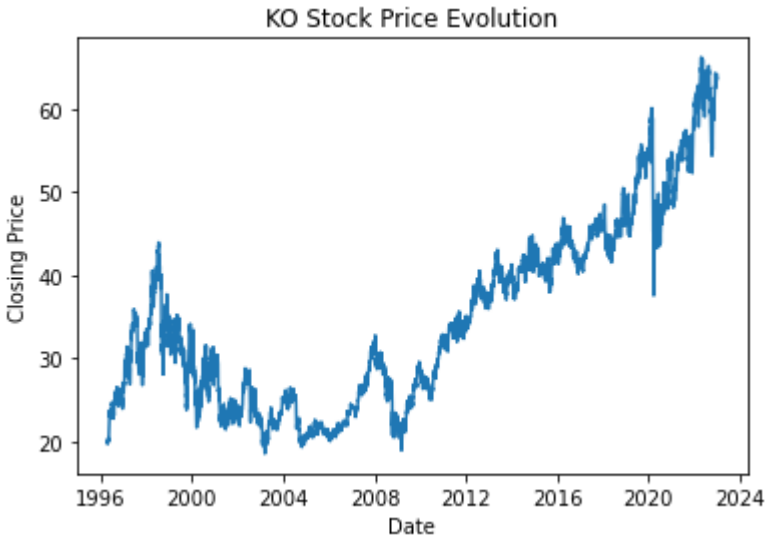
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6724 entries, 1996-04-17 to 2022-12-30
Data columns (total 7 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   Open          6724 non-null   float64
 1   High          6724 non-null   float64
 2   Low           6724 non-null   float64
 3   Close         6724 non-null   float64
 4   Adj Close     6724 non-null   float64
 5   Volume        6724 non-null   int64  
 6   Reported EPS  6724 non-null   float64
dtypes: float64(6), int64(1)
memory usage: 420.2 KB
```

In the following we will choose the `Close` column as our independent variable to model.

Data visualization

Let us begin by plotting the `Close` stock price over the period of time contained in our dataframe.

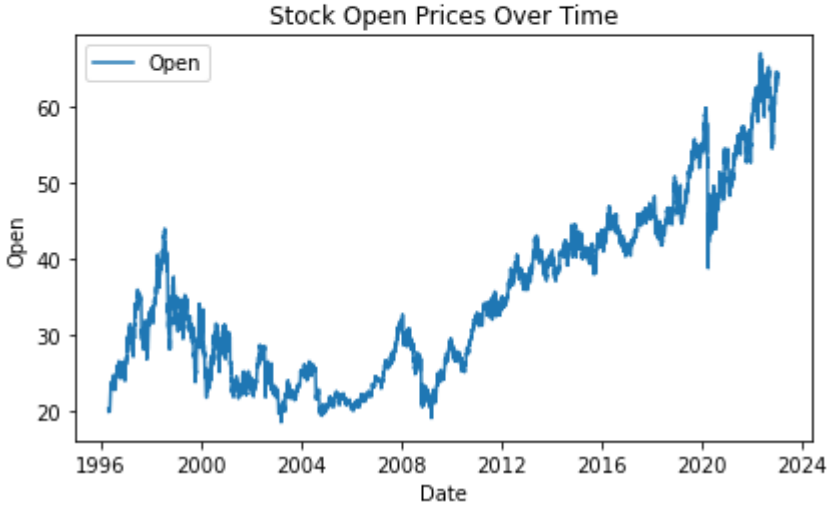
In [5]:
```
plt.plot(df_KO.index, df_KO['Close'])
plt.xlabel('Date')
plt.ylabel('Closing Price')
plt.title('KO Stock Price Evolution')
plt.show()
```



We observe a phase of growth from 1996 to 1999 followed by a contracting trend that lasts until 2006. Then a short phase of exponential growth starts and is killed by the 2008 economic crisis, with a subsequent devaluation trend until 2009. From that point to the present, the stock has experienced a general trend of growth at different rates. affected y the COVID pandemic in 2020.

The columns `Open`, `High` and `Low` exhibit a nearly identical distribution, which signals a very high correlation between the four columns.

In [6]:
```
# Create a function to display the plots of 'Open', 'High', 'Low' and 'Close' price

def plot_stock_prices(df, columns):
    fig, axes = plt.subplots(len(columns), 1, figsize=(6, 11))

    for ax, column in zip(axes, columns):
        ax.plot(df.index, df[column], label=column)
        ax.set_title('Stock ' + column + ' Prices Over Time')
        ax.set_xlabel('Date')
        ax.set_ylabel(column)
        ax.legend(loc='best')

    plt.tight_layout()
    plt.show()

# Plot the 'High', 'Low' and 'Open' columns
columns = ['High', 'Low', 'Open']
plot_stock_prices(df_KO, columns)
```



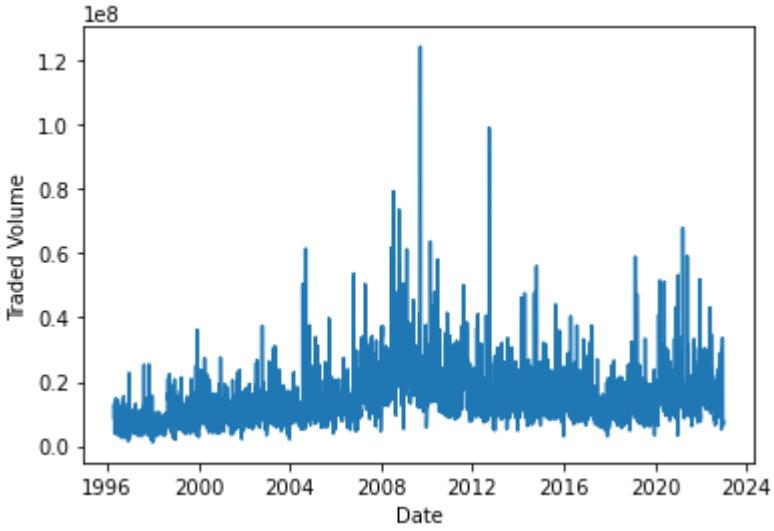
This is a general phenomenon for stock prices, which intuitively makes sense as they are all affected by the same overall market conditions, news, and sentiment surrounding the company and broader economic factors.

For the identification of trends, bullish and bearish signs, price reversals, liquidity, or interest level it is useful to plot the traded volume. For instance, volume can confirm the trend direction: in an uptrend, higher volume on up days and lower volume on down days

confirms the trend, whereas the opposite would hold true in a downtrend. On the other hand, A sudden spike in volume can often signify a price reversal. And high trading volumes typically suggest higher liquidity, which means lower transaction costs and easier ability to buy or sell without affecting the price too much.

In our case, we obtain we following volume plot for KO stock;

In [7]:
```
# Plot the 'Volume' feature over time
plt.plot(df_KO.index, df_KO['Volume'])
plt.xlabel('Date')
plt.ylabel('Traded Volume')
plt.show()
```



We can observe for instance that the highest spike in trading volume, which occured in 2009, signals the reversal of a bearish trend to a bullish phase.

Trend + seasonality + noise decomposition

A common step in the analysis of underlying patterns and behaviors of time series is the decomposition into trend, seasonality and noise. The trend component captures the underlying pattern of growth or decline in the time series data over a long period. The seasonal component captures patterns that repeat at regular intervals, such as daily, weekly, monthly, or quarterly, whereas the residual component captures the irregular fluctuations that cannot be attributed to the trend or seasonal components.

In the case of stocks, one expects the seasonality component to be of low significance. While there might be some seasonality in stock prices due to factors like quarterly earnings reports or regular economic cycles, trends and irregular fluctuations (noise) are generally considerably more significant.

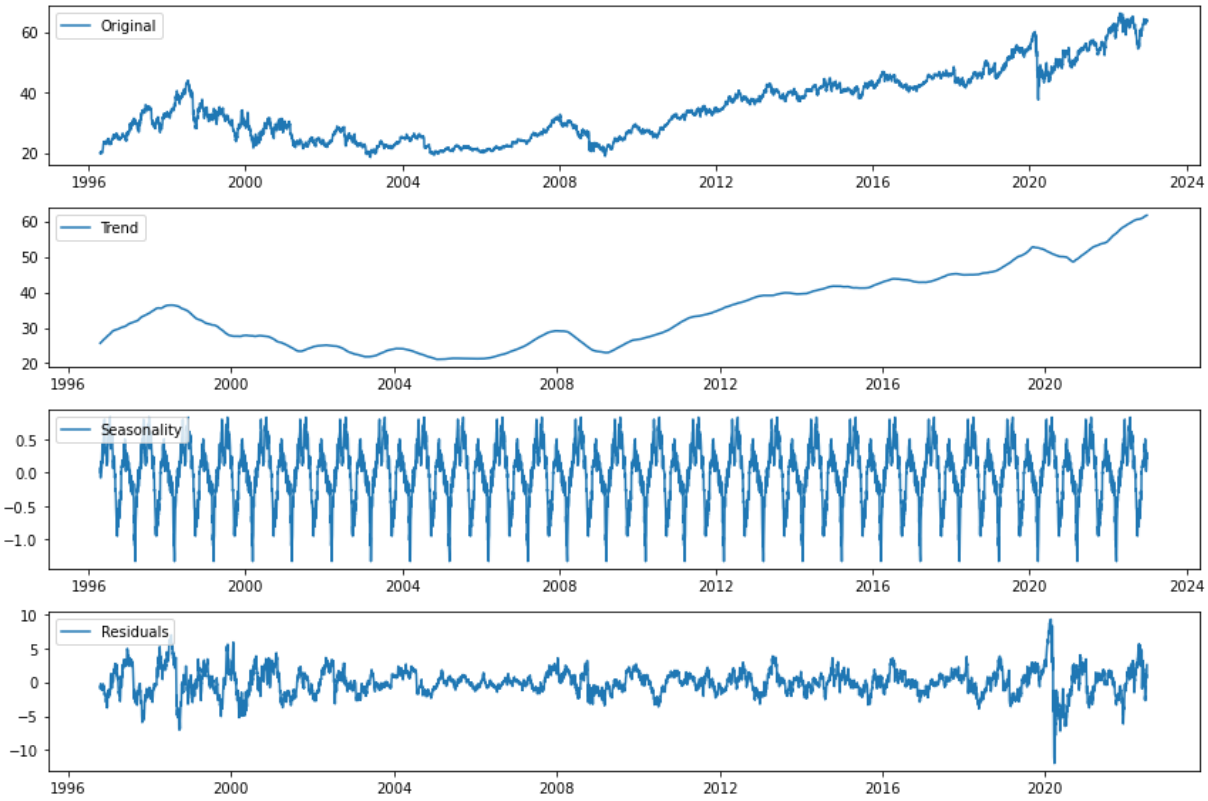
Let us perform the trend + seasonality + noise decomposition for the `Close` price of the KO stock. We use as a period the value `period=252`, namely the average number of yearly

trading days in the US. Since we will be using this decomposition throught the notebook, we will define a function that accepts as input the dataframe and column where the decomposition is to be performed, with an adjustable period.

In [8]:
```
from statsmodels.tsa.seasonal import seasonal_decompose

# Define a function that accepts as input the dataframe 'df' and column 'column_nam
# a decomposition of a time series into trend + seasonality + noise using an additi

def decompose_timeseries(df, column_name, model='additive', period=1):
    '''
    Decompose a time series into its components.

    Parameters:
    df: DataFrame
    column_name: str, the name of the column containing the time series
    model: str, 'additive' or 'multiplicative', defines the type of seasonal compon
    period: int, the period of the seasonality

    Returns:
    result: a naive decomposition of the input time series
    '''
    # Decompose the time series
    result = seasonal_decompose(df[column_name], model=model, period=period)

    # Plot the original time series, trend, seasonal component, and residuals
    plt.figure(figsize=(12,8))

    # Original
    plt.subplot(411)
    plt.plot(df[column_name], label='Original')
    plt.legend(loc='upper left')

    # Trend
    plt.subplot(412)
    plt.plot(result.trend, label='Trend')
    plt.legend(loc='upper left')

    # Seasonality
    plt.subplot(413)
    plt.plot(result.seasonal, label='Seasonality')
    plt.legend(loc='upper left')

    # Residuals
    plt.subplot(414)
    plt.plot(result.resid, label='Residuals')
    plt.legend(loc='upper left')

    plt.tight_layout()

    return result
```

Let's apply this function to our dataframe `df_KO` and column `Close` with period=252:

In [9]:
```
decompose_KO_Close = decompose_timeseries(df=df_KO, column_name='Close', period=252
```



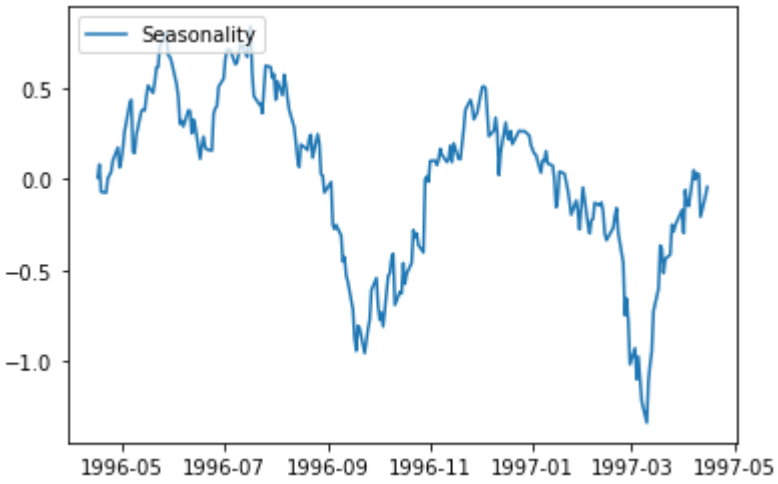
The plot of the trend confirms our previous insights: a bullish trend from 1996 to 1999 followed by a bearish phase until 2006, where an exponentail growth phase started and was frustrated by the 2008 banking crisis. A general trend of growth has been occuring since 2009.

On the other hand, we observe that the seasonality component is not very significant in magnitude, as we expected (note that the order of magnitude of the seasonality component is around 1/100 of the stock price). We further observe that movements in stock prices related to the 2020 COVID pandemic and the 2022 Ukraine war are captured by the noise signal.

We can have a closer look at the seasonality component. Let us plot a single cycle.

In [10]:
```
# Define the season length
season_length = 252

# Plot the seasonality component for one cycle
plt.subplot()
plt.plot(decompose_KO_Close.seasonal[:season_length], label='Seasonality')
plt.legend(loc='upper left')
```

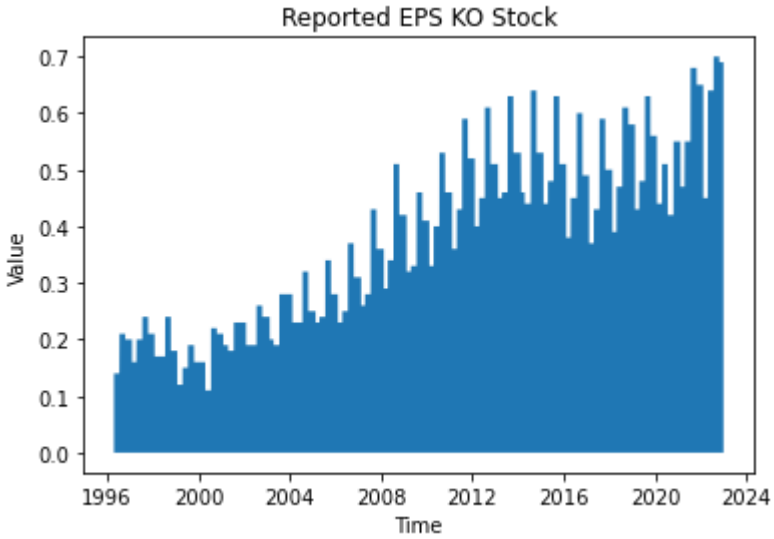
Out[10]:
```
<matplotlib.legend.Legend at 0x7fd47320bdc0>
```



We find a seasonality component that has a wave-like pattern, with peaks around December and July and valleys around March and September. Understanding seasonality can be particularly useful in trading, however a complete seasonality analysis falls beyond the scope of this project.

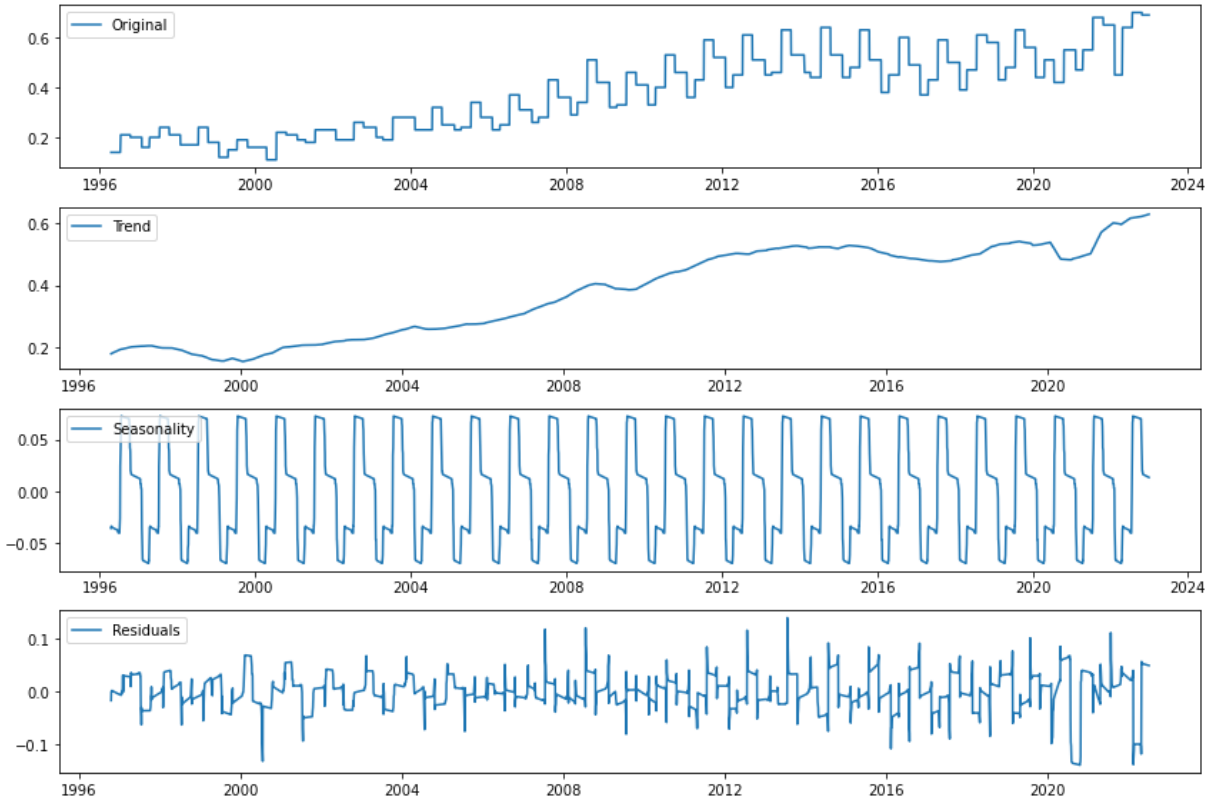
So far we have focused on the analysis of stock prices. Let us now turn to the reported earnings per share. The plot of the Reported EPS looks as follows:

In [11]:
```
# Plot the 'Reported Earnings' data
plt.fill_between(df_KO.index, df_KO['Reported EPS'])
plt.title('Reported EPS KO Stock')
plt.xlabel('Time')
plt.ylabel('Value')
plt.show()
```



We observe a general trend of growth over time with some modulation, as well as a marked seasonality signal. In order to gain a better understanding of the underlying patters, let us decompose into trend + seasonality + signal:

In [12]:
```
decompose_KO_EPS = decompose_timeseries(df=df_KO, column_name='Reported EPS', perio
```

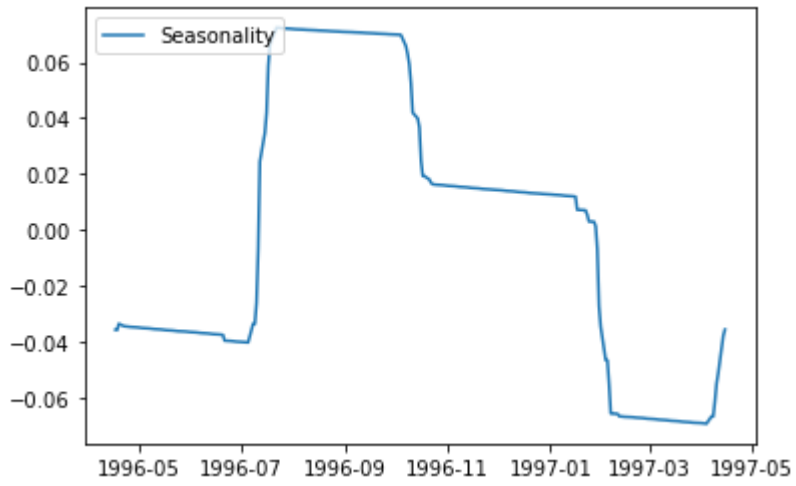


The trend plot shows indeed overall growth with some intermediate phase of stagnation. The order of magnitude of the seasonality signal is now only 1/10 of the trend and thus has a significant impact, as hinted before.

The seasonality component now has the following form:

In [13]:
```
# Plot the seasonality component for one cycle
plt.subplot()
plt.plot(decompose_KO_EPS.seasonal[:season_length], label='Seasonality')
plt.legend(loc='upper left')
```

Out[13]:
```
<matplotlib.legend.Legend at 0x7fd473226bb0>
```

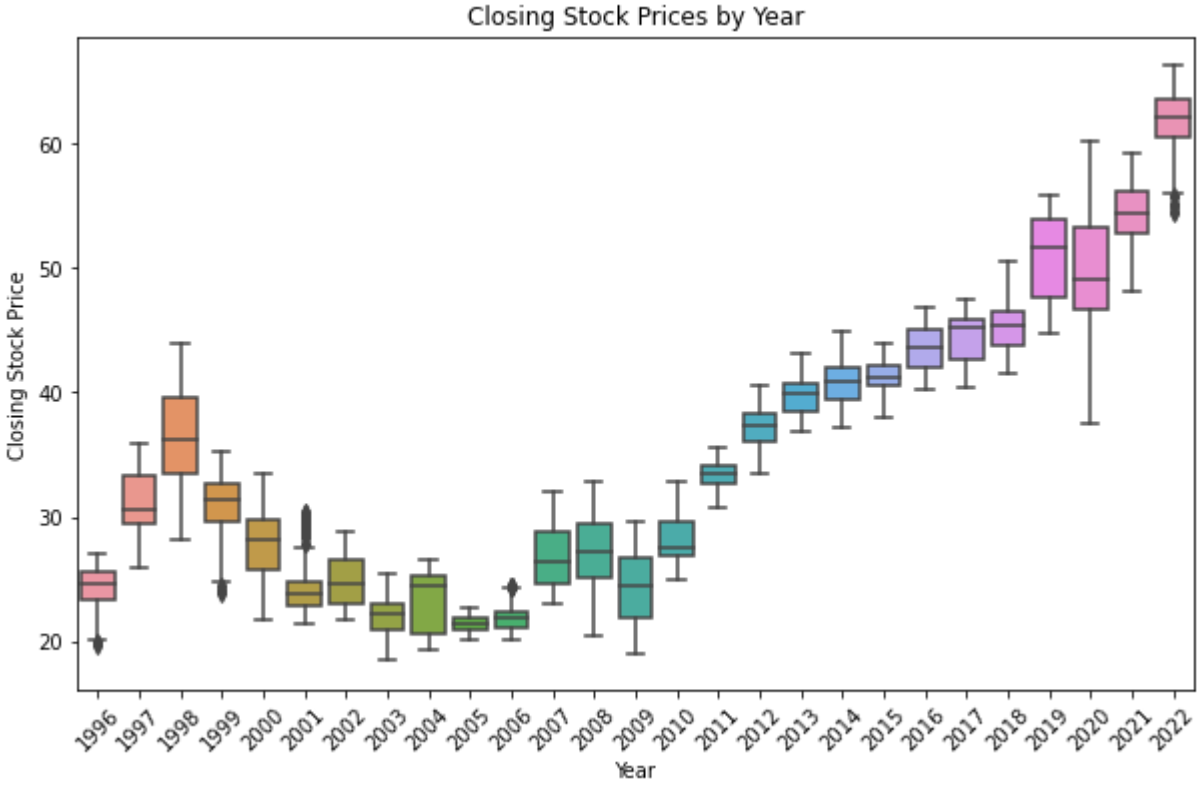


It would be interesting to relate this shape to the sells of Coca-Cola products over the year. For instance, one would expect higher sells during the spring and summer, which should result in higher EPS in July.

Outliers

In order to check for outliers, it is useful to create a boxplot of closing prices by year:

In [14]:
```
# Create a 'Year' column in the dataframe 'df_KO'
df_KO['Year'] = df_KO.index.year

# Create a boxplot of the closing prices by year
plt.figure(figsize=(10,6))
sns.boxplot(x='Year', y='Close', data=df_KO)
plt.title('Closing Stock Prices by Year')
plt.xlabel('Year')
plt.ylabel('Closing Stock Price')
plt.xticks(rotation=45)
plt.show()
```

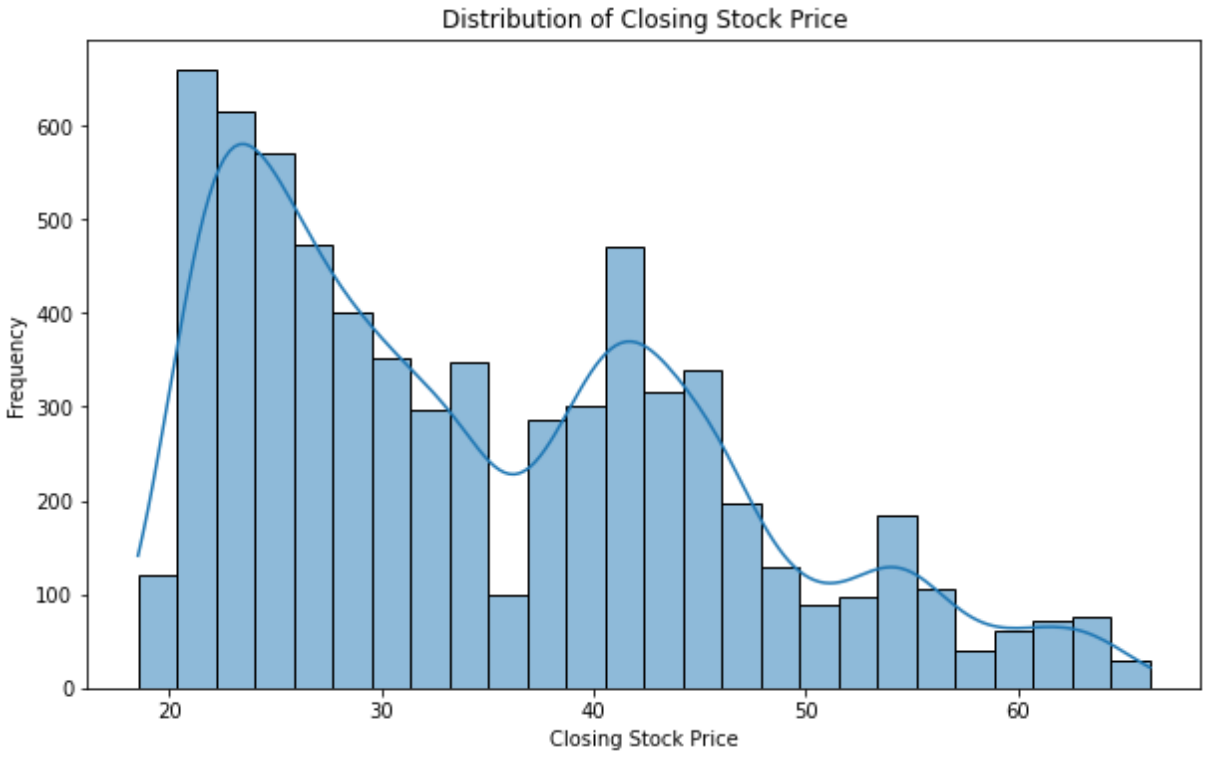


We see that only the years 1996, 1999, 2011, 2006 and 2022 have outliers. These do not deviate dramatically from the distribution, so we choose to keep them.

HIstogram and KDE estimation of probability density distribution

Let us now have a look at the histogram of the 'Close' price. Stock prices can be modeled as random variables, and so we can estimate a probability distribution for the histogram of stock prices at closing time. For this task we will use the Kernel Density Estimation (KDE) technique. KDE is a non-parametric method for estimating the probability density function of a given random variable, providing a way of smoothing data, and it is often used to identify the underlying distribution of the data or find the modes (peaks) in the data. It is independent of the choice of the number of bins.

In [15]:
```
# Plot the histogram of the 'Close' stock price column.
plt.figure(figsize=(10, 6))
sns.histplot(df_KO['Close'], kde=True) # 'kde=True' introduces the Kernel Density E
plt.title('Distribution of Closing Stock Price')
plt.xlabel('Closing Stock Price')
plt.ylabel('Frequency')
plt.show()
```



We know from the prices plot that most of the prices are comprised within the range between 20 and 30 dollars. This makes the KDE estimation curve skewed to the low end. We also observe four peaks around the closing stock prices of 23, 41, 55 and 62 dollars, which can be interpreted as the modes.

Volatility

Volatility is a statistical measure of the degree of variation or dispersion in a series of data. In simple terms, it represents the amount of uncertainty or risk related to the size of changes in the values of a data series. In the context of financial markets, volatility is typically calculated

as the standard deviation of logarithmic returns, often annualized for easier comparison across assets and markets (recall that the standard deviation measures how spread out the numbers in a data set are around the mean).

When computing the volatility of a stock price we first compute the logarithmic returns, as they are additive (as opposed to simple returns). Logarithmic returns are defined as the difference in the logarithm of two consecutive prices. Once these returns are calculated, the standard deviation of their distribution is computed. As mentioned earlier, this provides a measure of how much the price is expected to deviate from its average value on a day-to-day basis.

Let us first compute the log returns of the KO stock and visualize its distribution:

In [16]:
```
# Calculate log returns
df_KO['Log_Returns'] = np.log(df_KO['Close'] / df_KO['Close'].shift(1))

# Drop NA values
df_KO = df_KO.dropna()

# Plot log returns over time
plt.figure(figsize=(10, 6))
plt.plot(df_KO['Log_Returns'])
plt.title('Log Returns Over Time')
plt.xlabel('Date')
plt.ylabel('Log Returns')
plt.show()
```

We observe that this plot looks like withe noise. We can use the Augmented Dickey-Fuller test to check that indeed this distribution is stationary.

In [17]:
```
from statsmodels.tsa.stattools import adfuller

result = adfuller(df_KO['Log_Returns'])
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])

print('Critical Values:')
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: -42.928660
p-value: 0.000000
Critical Values:
        1%: -3.431
        5%: -2.862
        10%: -2.567
```

The null hypothesis of the ADF test is that the time series is non-stationary. So, when the p-value is less than the level of significance (0.05), we reject the null hypothesis and infer that the time series is stationary. The more negative the vaue of the ADF statistic, the more confident we are the data series behaves as stationary. More concretely: 1%: -3.431: If the test statistic is less than -3.431, then the null hypothesis can be rejected with a 99% level of confidence.

5%: -2.862: If the test statistic is less than -2.862, then the null hypothesis can be rejected with a 95% level of confidence.

10%: -2.567: If the test statistic is less than -2.567, then the null hypothesis can be rejected with a 90% level of confidence.

In our case, the ADF statistic value and p-value clearly indicate a stationary behavior.

We shall now compute the annualized volatility of the KO stock.

In [18]:
```
# Calculate the standard deviation of log returns (volatility)
volatility = df_KO['Log_Returns'].std()

# Calculate the annualized volatility
annualized_volatility = volatility * np.sqrt(252) * 100

print("Volatility: ", annualized_volatility)
```

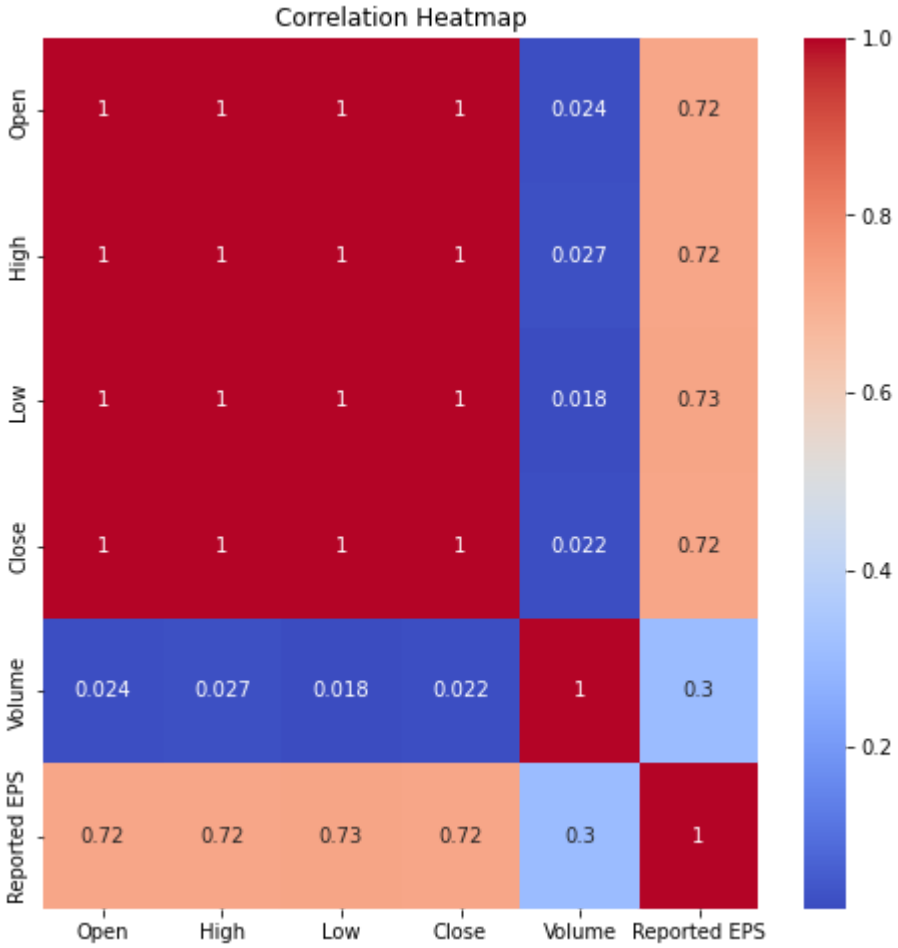
```
Volatility:  22.309793118514104
```

Generally speaking, a stock with an annualized volatility above 30% might be considered highly volatile. A stock with annualized volatility below 20% might be considered of low volatility. In the case of the KO stock, we can say the volatility is rather on the low edge.

Correlations

We turn to the investigation of the correlations between the different columns in our dataframe:

In [19]:
```
# Create a heatmap of the correlation between stock prices
corr = df_KO[['Open', 'High', 'Low', 'Close', 'Volume', 'Reported EPS']].corr()
plt.figure(figsize=(8,8))
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```



The `Open`, `High`, `Low` and `Close` prices are highly correlated among themselves, as we expected from the similarity between their plots.

Price and Volume (0.024): this very low correlation suggests that the daily fluctuations in Coca Cola's trading volume do not have a strong linear impact on the price of the stock on a day-to-day basis. Nevertheless, volume can play a role in price movements during punctual events such as earnings announcements, product launches, or other news events.

Price and Reported EPS (0.72): this is a significant positive correlation. The reason for this is that the EPS is a key indicator of a company's profitability, which is one of the primary drivers of stock price. It suggests that as Coca Cola's earnings grow, so does its stock price. Coca Cola is a mature, dividend-paying company and consistent earnings growth could be

interpreted as a sign of stability and predictability, which could increase investor confidence and drive up the stock price.

Volume and Reported EPS (0.3): a positive but relatively weaker correlation between trading volume and reported EPS may suggest that trading activity increases slightly when earnings are announced. This could be due to a variety of reasons: investors reacting to the news, increased interest from traders looking to profit from the increased volatility, etc. The correlation is relatively weak, so it is likely that other factors also have a significant impact on trading volume.

It is also interesting to explore correlations between the KO stock prices and the S&P500. First we ned to import the S&P500 data for the same period of time comprised in `df_KO`:

In [20]:
```
import yfinance as yf

# Define the SP500 ticker symbol and start and end date
ticker = "^GSPC"
start_date = "1996-04-17"
end_date = "2022-12-31"

# Fetch the historical data using yfinance and store in the dataframe 'df_SP'
df_SP = yf.download(ticker, start=start_date, end=end_date)
```

```
[**********100%**********]  1 of 1 completed
```

Let's check that the dataframe `df_SP` has the correct format:

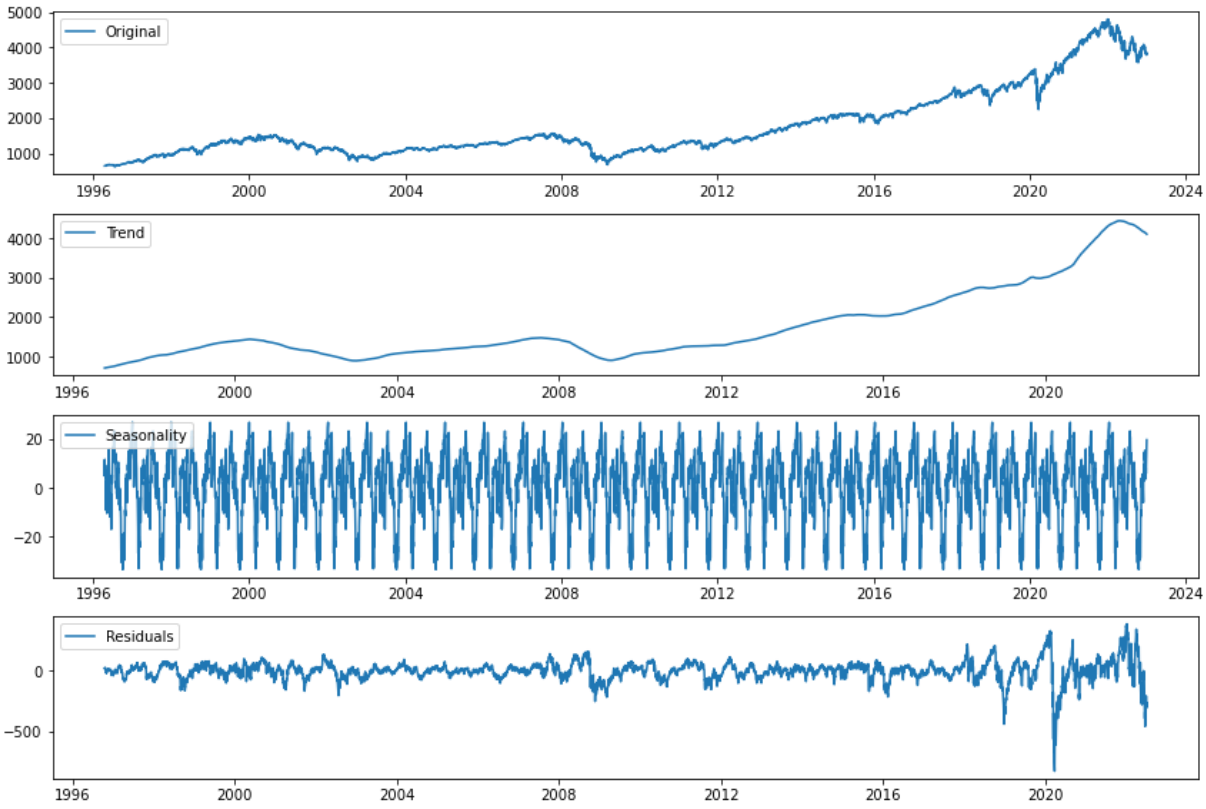
In [21]:
```
df_SP.head()
```

Out[21]:

Date	Open	High	Low	Close	Adj Close	Volume
1996-04-17	645.000000	645.000000	638.710022	641.609985	641.609985	465200000
1996-04-18	641.609985	644.659973	640.760010	643.609985	643.609985	415150000
1996-04-19	643.609985	647.320007	643.609985	645.070007	645.070007	435690000
1996-04-22	645.070007	650.909973	645.070007	647.890015	647.890015	395370000
1996-04-23	647.890015	651.590027	647.700012	651.580017	651.580017	452690000

For the sake of completeness let's visualize the data for the timeframe we are focusing on and perform a trend + season + noise decomposition:

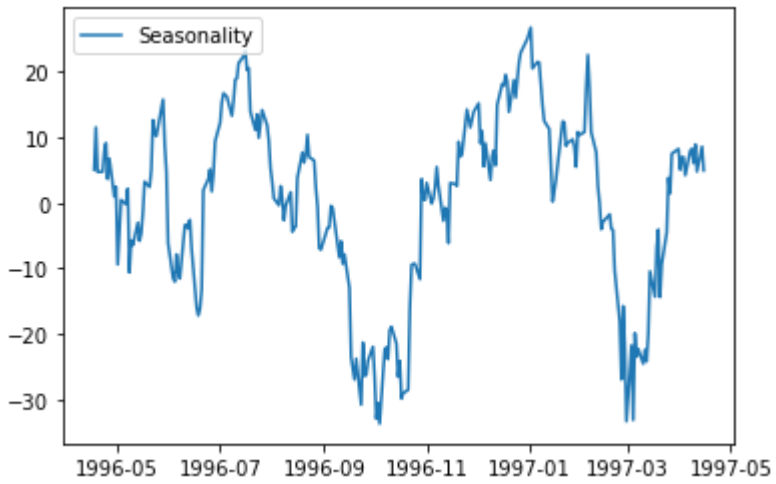
In [22]:
```
decompose_SP_Close = decompose_timeseries(df=df_SP, column_name='Close', period=252
```



The seasonality components has the following shape:

In [23]:
```
# Plot the seasonality component for one cycle
plt.subplot()
plt.plot(decompose_SP_Close.seasonal[:season_length], label='Seasonality')
plt.legend(loc='upper left')
```

Out[23]:
```
<matplotlib.legend.Legend at 0x7fd4734f0dc0>
```

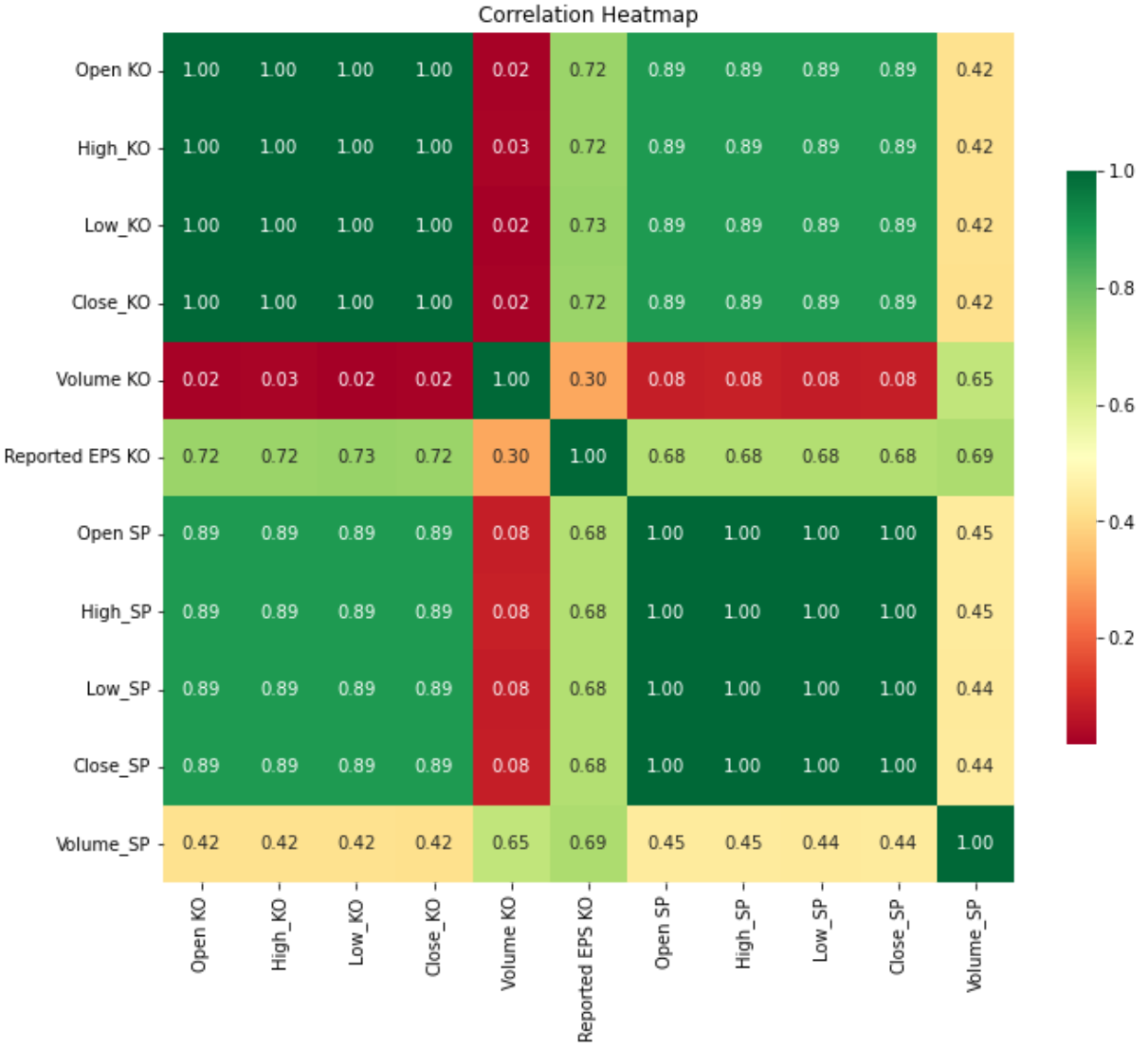


We observe a patter similar to that of the KO ticker.

Now we compute correlations:

In [24]:
```
selected_data = pd.DataFrame({
    'Open KO': df_KO['Open'],
```

```
    'High_KO': df_KO['High'],
    'Low_KO': df_KO['Low'],
    'Close_KO': df_KO['Close'],
    'Volume KO': df_KO['Volume'],
    'Reported EPS KO': df_KO['Reported EPS'],
    'Open SP': df_SP['Open'],
    'High_SP': df_SP['High'],
    'Low_SP': df_SP['Low'],
    'Close_SP': df_SP['Close'],
    'Volume_SP': df_SP['Volume']
})

# calculate correlation matrix
corr = selected_data.corr()

# plot the heatmap
plt.figure(figsize=(10, 10))
sns.heatmap(corr, annot=True, fmt=".2f", square=True, cmap='RdYlGn', cbar_kws={"shr
plt.title("Correlation Heatmap")
plt.tight_layout()
plt.show()
```



The upper left square is just the correlation heatmap we plotted previously. The off-diagonal squares includes the correlations with the S&P500 data.

Coca Cola Prices and S&P 500 Prices (0.89): this high positive correlation suggests that Coca Cola's stock price tends to move in the same direction as the overall market. This can happen when there are shared influences, like market factors (the overall market sentiment can impact all stocks), economic factors (certain macroeconomic factors, like changes in interest rates), company's market exposure (if the company has a broad market exposure, its stock price could be highly correlated with the S&P 500), and investment strategy (some funds and investors use index-based strategies, so stocks included in the index will often move in tandem with the index). Coca Cola, being a large and established company, would be expected to broadly follow these market trends.

Coca Cola Volume and S&P 500 Prices (0.08): this very low correlation suggests that there is no linear relationship between the trading volume of Coca Cola's stock and the price movements of the S&P 500. This could indicate that the daily trading volume of Coca Cola's shares is more influenced by company-specific news and events rather than the general market trends.

Coca Cola Prices and S&P 500 Volume (0.42): this moderate correlation might suggest that on days when overall market activity is high, there is a tendency for Coca Cola's stock price to increase. This could be due to various reasons, including increased buying activity in the market on high volume days which could push up prices of stocks including Coca Cola.

Coca Cola Volume and S&P 500 Volume (0.65): this relatively high positive correlation indicates that the trading volume of Coca Cola's stock tends to increase when the overall market volume (S&P 500 volume) increases, and vice versa. This suggests that investor activity in Coca Cola's stock is somewhat tied to overall market activity. This could potentially be due to a few factors. One is market-wide trends: on days when overall market activity is high (perhaps due to macroeconomic news or events affecting many or all stocks), both the S&P 500 volume and the volume of individual stocks might be expected to increase. Another one is investor sentiment: for example, in times of high uncertainty, trading volumes across the market, including both the individual stock and the S&P 500, could rise as investors reassess their positions.

Coca Cola EPS and S&P 500 Prices (0.68):this suggests a fairly strong relationship between Coca Cola's earnings per share and the overall market price level, and might be due to the fact that Coca Cola's profitability can be influenced by the same macroeconomic factors that affect the overall market. This hypothesis is reinforced by the fact that the correlation between Coca Cola stock price and its EPS is quite similar (0.73).

Coca Cola EPS and S&P 500 Volume (0.69): a significant correlation here might imply that higher market activity (S&P 500 volume) is associated with higher Coca Cola's earnings per

share. This could be because high trading volumes in the market often occur during periods of strong economic performance, which could also lead to higher corporate earnings.

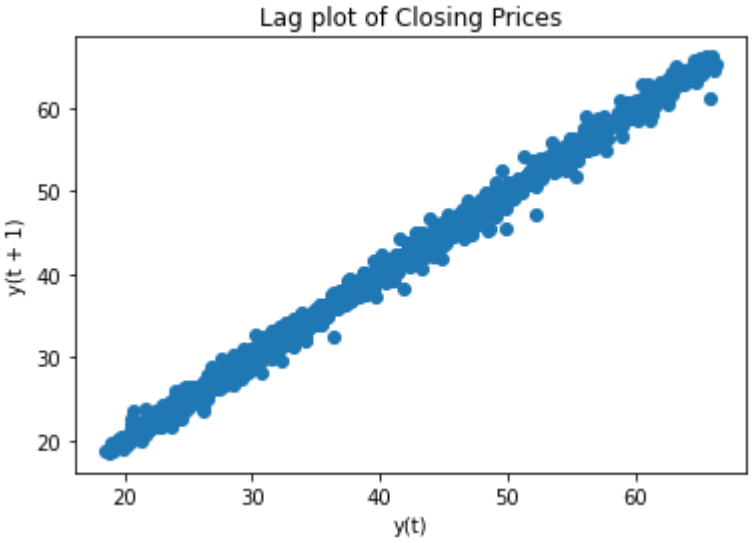
Lag analysis

Lag analysis is a method used to examine the relationship between data points at different times in a time series. It involves creating a series of lagged variables, where values at one period are compared with values at a previous period. The purpose is to identify patterns over time, such as autocorrelation or seasonality.

A lag plot for a time series will scatter plot the value at time t on the x-axis and the value at time t+lag on the y-axis. If the points cluster along a diagonal line from the bottom-left to the top-right of the plot, it suggests a positive correlation relationship. If the points cluster along a diagonal line from the top-left to the bottom-right, it suggests a negative correlation relationship. If points are scattered around or form a cloud without a noticeable pattern, it indicates that there is little to no autocorrelation, meaning past values do not help to predict future values.

Let us plot the lag diagram for the KO stock:

In [25]:
```
from pandas.plotting import lag_plot, autocorrelation_plot

# Create a lag plot
plt.figure() # create a new figure for the plot
lag_plot(df_KO['Close'])
plt.title('Lag plot of Closing Prices')
plt.show()
```



We obtain a lag plot for the KO stock price tightly clustered along a diagonal line from the bottom-left to the top-right, in a rather homogeneous fashion throughout the plot. This suggests a strong positive correlation relationship between the stock price at time 't' and at time 't+1'. Essentially, this means that if the stock price is high (or low) at time 't', it's likely to

be high (or low) at the next time point 't+1'. Generally speaking, this pattern can indicate a form of momentum or trend in the price movement: if the price is increasing, it continues to increase, and if it's decreasing, it continues to decrease. Note that this is indeed the overall pattern we found in our data visualization analysis of the KO stock price: we identified a succession of different phases with trends of growth and contraction in the plot for the `Close` KO price. The fact that the data points are clustered around the diagonal in a rather homogeneous way indicates homoscedasticity, meaning that the stock's volatility is can be considered approximately constant over time. This hypothesis is reinforced by the observation that, in the plot of KO stock price, the highs and lows within every trend phase tend be of similar magnitude.

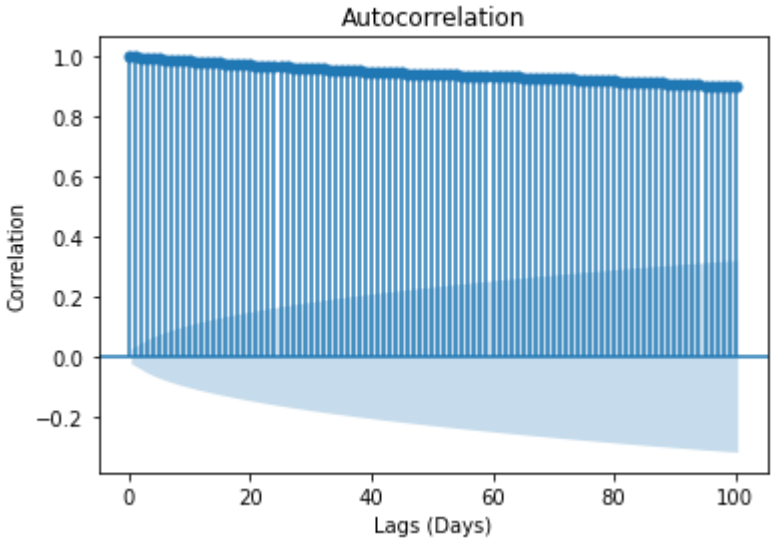
The ACF (Autocorrelation Function) plot gives us insights into the time dependency of a given time series. The x-axis represents the number of lags we look back in the time series. The y-axis represents the correlation values. Each vertical line (stem) in the plot corresponds to the correlation between the time series and its lagged version.

The blue shaded region in the plot represents a confidence interval (often set at 95%). If the stem extends beyond this region, it means the correlation for the specific lag number is statistically significant.

Autocorrelation measures the relationship between a variable's current value and its past values. If the ACF plot shows slow decay, that usually means the series has a strong trend. If the plot shows a sharp drop after a certain number of lags, that could mean the series is seasonal. If the autocorrelation is close to 1 for the first few lags, then decreases to zero, that often means the series is a random walk or has a strong autoregressive component. If the autocorrelations are near zero for all lags, the series is likely purely random (white noise).

Let us plot the ACF function for the KO stock:

In [58]:
```
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(df_KO['Close'], lags=100)
plt.xlabel('Lags (Days)', fontsize=10)
plt.ylabel('Correlation', fontsize=10)
plt.show()
```



We notice that the autocorrelations for small lags are large and positive, then slowly decrease as the lags increase. This is interpreted as an indication of a trend and non-stationarity. Tis is indeed a common pattern for stock price time series.

2. Tesla stock

We now turn to the EDA of the TSLA stock prices. First we import the csv datafile as a dataframe:

In [28]:
```
# Read the csv file containing the data for the KO stock.
# index_col=0 tells pandas to use the first column as the index.
# parse_dates=True tells pandas to interpret the index as a DateTimeIndex.
df_TSLA = pd.read_csv('TSLA.csv', parse_dates=True, index_col=0)
```

Let's check that our dataframe has the right format:

In [29]:
```
df_TSLA.head()
```

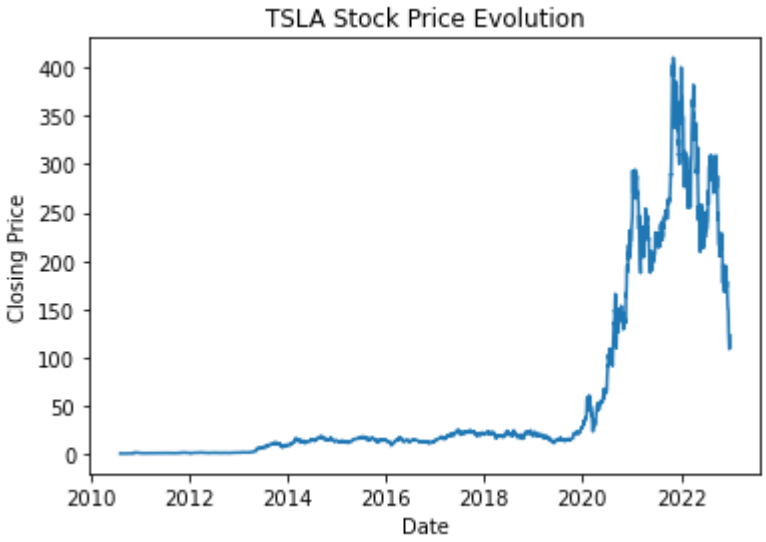
Out[29]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2010-08-05	1.436000	1.436667	1.336667	1.363333	1.363333	11943000	-0.0271
2010-08-06	1.340000	1.344000	1.301333	1.306000	1.306000	11128500	-0.0271
2010-08-09	1.326667	1.332000	1.296667	1.306667	1.306667	12190500	-0.0271
2010-08-10	1.310000	1.310000	1.254667	1.268667	1.268667	19219500	-0.0271
2010-08-11	1.246000	1.258667	1.190000	1.193333	1.193333	11964000	-0.0271

Data visualization

Let us plot the `Close` price data of the TSLA stock:

In [30]:
```
plt.plot(df_TSLA.index, df_TSLA['Close'])
plt.xlabel('Date')
plt.ylabel('Closing Price')
plt.title('TSLA Stock Price Evolution')
plt.show()
```

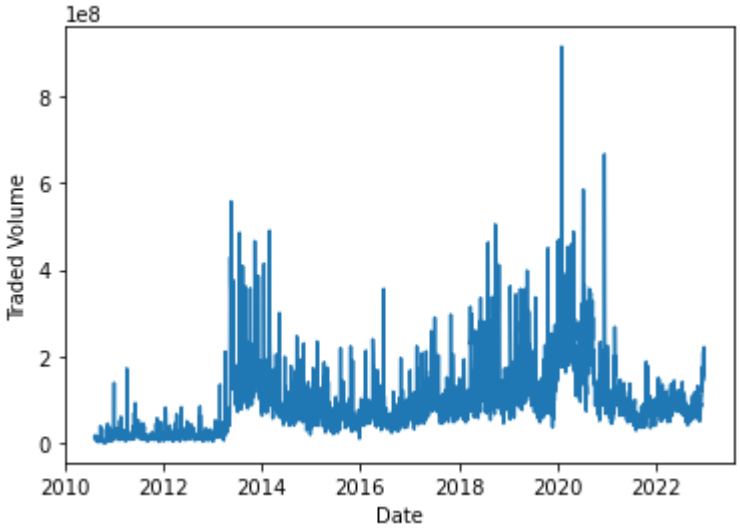


We observe a first phase of almost constant closing price that lasts from 2010 to 2020, followed by two phases of exponential growth between 2020 and 2022 that are separated by a slight decline in the stock price. In 2022, the stock price has declined throughut the year.

We will see later that the `Open`, `High` and `Low` prices have a highly correlated distribution with `Close` price, so we shall not plot them.

The series of trading volume values displays more frequent spikes over time than in the case of the KO stock:

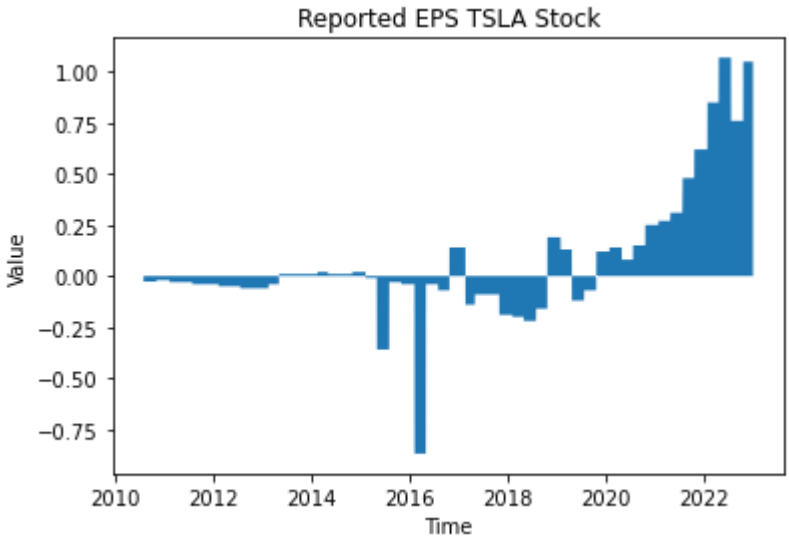
In [31]:
```
plt.plot(df_TSLA.index, df_TSLA['Volume'])
plt.xlabel('Date')
plt.ylabel('Traded Volume')
plt.show()
```



A volume plot that features multiple spikes indicates periods of significant trading activity for the stock. Such a spike in trading volume usually happens because of an event or situation that draws increased attention to the stock from investors, and may be related to the high volatility of TSLA.

Let's have a look now at the earnings per share:

In [32]:
```
plt.fill_between(df_TSLA.index, df_TSLA['Reported EPS'])
plt.title('Reported EPS TSLA Stock')
plt.xlabel('Time')
plt.ylabel('Value')
plt.show()
```



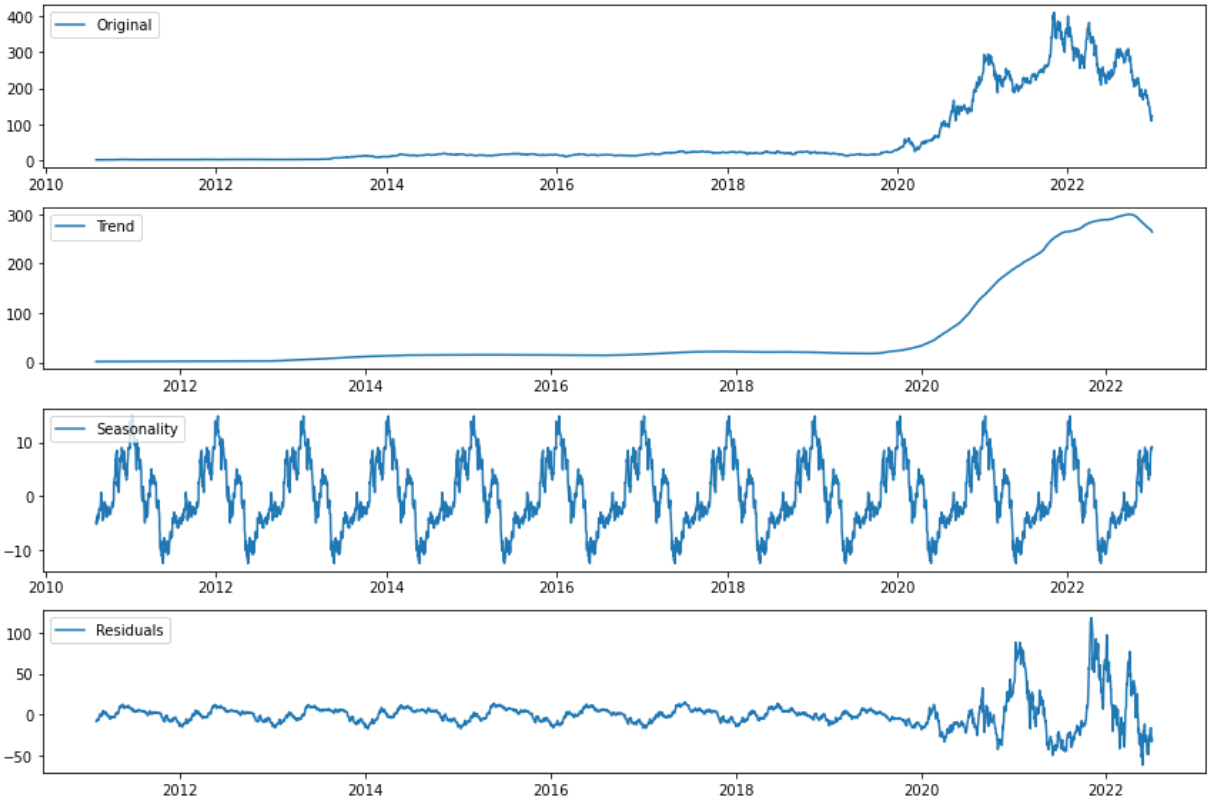
We observe that for the first ten years of the company's existence the reported EPS are negative most of the times, usually not going below -0.25, with a very sharp negative value around -0.8 in 2016. A negative EPS indicates that the company has reported a net loss for that period. This can happen if a company's expenses exceed its revenue. It's not an uncommon scenario for young companies, particularly in sectors like technology, where

initial operating expenses can be high, and the company might not be profitable yet. In the case of TSLA, a profitable phase of exponential growth followed from 2020 to 2022, with oscillations ever since.

Trend + seasonality + noise decomposition

Now we proceed to the decomposition of the TSLA stock price data series into trend, seasonality and noise components.

In [33]:
```
decompose_TSLA_Close = decompose_timeseries(df=df_TSLA, column_name='Close', period
```



We observe that the trend component follows the general pattern we have described previosuly: a rather stable phase for the first ten years with a slight upward trend, followed by an explosive growth that eventually gets modulated to start a phase of contraction, with the stock price remaining on the high side.

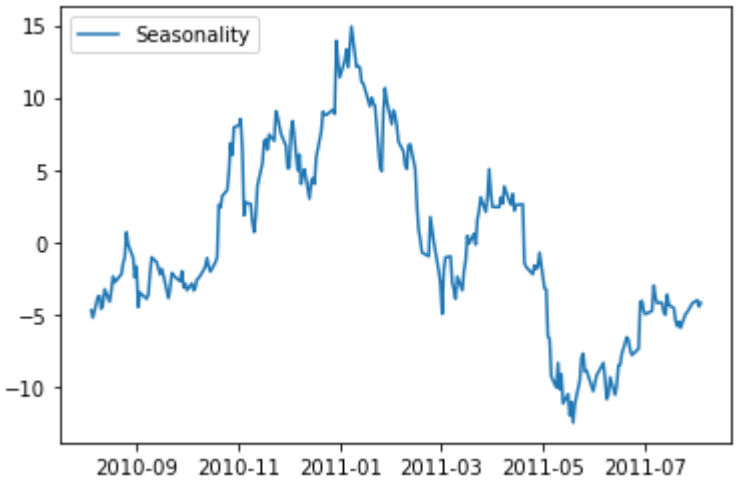
In this case the seasonaility component is is stronger than for the KO stock, being only 1/10 of the order of magnitude of the highest values of the trend component. The noise signal appears to be rather periodic and of the same magnitude as the seasonality component for the first ten years, then becoming more irregular and more significant.

We can also have a closer look at one period of the seasonality signal:

In [34]:
```
# Seasonality for one period
plt.subplot()
```

```
plt.plot(decompose_TSLA_Close.seasonal[:season_length], label='Seasonality')
plt.legend(loc='upper left')
```

Out[34]: <matplotlib.legend.Legend at 0x7f8c7810e0a0>

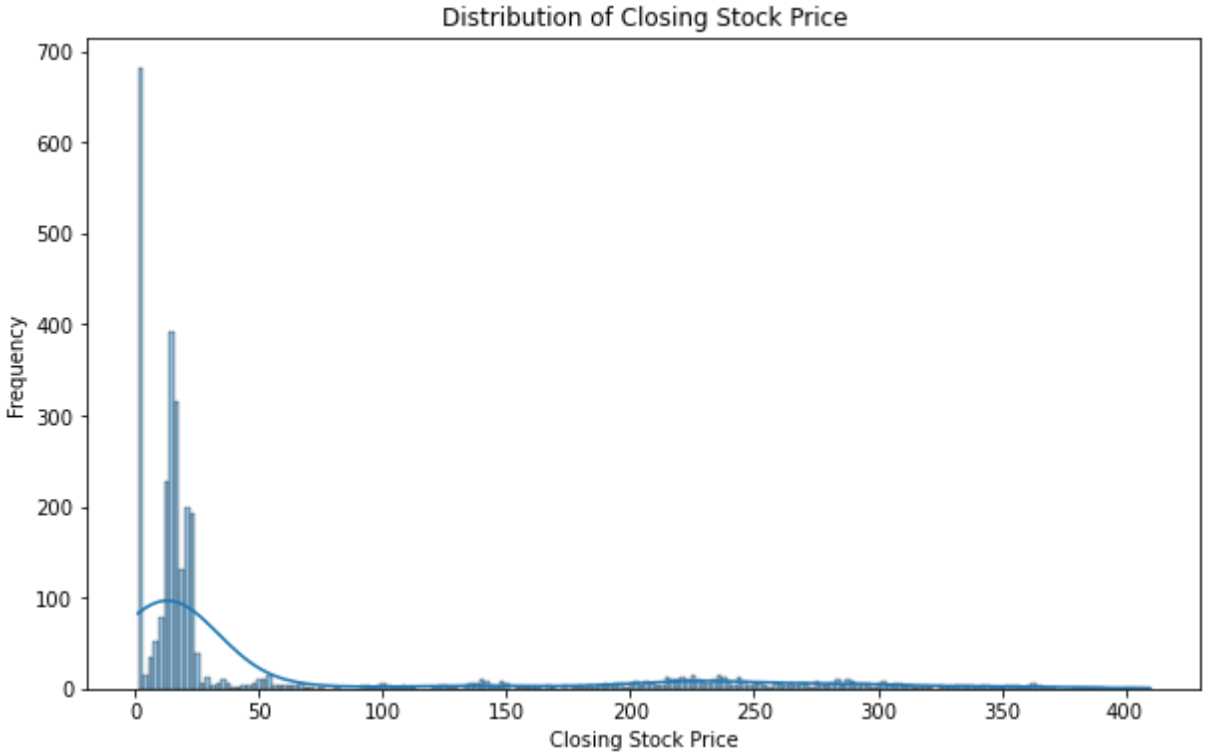


This also has a wave-like form with peaks and valleys, but its pattern is different from the KO and SP500 stocks.

Histogram and KDE estimation of probability distribution

In this section we plot the histogram of stock prices of TSLA ticker and perform a KDE estimation of the probability density:

In [35]:
```
# Plot the histogram of the 'Close' stock price column.
plt.figure(figsize=(10, 6))
sns.histplot(df_TSLA['Close'], kde=True) # 'kde=True' introduces the Kernel Density
plt.title('Distribution of Closing Stock Price')
plt.xlabel('Closing Stock Price')
plt.ylabel('Frequency')
plt.show()
```

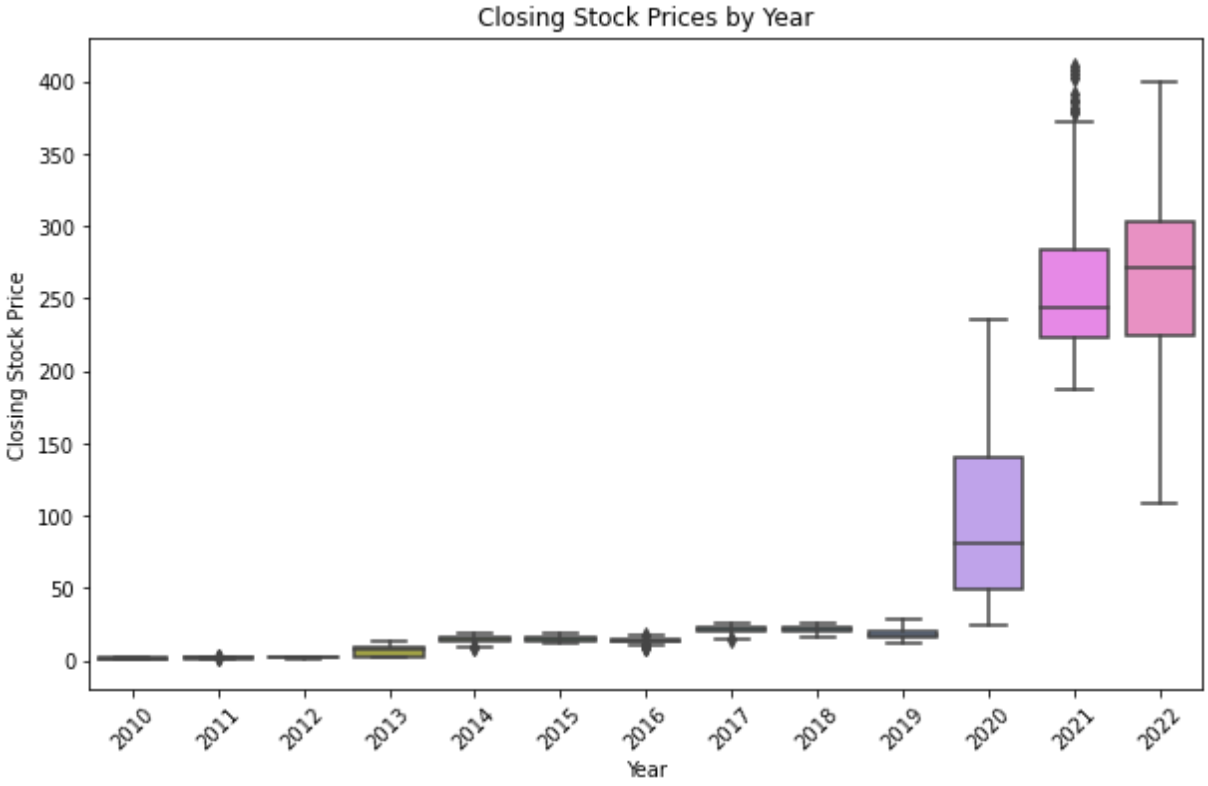


The histogram is extremely skewed to the left, as the value of the TSLA share remained under 30 dollars for more than half of the timeframe of its existence. The KDE has a single peak around 20 dollars.

Outliers

Let us check now for outliers in the distribution of TSLA stock prices:

In [36]:
```
# Create a 'Year' column in the dataframe 'df_KO'
df_TSLA['Year'] = df_TSLA.index.year

# Create a boxplot of the closing prices by year
plt.figure(figsize=(10,6))
sns.boxplot(x='Year', y='Close', data=df_TSLA)
plt.title('Closing Stock Prices by Year')
plt.xlabel('Year')
plt.ylabel('Closing Stock Price')
plt.xticks(rotation=45)
plt.show()
```



We observe a significant amount of outliers in 2021. This is easily explained as in this year the share price witnessed a sharp, volatile phase of exponential growth.

Volatility

It is often stated in media that the TSLA stock is highly volatile. We will check this claim in this section by computing its annualized volatility.

First, we compute the log returns and plot the resulting series:

In [37]:
```
# Calculate log returns
df_TSLA['Log_Returns'] = np.log(df_TSLA['Close'] / df_TSLA['Close'].shift(1))

# Drop NA values
df_TSLA = df_TSLA.dropna()

# Plot log returns over time
plt.figure(figsize=(10, 6))
plt.plot(df_KO['Log_Returns'])
plt.title('Log Returns Over Time')
plt.xlabel('Date')
plt.ylabel('Log Returns')
plt.show()
```

Again, this looks like white noise. The ADF test yields the following values:

In [38]:
```
result = adfuller(df_KO['Log_Returns'])
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])

print('Critical Values:')
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: -42.928660
p-value: 0.000000
Critical Values:
        1%: -3.431
        5%: -2.862
        10%: -2.567
```

We can conclude with enough confidence that the series is stationary.

In order to compute the volatility, we have to compute the standard deviation:

In [39]:
```
# Calculate the standard deviation of log returns (volatility)
volatility = df_TSLA['Log_Returns'].std()

# Calculate the annualized volatility
annualized_volatility = volatility * np.sqrt(252) * 100

print("Volatility: ", annualized_volatility)
```

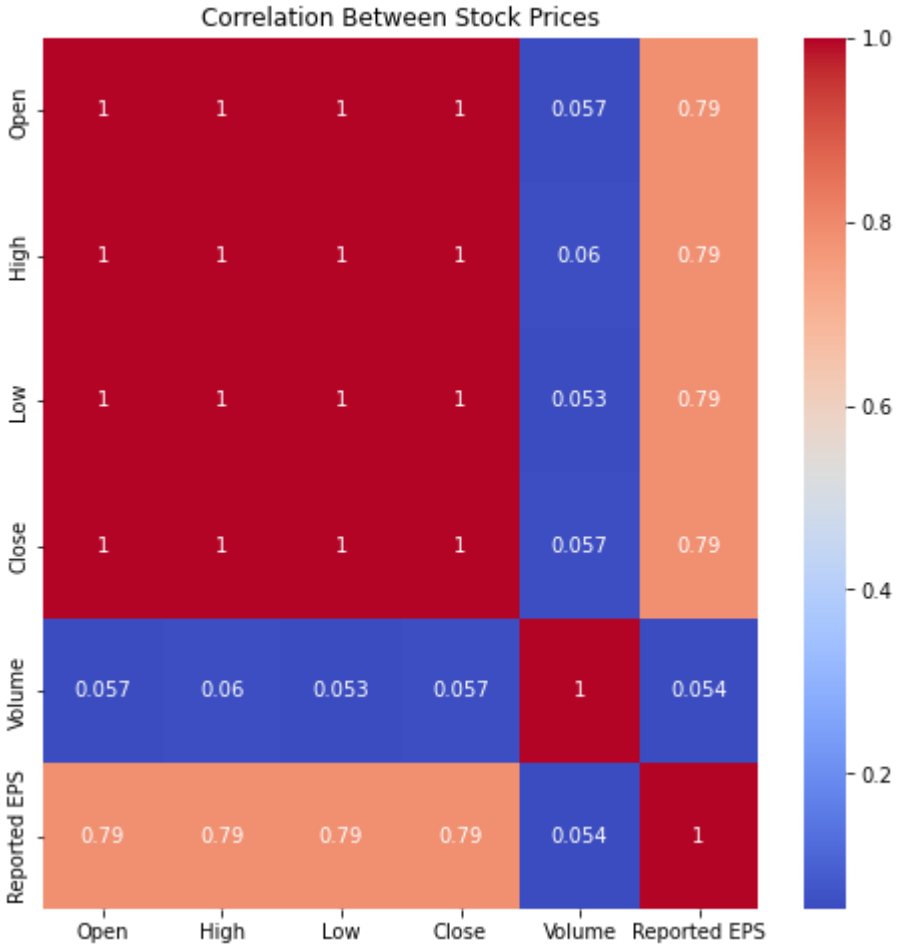
```
Volatility:  56.461429838933256
```

We obtain a volatility of around 56%, which is quite high (recall that the threshold for high volatility is 30%). Thus we corroborate the claim of the high volatility of TSLA stock.

Correlations

We now wonder about how correlated are the different features of our dataset of the TSLA time series. Let's start by plotting the correlation heatmap:

In [40]:
```
# Create a heatmap of the correlation between stock prices
corr = df_TSLA[['Open', 'High', 'Low', 'Close', 'Volume', 'Reported EPS']].corr()
plt.figure(figsize=(8,8))
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title('Correlation Between Stock Prices')
plt.show()
```

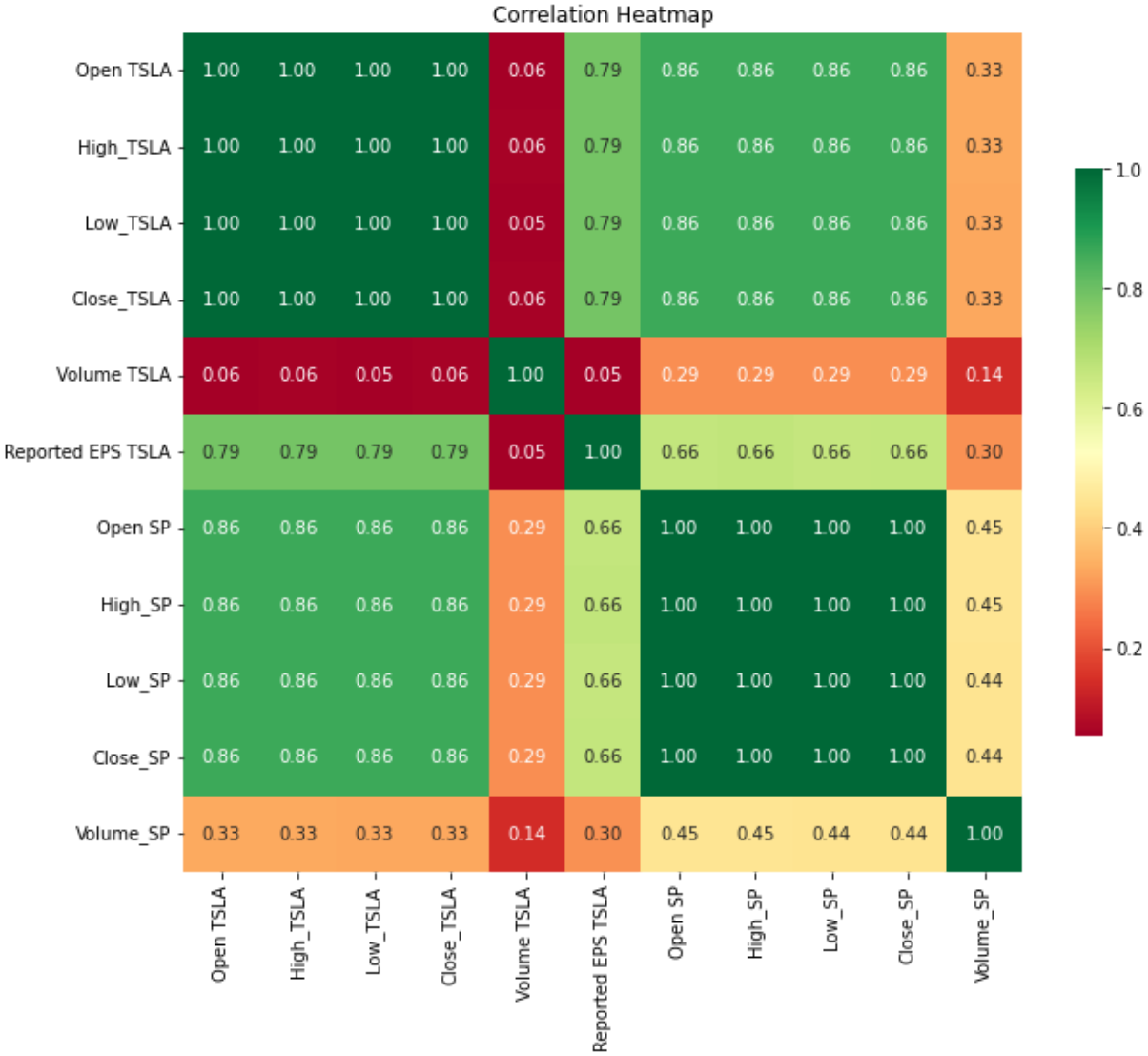


Again, we observe an almost perfect correlation between the different prices, as could be expected. We also notice an almost inexistent correlation between trading volume and prices, similarly to what happened with the KO stock, and we may provide the same interpretation: it is possible that other factors such as earnings reports, news events, and broader market conditions play a larger role in determining TSLA's stock price.

The reported EPS and prices are significantly positively correlated, which we explained previously that is to be expected. Furthermore, in this case, there is an almost vanishing correlation between trading volume and reported EPS, in contrast with the little correlation we found for the KO stock. This may indicate that the company's profitability does not seem to significantly impact the amount of shares being traded on a daily basis, with factors like market sentiment, news events impacting trading volume more.

Since TSLA has not been an S&P500 company during the whole timeframe we are studying, we could expect somewhat less correlation with S&P500 data than in the case of KO stock. A correlation heatmap shows that this is indeed the case:

In [41]:
```
selected_data = pd.DataFrame({
    'Open TSLA': df_TSLA['Open'],
    'High_TSLA': df_TSLA['High'],
    'Low_TSLA': df_TSLA['Low'],
    'Close_TSLA': df_TSLA['Close'],
    'Volume TSLA': df_TSLA['Volume'],
    'Reported EPS TSLA': df_TSLA['Reported EPS'],
    'Open SP': df_SP['Open'],
    'High_SP': df_SP['High'],
    'Low_SP': df_SP['Low'],
    'Close_SP': df_SP['Close'],
    'Volume_SP': df_SP['Volume']
})

# calculate correlation matrix
corr = selected_data.corr()

# plot the heatmap
plt.figure(figsize=(10, 10))
sns.heatmap(corr, annot=True, fmt=".2f", square=True, cmap='RdYlGn', cbar_kws={"shr
plt.title("Correlation Heatmap")
plt.tight_layout()
plt.show()
```



TSLA Price and SP500 Price (0.86): this high correlation indicates that TSLA's stock price moves in the same direction as the S&P 500 index. This could imply that TSLA, like many other large companies, is influenced by the same macroeconomic factors that drive the overall market.

Volumes of TSLA and SP500 (0.14): we find a weak relationship between the trading volume of TSLA and the overall market. This could be due to the specific factors that affect TSLA's trading volume, such as news about the company or sentiment about the electric vehicle industry, which may not necessarily influence the entire market's volume.

SP500 Volume and TSLA Price (0.33): a moderate correlation suggesting that when there is more trading in the broader market, it tends to somewhat coincide with higher TSLA prices. However, the correlation is not very strong, so other factors are also significantly influencing TSLA's price.

TSLA Volume and SP500 Price (0.29): similar to the above, this moderate correlation indicates that when the broader market is doing well (higher S&P500 prices), there might be slightly more trading activity in TSLA.

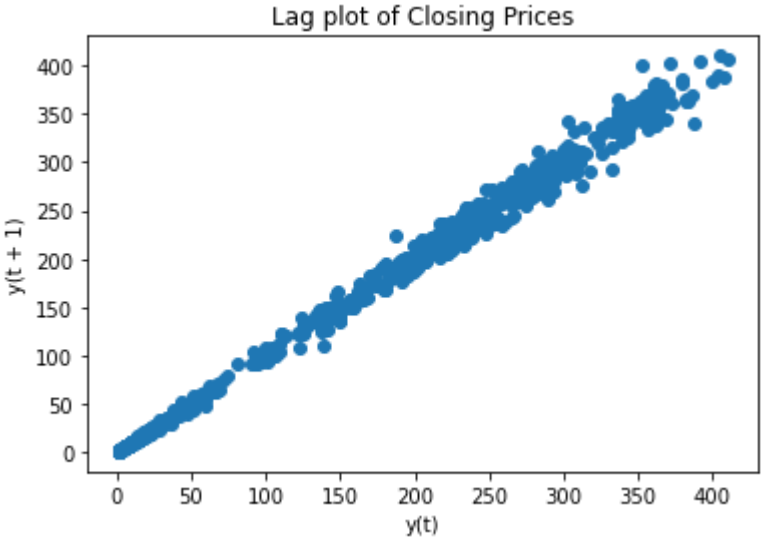
TSLA Reported EPS and SP500 Volume (0.30): this weak correlation suggests that when TSLA's earnings are higher, there might be slightly more trading activity in the broader market. This could be due to investor sentiment or reactions to TSLA's earnings, considering its status as a major technology company.

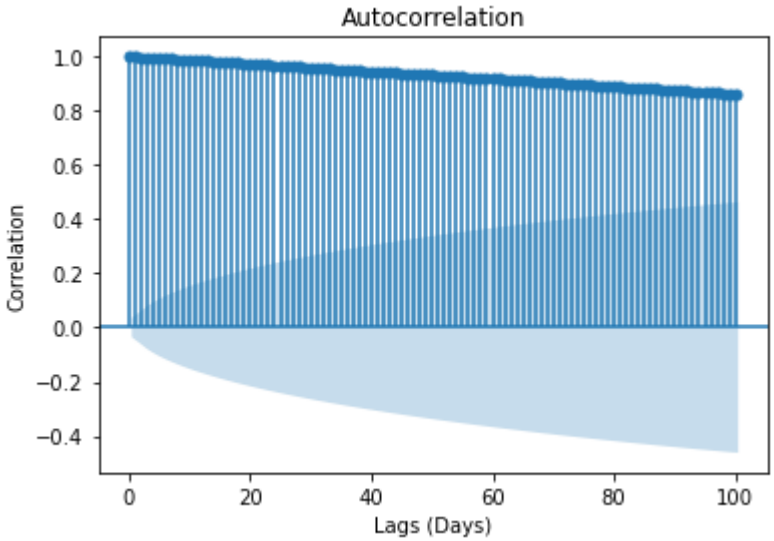
TSLA Reported EPS and SP500 prices (0.66): this rather significant correlation indicates that TSLA's earnings performance is closely tied to broader market trends. When the overall market (S&P500) is doing well, TSLA tends to report higher earnings. This makes sense as broader market conditions can influence consumer spending.

Lag analytsis

We shall now perform the lag analysis for the TSLA time series and plot the lag and autocorrelation plots.

In [59]:
```
# Create a lag plot
plt.figure() # create a new figure for the plot
lag_plot(df_TSLA['Close']) # create a lag plot using the 'Close' column of the Data
plt.title('Lag plot of Closing Prices')
plt.show()

plot_acf(df_TSLA['Close'], lags=100)
plt.xlabel('Lags (Days)', fontsize=10)
plt.ylabel('Correlation', fontsize=10)
plt.show()
```





In this case we obtain a lag plot where the data points cluster along the diagonal and gradually spread out, forming a funnel shape. This suggests that the stock price series may have a characteristic known as heteroscedasticity: essentially this means that the volatility of the stock price may be increasing over time. This was actually the conclusion from the data visualization step: after a period of 10 years of nearly constant value of the share price, a growing trend followed by a contracting phase happened, with a succession of several high peaks indicating large variations of the price in short periods of time. Traditional analysis of time series do not deal with situations with heteroscedasticity, so this is an instance where machine learning models can prove to be particularly helpful.

Similarly to what we found for the KO stock, we find that the autocorrelations for small lags are large and positive, then slowly decrease as the lags increase, which signals a trend pattern and non-stationarity.

In [ ]:

Section 3: Feature Engineering

We shall focus now on the feature engineering part of our project. We will compute financial indicators based on stock prices and earnings per share that will be considered as new features and added to our dataframe. Subsequently, we will perform a rescaling of the numerical values in our dataframe and separate it into a train and test set. It is important to notice that we are dealing with a time series, so the usual random splitting into train and test sets is not suitable here.

Financial indicators as features

We are going to introduce a series of financial indicators that are typically used in finance and add them to our dataframe as new features. As we will need to compute them for two stocks, it is useful to define functions that perform the computation in order to avoid repeating our code. We will introduce them one by one along with the corresponding code.

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import RobustScaler
import matplotlib.pyplot as plt
import seaborn as sns
import joblib
```

In finance, a momentum oscillator is an indicator that issues a signal when a price move or trend is about to start. It can fluctuate between an upper and lower band or across a zero line, highlighting relative strength or weakness within a specific time frame. Momentum oscillators are used to identify potential overbought or oversold conditions in a market, oscillating between 0 and 100. They can provide signals that suggest when an asset's price is deviating from its underlying trend, which may suggest a price correction or reversal. We will be dealing with different types of oscillators in the following.

Relative Strength Index (RSI)

Relative Strength Index is a momentum oscillator that measures the speed and change of price movements. It is primarily used to identify overbought or oversold conditions in a trading instrument.

The RSI calculation is based on 14 periods, which can be intraday, daily, weekly, or monthly price data. The result is represented as a value between 0 and 100. An RSI of 70 or above typically indicates that a stock is overbought, and could be primed for a trend reversal or corrective pullback in price. An RSI of 30 or below indicates that a stock may be oversold,

and could also be primed for a trend reversal or corrective price bounce. An RSI of 50 is considered neutral.

For its computation, one first computes the average gain and average loss. For a given timeframe (typically 14 periods), we compute the average of price gains and losses during upward and downward movements respectively. Then one calculates the Relative Strength (RS), the average gain divided by the average loss. The RSI is defined as RSI = 100 - (100 / (1 + RS)).

In [2]:
```python
# Create a function that computes the RSI and adds it to our dataframe

def rsi(data, period=14):
    '''
    Calculate the Relative Strength Index (RSI) of a given pandas DataFrame.

    Parameters:
    data: DataFrame, the input pandas DataFrame.
    period: int, the period over which to calculate the RSI. Default is 14.

    Returns:
    data: DataFrame, the DataFrame with the added RSI column.
    '''

    # Ensure data is a pandas DataFrame
    if not isinstance(data, pd.DataFrame):
        raise TypeError("Data should be a pandas DataFrame.")

    # Calculate the daily price changes
    data['Price Change'] = data['Close'].diff()

    # Split price changes into positive and negative lists
    positive_changes = data['Price Change'].apply(lambda x: x if x > 0 else 0)
    negative_changes = data['Price Change'].apply(lambda x: abs(x) if x < 0 else 0)

    # Calculate the average gains and losses over the period
    average_gain = positive_changes.rolling(window=period, min_periods=1).mean()
    average_loss = negative_changes.rolling(window=period, min_periods=1).mean()

    # Calculate the relative strength (RS)
    relative_strength = average_gain / average_loss

    # Calculate the RSI and add it as a new column in the dataframe
    data['RSI'] = 100 - (100 / (1 + relative_strength))

    return data
```

K value

The Stochastic Oscillator is a momentum indicator that compares a particular closing price of a stock to a range of its prices over a certain period of time. The formula to calculate the Stochastic Oscillator is:

K = 100(C - L14)/(H14 - L14)

where C is the most recent closing price, L14 is the lowest price traded of the 14 previous trading sessions and H14 is the highest price traded during the same 14-day period.

The Stochastic Oscillator generates values between 0 and 100. A reading above 80 is typically considered as overbought territory, which could indicate that a price pullback is likely. On the other hand, a reading below 20 is considered as oversold territory, which could indicate that a price bounce is coming.

In [3]:
```python
# Create a function that computes the K and R values and adds them to our dataframe

def k(data, period=14):
    '''
    Calculate the K value of a given pandas DataFrame.

    Parameters:
    data: DataFrame, the input pandas DataFrame.
    period: int, the period over which to calculate the values. Default is 14.

    Returns:
    data: DataFrame, the DataFrame with added K column.
    '''

    # Ensure data is a pandas DataFrame
    if not isinstance(data, pd.DataFrame):
        raise TypeError("Data should be a pandas DataFrame.")

    # Calculate highest high and lowest low over the period
    data['Highest High'] = data['High'].rolling(window=period).max()
    data['Lowest Low'] = data['Low'].rolling(window=period).min()

    # Calculate K value
    data['K'] = ((data['Close'] - data['Lowest Low']) / (data['Highest High'] - dat

    data.drop(['Highest High', 'Lowest Low'], axis=1, inplace=True)


    return data
```

Moving Average Convergence Divergence (MACD)

A moving average (MA) is a tool in technical analysis that helps smooth out price data by creating a constantly updated average price. The average is taken over a specific period of time, which is the trader's choice.

In technical analysis, the moving average can serve multiple purposes:

1. It can act as a simple trading signal when prices cross the moving average.
2. It can show a stock's price trend over a given period.

3. It can indicate a potential trend reversal when the moving average line itself changes direction.

The Exponential Moving Average (EMA) is a type of moving average that places a greater weight and significance on the most recent data points. The EMA is often used in time series analysis to smooth out short-term fluctuations and highlight longer-term trends or cycles. It is especially useful in volatile markets where price can fluctuate a lot, as it helps filter out the 'noise'.

The formula for EMA is:

EMA = (Closing price - EMA(previous day)) x multiplier + EMA(previous day)

The multiplier is calculated as follows: 2 / (selected time period + 1). The first value of the EMA serie sis typically computed as the Simple Mean Average over the specified period of time.

The Moving Average Convergence Divergence (MACD) is a momentum oscillator designed to reveal changes in the strength, direction, momentum, and duration of a trend in a stock's price. The MACD consists of two lines – the MACD line and the signal line – and a histogram. The MACD line is calculated by subtracting the 26-day EMA from the 12-day EMA. The signal line, which is the 9-day EMA of the MACD line, is then plotted on top of the MACD line. The MACD histogram is the difference MACD Histogram = MACD Line - Signal Line.

MACD can be interpreted along the following guidelines:

1. Crossovers: When the MACD line crosses above the signal line, it generates a bullish signal (time to buy). Conversely, when the MACD line crosses below the signal line, it's a bearish signal (time to sell). The Histogram aids in visualizing when these crossovers occur.

2. Divergence: When the price of an asset is moving in the opposite direction of the MACD histogram, it signals the end of the current trend.

3. Dramatic Rise: If the MACD rises dramatically, it means the asset is overbought and will likely return to normal soon.

4. A higher value of the MACD Histogram (either positive or negative) indicates a stronger level of momentum in the direction of the trend.

In [4]:
```
def macd(data, short_period=12, long_period=26, signal_period=9):
    '''
    Calculate the Moving Average Convergence Divergence (MACD) of a given pandas Da

    Parameters:
    data: DataFrame, the input pandas DataFrame.
    short_period: int, the period over which to calculate the short-term EMA. Defau
    long_period: int, the period over which to calculate the long-term EMA. Default
```

```
    signal_period: int, the period over which to calculate the signal EMA. Default

    Returns:
    data: DataFrame, the input pandas DataFrame with added MACD columns.
    '''

    # Ensure data is a pandas DataFrame
    if not isinstance(data, pd.DataFrame):
        raise TypeError("Data should be a pandas DataFrame.")

    # Calculate the short-term EMA
    data['EMA12'] = data['Close'].ewm(span=short_period, adjust=False).mean()

    # Calculate the long-term EMA
    data['EMA26'] = data['Close'].ewm(span=long_period, adjust=False).mean()

    # Calculate the MACD line
    data['MACD Line'] = data['EMA12'] - data['EMA26']

    # Calculate the signal line (9-day EMA of the MACD line)
    data['Signal Line'] = data['MACD Line'].ewm(span=signal_period, adjust=False).m

    # Calculate the MACD Histogram
    data['MACD Histogram'] = data['MACD Line'] - data['Signal Line']

    return data
```

Rate of Change (ROC)

The Rate of Change (ROC) is a financial indicator that measures the percentage change in price between the current price and the price a certain number of periods ago. The ROC is classified as a momentum oscillator, measuring the velocity of a stock's price movement. It can be used to identify price trends and also to spot potential buy and sell signals.

The ROC is calculated as follows:

ROC = (Current Price - Price n periods ago) / (Price n periods ago)* 100

where n represents the number of periods you are comparing.

The ROC oscillator moves around the zero line. If the ROC is rising, it gives a bullish signal, while a falling ROC gives a bearish signal. When the ROC is positive it suggests the price is trending upwards, and when it is negative it suggests a downward price trend.

In [5]:
```
def roc(data, period=9):
    '''
    Calculate the Rate of Change (ROC) of a given pandas DataFrame.

    Parameters:
    data: DataFrame, the input pandas DataFrame.
    period: int, the period over which to calculate the ROC. Default is 9.
```

```
    Returns:
    data: DataFrame, the input pandas DataFrame with the added ROC column.
    '''

    # Ensure data is a pandas DataFrame
    if not isinstance(data, pd.DataFrame):
        raise TypeError("Data should be a pandas DataFrame.")

    # Calculate the ROC value
    data['ROC'] = (data['Close'].pct_change(period) * 100)

    return data
```

On-Balance Volume (OBV)

On-balance volume (OBV) is a momentum indicator that uses volume flow to predict changes in stock price. OBV provides a running total of volume and shows whether this volume is flowing in or out of a given security. The idea behind is that when volume increases sharply without a significant change in the stock's price, the price will eventually jump upward, and vice versa.

OBV is calculated using a cumulative total volume, which adds or subtracts each period's volume depending on the price movement. This is done by comparing the closing prices of two consecutive periods (days):

If today's close is higher than yesterday's close, then: OBV = Yesterday's OBV + Today's Volume If today's close is lower than yesterday's close, then: OBV = Yesterday's OBV - Today's Volume If today's close equals yesterday's close, then: OBV = Yesterday's OBV (i.e., volume is added to the OBV total only if the closing price was up for the day)

Therefore, when prices are up, volume is added to the running total, and when prices are down, volume is subtracted from the running total. This is supposed to highlight when volume is flowing into a security versus when volume is flowing out of a security.

Here is a general interpretation of the OBV:

1. Bullish Scenario: If the price is rising (uptrend) and the OBV is rising too, this is a positive indicator that shows strong buyer interest. Traders might interpret this as a signal that the upward trend is likely to continue.

2. Bearish Scenario: If the price is falling (downtrend) and the OBV is falling too, this shows that selling volume is increasing. Traders might consider this as a signal that the downward trend is likely to continue.

3. Market Divergence: If the price is rising but OBV is falling, this could indicate that the price rise is not backed by strong buying and might soon end (bearish divergence). On the other hand, if the price is falling, but OBV is rising, this could suggest that the price fall is not backed by strong selling and might soon end (bullish divergence).

In [6]:
```python
def obv(data):
    '''
    Calculate the On-Balance Volume (OBV) of a given pandas DataFrame.

    Parameters:
    data: DataFrame, the input pandas DataFrame.

    Returns:
    data: DataFrame, the input pandas DataFrame with an added OBV column.
    '''

    # Ensure data is a pandas DataFrame
    if not isinstance(data, pd.DataFrame):
        raise TypeError("Data should be a pandas DataFrame.")

    # Initialize the OBV value
    data['OBV'] = 0

    # Calculate the OBV value for each period
    for i in range(1, len(data)):
        if data['Close'].iloc[i] > data['Close'].iloc[i-1]:
            data['OBV'].iloc[i] = data['OBV'].iloc[i-1] + data['Volume'].iloc[i]
        elif data['Close'].iloc[i] < data['Close'].iloc[i-1]:
            data['OBV'].iloc[i] = data['OBV'].iloc[i-1] - data['Volume'].iloc[i]
        else:
            data['OBV'].iloc[i] = data['OBV'].iloc[i-1]

    return data
```

Price to Earnings ratio (P/E)

Price to Earnings ratio (P/E) is a valuation ratio, calculated by dividing the stock price per share by the earnings per share (EPS) over a specific period. The P/E ratio measures the price paid for a share relative to the earnings that the company is generating. It is a widely used indicator of a company's earnings potential.

Here's how to interpret the P/E ratio:

High P/E: A high P/E ratio could mean that a company's stock is over-priced, or else that investors are expecting high growth rates in the future. Companies with high growth potential often have higher P/E ratios, as investors are willing to pay a premium for the possibility of high future earnings.

Low P/E: Conversely, a low P/E ratio could indicate that the company may currently be undervalued, or it could also indicate that the company is doing exceptionally well compared to its past trends. However, it could also suggest that the market has serious doubts about the company's prospects.

Negative P/E: A negative P/E ratio means the company is losing money. It could be a warning sign to investors, but it might also be a result of the company investing in its own

growth.

In [7]:
```python
def pe(data, close_column="Close", eps_column="Reported EPS"):
    """
    Calculate the P/E ratio of a given pandas DataFrame.

    Parameters:
    df: DataFrame, the input pandas DataFrame.
    close_column: str, the name of the closing price column. Default is "Close".
    eps_column: str, the name of the EPS column. Default is "Reported EPS".

    Returns:
    df: DataFrame, the input DataFrame with an added P/E ratio column.
    """

    # Ensure df is a pandas DataFrame
    if not isinstance(data, pd.DataFrame):
        raise TypeError("Data should be a pandas DataFrame.")

    # Calculate the P/E ratio
    data["P/E"] = data[close_column] / data[eps_column]

    return data
```

Coca-Cola stock

We will start by computing the financial indicators for the KO stock. Note that for RSI, K and ROC indicators the first entries will be NaN because of their definition, so we will drop these entries after we have computed all the indicators.

First we import our KO dataframe:

In [8]:
```python
# Read the csv file containing the data for the KO stock.
# index_col=0 tells pandas to use the first column as the index.
# parse_dates=True tells pandas to interpret the index as a DateTimeIndex.
df_KO = pd.read_csv('KO.csv', parse_dates=True, index_col=0)
```

Now we successively compute the RSI, K, MACD Line and Signal Line, ROC, OBV and P/E:

In [9]:
```python
rsi(df_KO)
```

Out[9]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS	Pri Chang
1996-04-17	20.281250	20.281250	19.843750	20.031250	10.141947	8906000	0.14	Na
1996-04-18	20.031250	20.156250	19.843750	19.875000	10.062837	9608000	0.14	-0.1562
1996-04-19	19.875000	20.062500	19.562500	19.687500	9.967900	13010400	0.14	-0.1875
1996-04-22	19.875000	20.187500	19.875000	20.156250	10.205229	7160800	0.14	0.4687
1996-04-23	20.156250	20.281250	20.062500	20.250000	10.252701	6218800	0.14	0.0937
...	...	...	...	...	...	...	...	
2022-12-23	63.500000	63.869999	63.200001	63.820000	62.855492	6463300	0.69	0.4800
2022-12-27	63.930000	64.290001	63.709999	64.209999	63.239597	7320700	0.69	0.3899
2022-12-28	64.459999	64.650002	63.490002	63.570000	62.609272	7159400	0.69	-0.6399
2022-12-29	63.799999	64.150002	63.700001	63.950001	62.983532	7169300	0.69	0.3800
2022-12-30	63.919998	63.919998	63.169998	63.610001	62.648666	7650200	0.69	-0.3400

6724 rows × 9 columns

In [10]:
```
k(df_KO)
```

Out[10]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS	Pri Chang
1996-04-17	20.281250	20.281250	19.843750	20.031250	10.141947	8906000	0.14	Na
1996-04-18	20.031250	20.156250	19.843750	19.875000	10.062837	9608000	0.14	-0.1562
1996-04-19	19.875000	20.062500	19.562500	19.687500	9.967900	13010400	0.14	-0.1875
1996-04-22	19.875000	20.187500	19.875000	20.156250	10.205229	7160800	0.14	0.4687
1996-04-23	20.156250	20.281250	20.062500	20.250000	10.252701	6218800	0.14	0.0937
...	...	...	...	...	...	...	...	
2022-12-23	63.500000	63.869999	63.200001	63.820000	62.855492	6463300	0.69	0.4800
2022-12-27	63.930000	64.290001	63.709999	64.209999	63.239597	7320700	0.69	0.3899
2022-12-28	64.459999	64.650002	63.490002	63.570000	62.609272	7159400	0.69	-0.6399
2022-12-29	63.799999	64.150002	63.700001	63.950001	62.983532	7169300	0.69	0.3800
2022-12-30	63.919998	63.919998	63.169998	63.610001	62.648666	7650200	0.69	-0.3400

6724 rows × 10 columns

In [11]:
```
macd(df_KO)
```

Out[11]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS	Pri Chang
1996-04-17	20.281250	20.281250	19.843750	20.031250	10.141947	8906000	0.14	Na
1996-04-18	20.031250	20.156250	19.843750	19.875000	10.062837	9608000	0.14	-0.1562
1996-04-19	19.875000	20.062500	19.562500	19.687500	9.967900	13010400	0.14	-0.1875
1996-04-22	19.875000	20.187500	19.875000	20.156250	10.205229	7160800	0.14	0.4687
1996-04-23	20.156250	20.281250	20.062500	20.250000	10.252701	6218800	0.14	0.0937
...	...	...	...	...	...	...	...	
2022-12-23	63.500000	63.869999	63.200001	63.820000	62.855492	6463300	0.69	0.4800
2022-12-27	63.930000	64.290001	63.709999	64.209999	63.239597	7320700	0.69	0.3899
2022-12-28	64.459999	64.650002	63.490002	63.570000	62.609272	7159400	0.69	-0.6399
2022-12-29	63.799999	64.150002	63.700001	63.950001	62.983532	7169300	0.69	0.3800
2022-12-30	63.919998	63.919998	63.169998	63.610001	62.648666	7650200	0.69	-0.3400

6724 rows × 15 columns

In [12]:
```
roc(df_KO)
```

Out[12]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS	Pri Chang
1996-04-17	20.281250	20.281250	19.843750	20.031250	10.141947	8906000	0.14	Na
1996-04-18	20.031250	20.156250	19.843750	19.875000	10.062837	9608000	0.14	-0.1562
1996-04-19	19.875000	20.062500	19.562500	19.687500	9.967900	13010400	0.14	-0.1875
1996-04-22	19.875000	20.187500	19.875000	20.156250	10.205229	7160800	0.14	0.4687
1996-04-23	20.156250	20.281250	20.062500	20.250000	10.252701	6218800	0.14	0.0937
...	...	...	...	...	...	...	...	
2022-12-23	63.500000	63.869999	63.200001	63.820000	62.855492	6463300	0.69	0.4800
2022-12-27	63.930000	64.290001	63.709999	64.209999	63.239597	7320700	0.69	0.3899
2022-12-28	64.459999	64.650002	63.490002	63.570000	62.609272	7159400	0.69	-0.6399
2022-12-29	63.799999	64.150002	63.700001	63.950001	62.983532	7169300	0.69	0.3800
2022-12-30	63.919998	63.919998	63.169998	63.610001	62.648666	7650200	0.69	-0.3400

6724 rows × 16 columns

In [13]:
```
obv(df_KO)
```

```
/home/sergio/anaconda3/lib/python3.9/site-packages/pandas/core/indexing.py:1732: Set
tingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  self._setitem_single_block(indexer, value, name)
```

Out[13]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS	Pri Chang
1996-04-17	20.281250	20.281250	19.843750	20.031250	10.141947	8906000	0.14	Na
1996-04-18	20.031250	20.156250	19.843750	19.875000	10.062837	9608000	0.14	-0.1562
1996-04-19	19.875000	20.062500	19.562500	19.687500	9.967900	13010400	0.14	-0.1875
1996-04-22	19.875000	20.187500	19.875000	20.156250	10.205229	7160800	0.14	0.4687
1996-04-23	20.156250	20.281250	20.062500	20.250000	10.252701	6218800	0.14	0.0937
...	...	...	...	...	...	...	...	
2022-12-23	63.500000	63.869999	63.200001	63.820000	62.855492	6463300	0.69	0.4800
2022-12-27	63.930000	64.290001	63.709999	64.209999	63.239597	7320700	0.69	0.3899
2022-12-28	64.459999	64.650002	63.490002	63.570000	62.609272	7159400	0.69	-0.6399
2022-12-29	63.799999	64.150002	63.700001	63.950001	62.983532	7169300	0.69	0.3800
2022-12-30	63.919998	63.919998	63.169998	63.610001	62.648666	7650200	0.69	-0.3400

6724 rows × 17 columns

In [14]:
```
pe(df_KO)
```

Out[14]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS	Pri Chang
1996-04-17	20.281250	20.281250	19.843750	20.031250	10.141947	8906000	0.14	Na
1996-04-18	20.031250	20.156250	19.843750	19.875000	10.062837	9608000	0.14	-0.1562
1996-04-19	19.875000	20.062500	19.562500	19.687500	9.967900	13010400	0.14	-0.1875
1996-04-22	19.875000	20.187500	19.875000	20.156250	10.205229	7160800	0.14	0.4687
1996-04-23	20.156250	20.281250	20.062500	20.250000	10.252701	6218800	0.14	0.0937
...	...	...	...	...	...	...	...	
2022-12-23	63.500000	63.869999	63.200001	63.820000	62.855492	6463300	0.69	0.4800
2022-12-27	63.930000	64.290001	63.709999	64.209999	63.239597	7320700	0.69	0.3899
2022-12-28	64.459999	64.650002	63.490002	63.570000	62.609272	7159400	0.69	-0.6399
2022-12-29	63.799999	64.150002	63.700001	63.950001	62.983532	7169300	0.69	0.3800
2022-12-30	63.919998	63.919998	63.169998	63.610001	62.648666	7650200	0.69	-0.3400

6724 rows × 18 columns

As mentioned earlier, some indicators start with NaN values because of their definition. We drop these rows:

In [15]:
```
# Drop rows with NaN values
df_KO = df_KO.dropna()

df_KO
```

Out[15]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS	Price Change
1996-05-06	19.937500	20.218750	19.750000	20.218750	10.236874	7170400	0.14	0.28125(
1996-05-07	20.218750	20.406250	20.156250	20.343750	10.300159	6702800	0.14	0.12500(
1996-05-08	20.343750	20.687500	20.062500	20.687500	10.474213	8292800	0.14	0.34375(
1996-05-09	20.687500	20.937500	20.593750	20.687500	10.474213	4820400	0.14	0.00000(
1996-05-10	20.718750	20.968750	20.718750	20.968750	10.616608	4942800	0.14	0.28125(
...	...	...	...	...	...	...	...	.
2022-12-23	63.500000	63.869999	63.200001	63.820000	62.855492	6463300	0.69	0.48000(
2022-12-27	63.930000	64.290001	63.709999	64.209999	63.239597	7320700	0.69	0.38999(
2022-12-28	64.459999	64.650002	63.490002	63.570000	62.609272	7159400	0.69	-0.63999(
2022-12-29	63.799999	64.150002	63.700001	63.950001	62.983532	7169300	0.69	0.38000(
2022-12-30	63.919998	63.919998	63.169998	63.610001	62.648666	7650200	0.69	-0.34000(

6711 rows × 18 columns

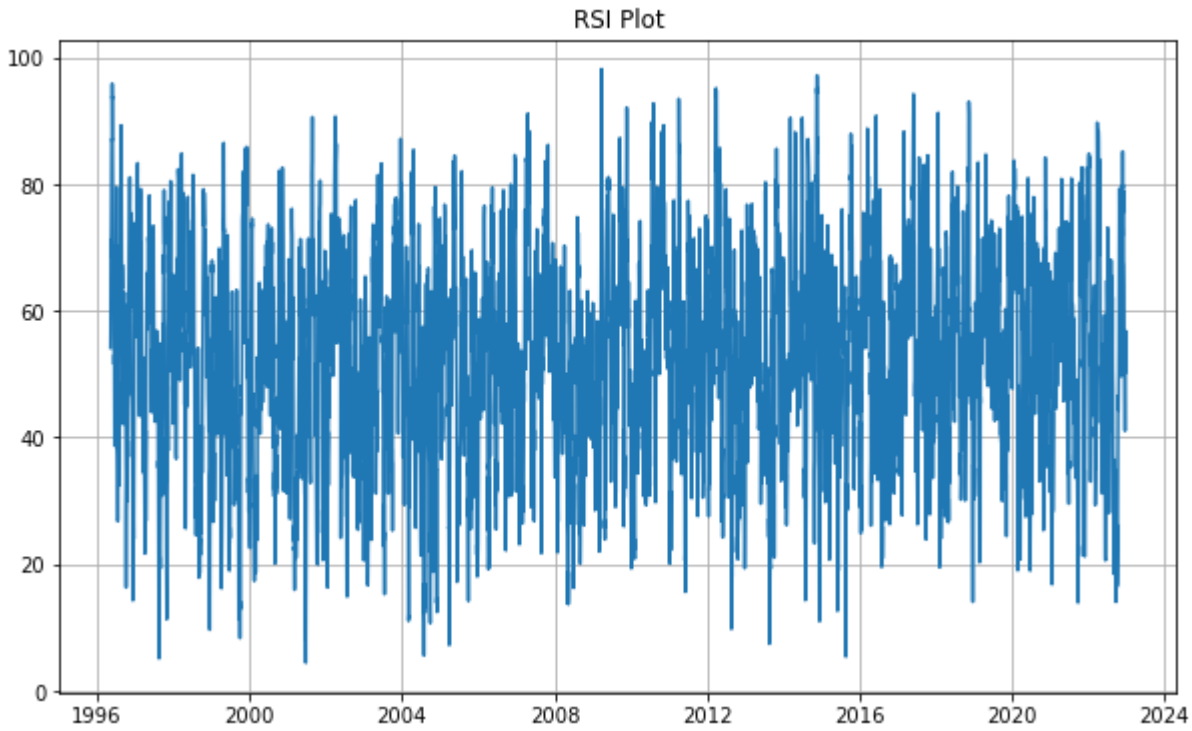
In [16]:
```
df_KO.to_csv('df_KO_.csv')
```

Plots

It is convenient to plot the series of the newly defined indicators to illustrate their behavior. We will create functions for plotting these indicators as we go.

In [17]:
```
def plot_rsi(df, rsi_col='RSI'):
    plt.figure(figsize=(10,6))
    plt.plot(df.index, df[rsi_col])
    plt.title('RSI Plot')
    plt.grid(True)
    plt.show()
```

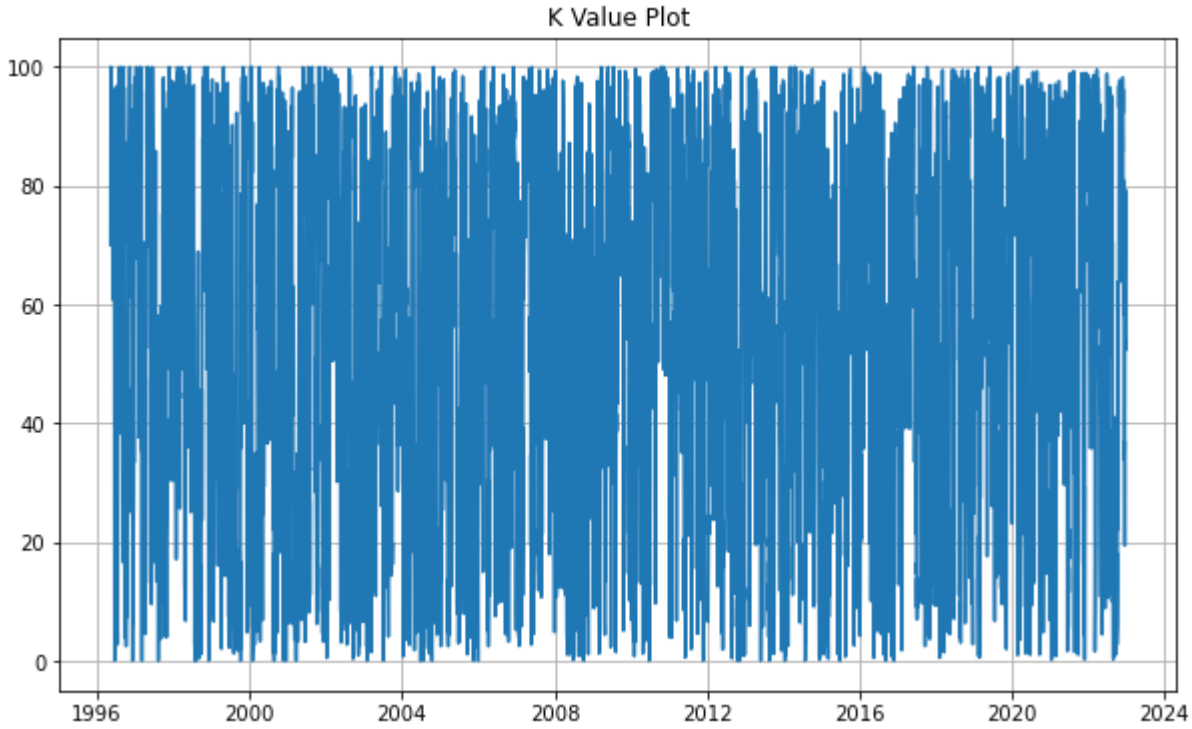
In [18]:
```
plot_rsi(df_KO)
```



In [ ]:

In [19]:
```
def plot_K(df, k_col='K'):
    plt.figure(figsize=(10,6))
    plt.plot(df.index, df[k_col])
    plt.title('K Value Plot')
    plt.grid(True)
    plt.show()
```

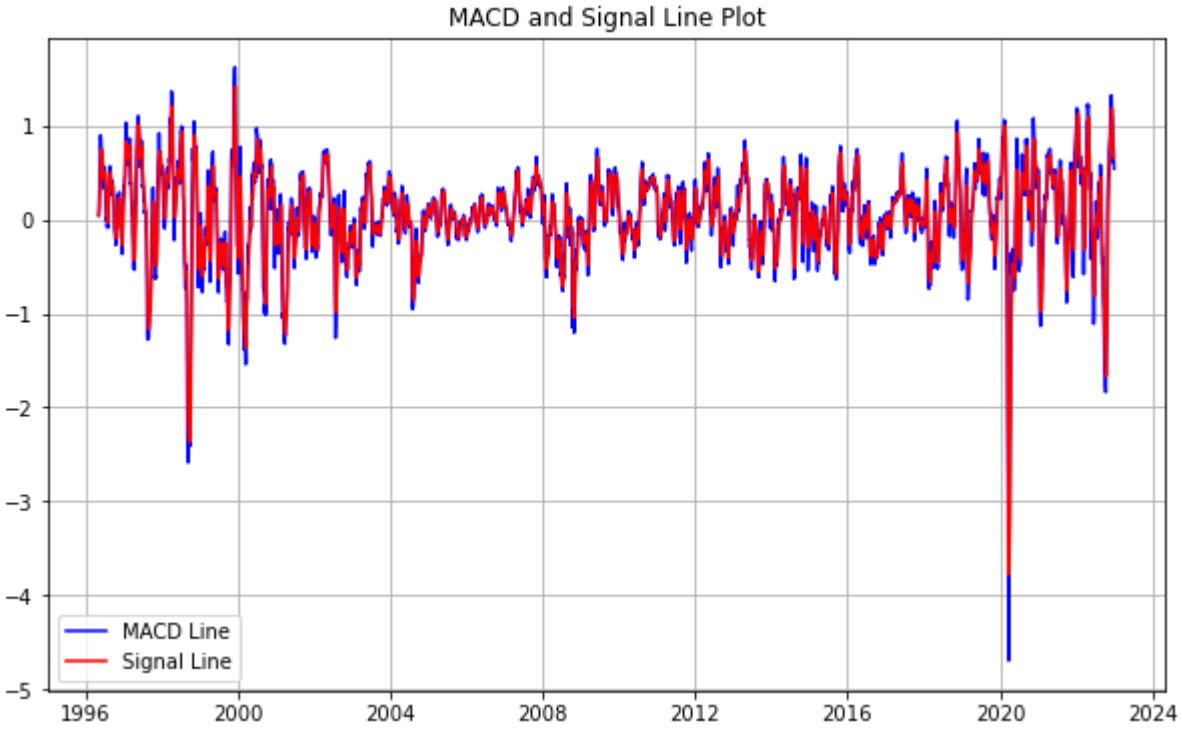
In [20]:
```
plot_K(df_KO)
```



Although the threshold values are frequently crossed, the RSI and K value plots oscillate rather uniformly in the long run, indicating that the market is experiencing regular periods of buying and selling pressure. The K value plot is denser than the RSI plot: this is a common feature as the K value tends to move more frequently.

In [21]:
```
def plot_MACD_and_Signal(df, macd_col='MACD Line', signal_col='Signal Line'):
    plt.figure(figsize=(10,6))
    plt.plot(df.index, df[macd_col], label='MACD Line', color='blue')
    plt.plot(df.index, df[signal_col], label='Signal Line', color='red')
    plt.title('MACD and Signal Line Plot')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()
```

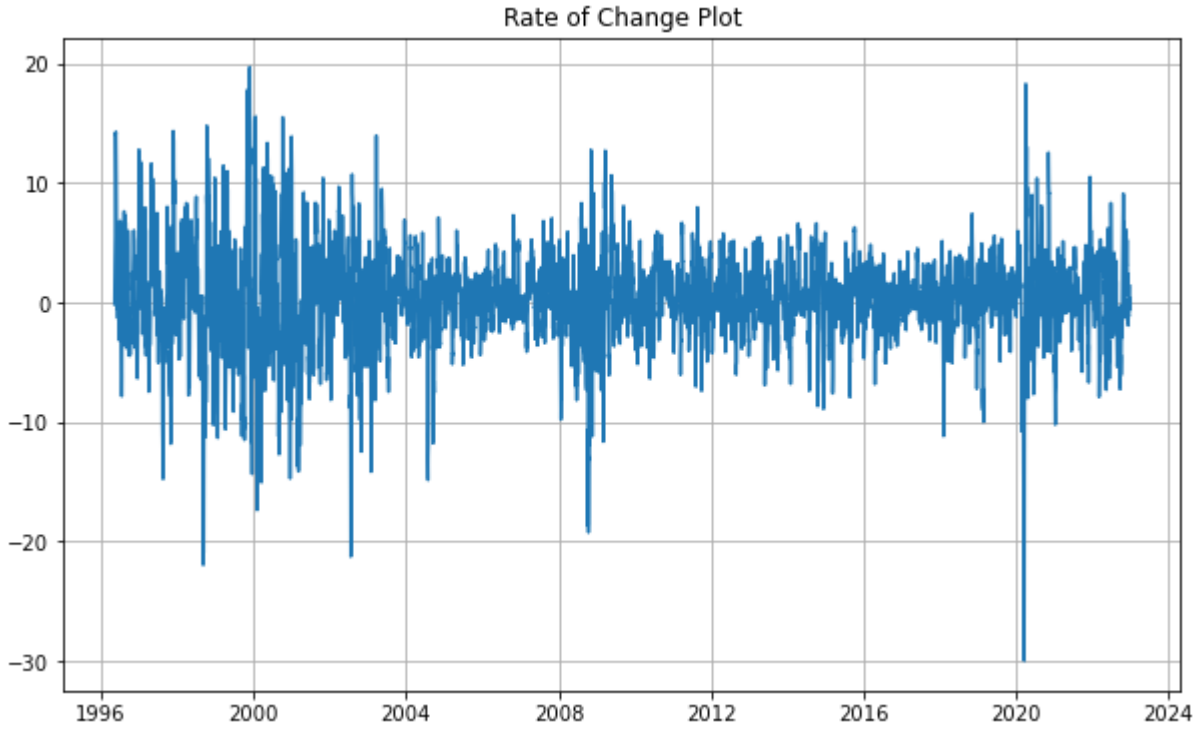
In [22]:
```
plot_MACD_and_Signal(df_KO)
```



We notice a few instances where the MACD line crosses above the Signal line indicating a bullish signal, most noticeably at the end of 1999 and during 2020. The last one can be interpreted as the result of the combination of a drop in the prices due to the COVID pandemic and the bullish trend of this particular stock. The most prominent instance where the MACD line crossed below the Signal line, indicating time to sell, was earlier in 2020 due to the COVID pandemic.

In [23]:
```
def plot_ROC(df, roc_col='ROC'):
    plt.figure(figsize=(10,6))
    plt.plot(df.index, df[roc_col])
    plt.title('Rate of Change Plot')
    plt.grid(True)
    plt.show()
```

In [24]:
```
plot_ROC(df_KO)
```



The ROC plot shows a few points above +10 and below -10, indicating strong bullish and bearish signs. Perhaps the most prominant instance is the dramatic drop corresponding to the onset of the COVID pandemic, followed by a peak indicating time to buy a stock with an upward trend at a cheaper price.

In [25]:
```
def plot_OBV(df, obv_col='OBV'):
    plt.figure(figsize=(10,6))
    plt.plot(df.index, df[obv_col])
    plt.title('On Balance Volume Plot')
    plt.grid(True)
    plt.show()
```

In [26]:
```
plot_OBV(df_KO)
```

This OBV distribution is very similar to the Close Price plot from the late 90s. There is a sign of bearish divergence from 1996 to 1998 as prices were rising but the OBV was rather stagnant. Indeed, a bearish trend followed from 1998 until around 2005, as we can check in the Close Price plot.

In [27]:
```
def plot_PE(df, pe_col='P/E'):
    plt.figure(figsize=(10,6))
    plt.plot(df.index, df[pe_col])
    plt.title('Price to Earnings Ratio Plot')
    plt.grid(True)
    plt.show()
```

In [28]:
```
plot_PE(df_KO)
```

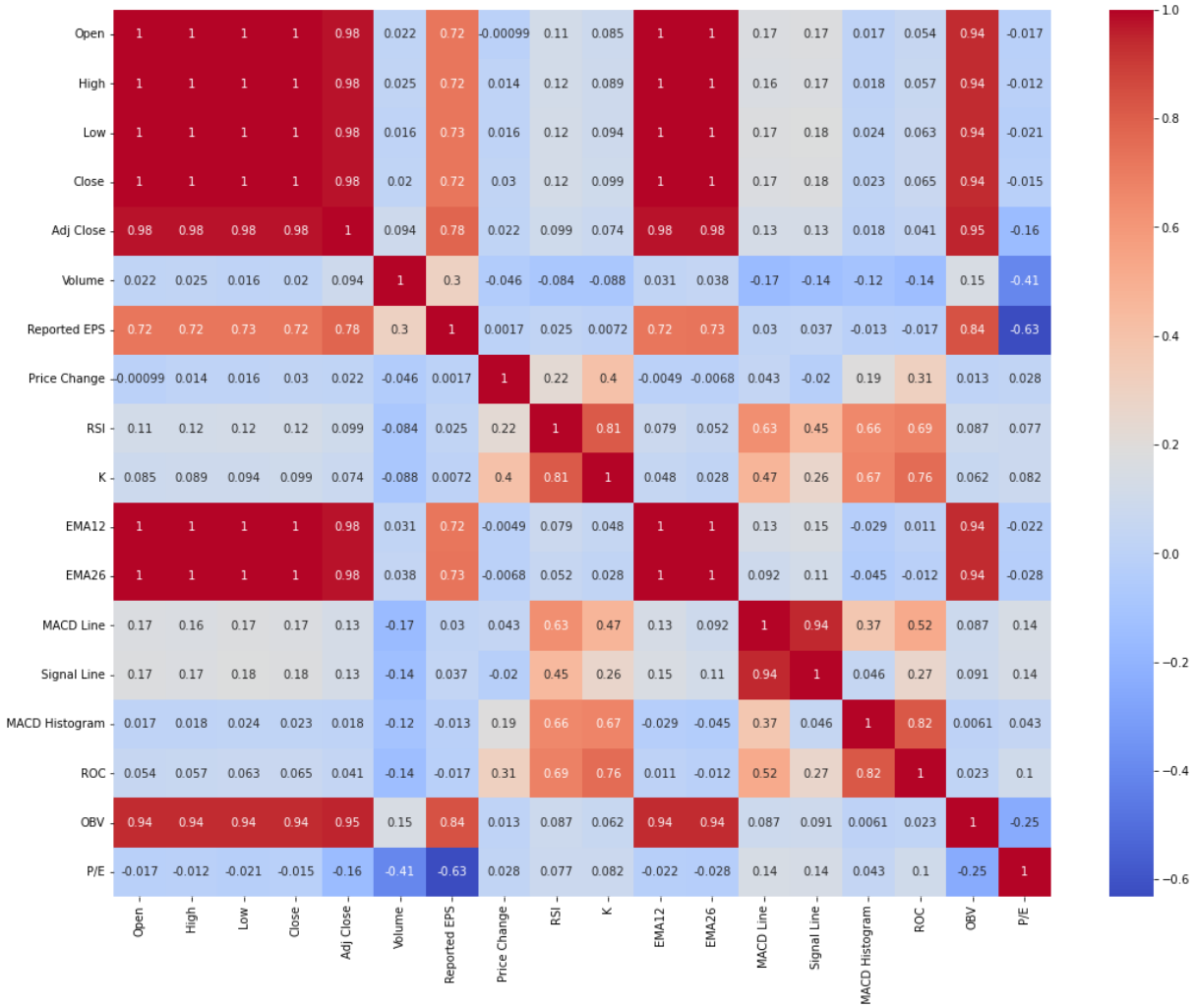
The P/E plot shows a strong seasonality component, inherited from the earnings per share. We observe extremely high values of P/E in the late 90s, indicating that the stock was overpriced. The bearish trend that followed the late 90s for this stock can be seen as a market correction of this overpricing.

Correlations

Since we have introduced new features, we may wonder how correlated they are to the original features. This could be useful in finding out patterns but also in identifying possible redundant features. Let's plot the correlation heatmap:

In [29]:
```
# Plot a correlation heatmap
correlation_matrix = df_KO.corr()
plt.figure(figsize=(18, 14))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
```

Out[29]:
```
<AxesSubplot:>
```



We observe a high correlation between EMA12, EMA 26 lines and prices, which was to be expected because of the definitions of the EMA12 and EMA26 indicators. The entries in all the corresponding columns have indeed very similar values. The OBV indicator is also highly correlated with prices, which again is not surprising as it is directly tied to the price through its calculation. Despite being highly correlated, all these features represent different information and we shall keep them in the sequel.

Train-test split

In time-series analysis, the order of the data points is important. Therefore, unlike traditional machine learning models where we perform a random train-test split, in time-series we often use a sequential split to preserve the temporal order of the observations. This is important because future data points are often dependent on past data points in time series data. So if we do a random split, we may be using future information to predict past events, which is not the right approach.

Instead, we can split the time series into training and test sets based on a certain threshold date or a proportion of data. In our case, we will implement our code so we can choose the

percentage of data we will be using for testing. Based on this percentage, we will compute the index in our dataframe that will serve us to split the data into train and test sets.

In [30]:
```python
# Create a function that to split our dataset into train and test sets.

def train_test_split(df, test_size=0.02, target='Close'):
    '''
    Perform train-test split with respect to time series structure.

    Parameters:
    df: DataFrame, the input pandas DataFrame.
    test_size: float, the proportion of the dataset to include in the test split.
    target: str, the target variable to predict. Default is 'Close'.

    Returns:
    X_train, X_test, y_train, y_test : series, split datasets.
    '''

    # Ensure df is a pandas DataFrame
    if not isinstance(df, pd.DataFrame):
        raise TypeError("Data should be a pandas DataFrame.")

    # Calculate the index at which to split the DataFrame into training and testing
    split_index = int(len(df) * (1 - test_size))

    # Split the DataFrame into training and testing data
    train_data = df.iloc[:split_index]
    test_data = df.iloc[split_index:]

    # Separate the features and the target variable in the training and testing set
    X_train = train_data.drop([target], axis=1)
    y_train = train_data[target].to_frame()
    X_test = test_data.drop([target], axis=1)
    y_test = test_data[target].to_frame()

    return X_train, X_test, y_train, y_test
```

It is time now to perform a train/test split using the function just defined:

In [31]:
```python
# Split our dataset into train and test sets.
X_train, X_test, y_train, y_test = train_test_split(df_KO)
```

Our function `train_test_split` has been defined in a way that it returns the train and test split as datasets. Let's check this:

In [32]:
```python
X_train
```

Out[32]:

Date	Open	High	Low	Adj Close	Volume	Reported EPS	Price Change	R
1996-05-06	19.937500	20.218750	19.750000	10.236874	7170400	0.14	0.281250	54.16666
1996-05-07	20.218750	20.406250	20.156250	10.300159	6702800	0.14	0.125000	56.57894
1996-05-08	20.343750	20.687500	20.062500	10.474213	8292800	0.14	0.343750	65.85365
1996-05-09	20.687500	20.937500	20.593750	10.474213	4820400	0.14	0.000000	71.05263
1996-05-10	20.718750	20.968750	20.718750	10.616608	4942800	0.14	0.281250	68.57142
...	...	...	...	...	...	...	...	
2022-06-13	60.750000	62.290001	60.669998	59.126160	23064200	0.64	-0.070000	39.50276
2022-06-14	60.730000	60.889999	58.660000	57.504799	24620000	0.64	-2.110001	20.75706
2022-06-15	60.000000	60.639999	58.970001	57.931984	19684700	0.64	0.439999	24.35897
2022-06-16	58.639999	59.520000	58.250000	57.349461	15053800	0.64	-0.599998	20.78209
2022-06-17	59.700001	60.130001	59.130001	57.698975	34781900	0.64	0.360001	20.60470

6576 rows × 17 columns

In [33]:
```
y_test
```

Out[33]:

Date	Close
2022-06-21	60.700001
2022-06-22	61.150002
2022-06-23	61.880001
2022-06-24	63.040001
2022-06-27	62.910000
...	...
2022-12-23	63.820000
2022-12-27	64.209999
2022-12-28	63.570000
2022-12-29	63.950001
2022-12-30	63.610001

135 rows × 1 columns

Rescaling numerical features

In the next notebook we will be training our machine learning models for stock price forecasting. We will be focusing on two models: a random forest and a Long-Short-Term-Memory (LSTM) network. Random forests are not sensitive to data rescaling, however neural networks are. So we will perform a rescaling of our dataframe here.

We choose the RobustScaler scaler, as it is robust to outliers. For future convenience, we will save the scaler object of the target, as we will need to undo the scaling transformation in the next notebook after training our LSTM model. This is done with `joblib.dump`.

In [34]:
```
# Initialize the scaler
scaler_features = RobustScaler()
scaler_target = RobustScaler()

# Fit the scaler on the training data
scaler_features.fit(X_train)
scaler_target.fit(y_train)

# Save the scaler object to a file
joblib.dump(scaler_target, 'robust_scaler.pkl')

# Transform the training and testing data
X_train_scaled_np = scaler_features.transform(X_train)
X_test_scaled_np = scaler_features.transform(X_test)
```

```
# Transform the training and testing data
y_train_scaled_np = scaler_target.transform(y_train)
y_test_scaled_np = scaler_target.transform(y_test)
```

The scaler returns a numpy object. We can convert back to a dataframe object using `pd.DataFrame`:

In [35]:
```
X_train_scaled = pd.DataFrame(X_train_scaled_np, columns=X_train.columns, index=X_t
```

In [36]:
```
X_test_scaled = pd.DataFrame(X_test_scaled_np, columns=X_test.columns, index=X_test
```

In [37]:
```
y_train_scaled = pd.DataFrame(y_train_scaled_np, columns=y_train.columns, index=X_t
```

In [38]:
```
y_test_scaled = pd.DataFrame(y_test_scaled_np, columns=y_test.columns, index=X_test
```

Let's check what the scaled dataframes look like:

In [39]:
```
X_train_scaled
```

Out[39]:

Date	Open	High	Low	Adj Close	Volume	Reported EPS	Price Change	RS
1996-05-06	-0.689841	-0.693677	-0.678770	-0.422506	-0.655789	-0.888889	0.630819	0.070245
1996-05-07	-0.673631	-0.682887	-0.655356	-0.419209	-0.716175	-0.888889	0.267444	0.175655
1996-05-08	-0.666426	-0.666703	-0.660759	-0.410142	-0.510840	-0.888889	0.776169	0.580935
1996-05-09	-0.646614	-0.652316	-0.630142	-0.410142	-0.959272	-0.888889	-0.023257	0.808116
1996-05-10	-0.644813	-0.650518	-0.622938	-0.402724	-0.943465	-0.888889	0.630819	0.699694
...	...	...	...	...	...	...	...	...
2022-06-13	1.662464	1.727342	1.679562	2.124342	1.396766	0.962963	-0.186048	-0.570527
2022-06-14	1.661311	1.646778	1.563720	2.039879	1.597684	0.962963	-4.930275	-1.389663
2022-06-15	1.619236	1.632391	1.581587	2.062133	0.960331	0.962963	1.000004	-1.232269
2022-06-16	1.540850	1.567940	1.540091	2.031787	0.362288	0.962963	-1.418613	-1.388569
2022-06-17	1.601945	1.603043	1.590808	2.049994	2.910011	0.962963	0.813961	-1.396321

6576 rows × 17 columns

In [40]:
```
X_test_scaled
```

Out[40]:

Date	Open	High	Low	Adj Close	Volume	Reported EPS	Price Change	RSI	
2022-06-21	1.589842	1.646202	1.583892	2.114227	0.704669	0.962963	2.930257	-0.769740	-
2022-06-22	1.653818	1.679003	1.659391	2.136986	0.382085	0.962963	1.023265	-0.575868	-
2022-06-23	1.694164	1.705474	1.711836	2.173907	0.638277	0.962963	1.674431	-0.555514	
2022-06-24	1.740850	1.771652	1.762553	2.232577	0.761762	0.962963	2.674440	-0.095745	
2022-06-27	1.790994	1.784887	1.792522	2.226002	-0.013734	0.962963	-0.325587	-0.102666	
...	...	...	...	...	...	...	...	...	
2022-12-23	1.820965	1.818264	1.825373	2.318619	-0.747105	1.148148	1.093031	0.033830	
2022-12-27	1.845749	1.842433	1.854766	2.338629	-0.636379	1.148148	0.883726	0.188057	
2022-12-28	1.876297	1.863149	1.842087	2.305792	-0.657209	1.148148	-1.511639	-0.101169	-
2022-12-29	1.838256	1.834377	1.854189	2.325289	-0.655931	1.148148	0.860474	-0.062964	
2022-12-30	1.845173	1.821141	1.823644	2.307845	-0.593826	1.148148	-0.813961	0.061342	-

135 rows × 17 columns

In [41]:
```
y_train_scaled
```

Out[41]:

Date	Close
1996-05-06	-0.676602
1996-05-07	-0.669385
1996-05-08	-0.649538
1996-05-09	-0.649538
1996-05-10	-0.633300
...	...
2022-06-13	1.697604
2022-06-14	1.575779
2022-06-15	1.601184
2022-06-16	1.566542
2022-06-17	1.587327

6576 rows × 1 columns

In [42]:
```
y_test_scaled
```

Out[42]:

Date	Close
2022-06-21	1.660653
2022-06-22	1.686634
2022-06-23	1.728782
2022-06-24	1.795756
2022-06-27	1.788251
...	...
2022-12-23	1.840791
2022-12-27	1.863308
2022-12-28	1.826357
2022-12-29	1.848297
2022-12-30	1.828666

135 rows × 1 columns

We observe the presence of negative values for price in the `X_train_scaled` and `y_train_scaled` dataframes. This is fine for the purpose of training our models. We will just need to undo the rescaling transformation in the predicted values for a more straightforward interpretation.

Including temporal categorial features

Including temporal features such as month and year in our dataset can potentially improve the performance of machine learning models, as these features can help capture any seasonal or yearly trends in the data. This can be especially useful for stock price prediction if there are patterns that repeat on a monthly or yearly basis. For instance, some stocks might have recurring patterns related to the time of year due to factors like seasonality in the company's business, fiscal reporting cycles, or broader market trends. By providing the month and year as additional input features, our model may be better able to learn these patterns and make more accurate predictions.

We shall include as additional categorical features the day, month and year of the observed stock price.

In [43]:
```
# Add 'Day', 'Month' and 'Year' columns to X_train_scaled dataset
X_train_scaled['Day'] = X_train_scaled.index.day
X_train_scaled['Month'] = X_train_scaled.index.month
X_train_scaled['Year'] = X_train_scaled.index.year
```

In [44]:
```
# Add 'Day', 'Month' and 'Year' columns to X_test_scaled dataset
X_test_scaled['Day'] = X_test_scaled.index.day
X_test_scaled['Month'] = X_test_scaled.index.month
X_test_scaled['Year'] = X_test_scaled.index.year
```

One of the models we will be using, the Random Forest, is insensitive to rescaling. In this case, we find more intuitive to work with the original dataframe without rescaling. So we will also add the categorical features to the oriignal X_train and X_test features.

In [45]:
```
# Add 'Day', 'Month' and 'Year' columns to X_train dataset
X_train['Day'] = X_train.index.day
X_train['Month'] = X_train.index.month
X_train['Year'] = X_train.index.year
```

In [46]:
```
# Add 'Day', 'Month' and 'Year' columns to X_test dataset
X_test['Day'] = X_test.index.day
X_test['Month'] = X_test.index.month
X_test['Year'] = X_test.index.year
```

In [47]:
```
X_train
```

Out[47]:

Date	Open	High	Low	Adj Close	Volume	Reported EPS	Price Change	R
1996-05-06	19.937500	20.218750	19.750000	10.236874	7170400	0.14	0.281250	54.1666
1996-05-07	20.218750	20.406250	20.156250	10.300159	6702800	0.14	0.125000	56.5789
1996-05-08	20.343750	20.687500	20.062500	10.474213	8292800	0.14	0.343750	65.8536
1996-05-09	20.687500	20.937500	20.593750	10.474213	4820400	0.14	0.000000	71.0526
1996-05-10	20.718750	20.968750	20.718750	10.616608	4942800	0.14	0.281250	68.5714
...	...	...	...	...	...	...	...	
2022-06-13	60.750000	62.290001	60.669998	59.126160	23064200	0.64	-0.070000	39.5027
2022-06-14	60.730000	60.889999	58.660000	57.504799	24620000	0.64	-2.110001	20.7570
2022-06-15	60.000000	60.639999	58.970001	57.931984	19684700	0.64	0.439999	24.3589
2022-06-16	58.639999	59.520000	58.250000	57.349461	15053800	0.64	-0.599998	20.7820
2022-06-17	59.700001	60.130001	59.130001	57.698975	34781900	0.64	0.360001	20.6047

6576 rows × 20 columns

Now we need to store this train/test split and its rescaled version in respective csv files for future use.

In [48]:
```
X_train_scaled.to_csv('X_train_KO_scaled.csv')
X_test_scaled.to_csv('X_test_KO_scaled.csv')
```

In [49]:
```
X_train.to_csv('X_train_KO.csv')
X_test.to_csv('X_test_KO.csv')
```

In [50]:
```
y_train_scaled.to_csv('y_train_KO_scaled.csv')
y_test_scaled.to_csv('y_test_KO_scaled.csv')
```

In [51]:
```
y_train.to_csv('y_train_KO.csv')
y_test.to_csv('y_test_KO.csv')
```

Tesla stock

We shall now repeat the steps we followed previously but focusing this time on the TSLA stock. First we load our dataset:

In [52]:
```
# Read the csv file containing the data for the TSLA stock.
# index_col=0 tells pandas to use the first column as the index.
# parse_dates=True tells pandas to interpret the index as a DateTimeIndex.
df_TSLA = pd.read_csv('TSLA.csv', parse_dates=True, index_col=0)
```

Let's compute the different financial indicators for the TSLA ticker:

In [53]:
```
rsi(df_TSLA)
```

Out[53]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2010-08-05	1.436000	1.436667	1.336667	1.363333	1.363333	11943000	-0.0271
2010-08-06	1.340000	1.344000	1.301333	1.306000	1.306000	11128500	-0.0271
2010-08-09	1.326667	1.332000	1.296667	1.306667	1.306667	12190500	-0.0271
2010-08-10	1.310000	1.310000	1.254667	1.268667	1.268667	19219500	-0.0271
2010-08-11	1.246000	1.258667	1.190000	1.193333	1.193333	11964000	-0.0271
...	...	...	...	...	...	...	...
2022-12-23	126.370003	128.619995	121.019997	123.150002	123.150002	166989700	1.0500
2022-12-27	117.500000	119.669998	108.760002	109.099998	109.099998	208643400	1.0500
2022-12-28	110.349998	116.269997	108.239998	112.709999	112.709999	221070500	1.0500
2022-12-29	120.389999	123.570000	117.500000	121.820000	121.820000	221923300	1.0500
2022-12-30	119.949997	124.480003	119.750000	123.180000	123.180000	157777300	1.0500

3124 rows × 9 columns

In [54]:
```
k(df_TSLA)
```

Out[54]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2010-08-05	1.436000	1.436667	1.336667	1.363333	1.363333	11943000	-0.0271
2010-08-06	1.340000	1.344000	1.301333	1.306000	1.306000	11128500	-0.0271
2010-08-09	1.326667	1.332000	1.296667	1.306667	1.306667	12190500	-0.0271
2010-08-10	1.310000	1.310000	1.254667	1.268667	1.268667	19219500	-0.0271
2010-08-11	1.246000	1.258667	1.190000	1.193333	1.193333	11964000	-0.0271
...	...	...	...	...	...	...	...
2022-12-23	126.370003	128.619995	121.019997	123.150002	123.150002	166989700	1.0500
2022-12-27	117.500000	119.669998	108.760002	109.099998	109.099998	208643400	1.0500
2022-12-28	110.349998	116.269997	108.239998	112.709999	112.709999	221070500	1.0500
2022-12-29	120.389999	123.570000	117.500000	121.820000	121.820000	221923300	1.0500
2022-12-30	119.949997	124.480003	119.750000	123.180000	123.180000	157777300	1.0500

3124 rows × 10 columns

In [55]:
```
macd(df_TSLA)
```

Out[55]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2010-08-05	1.436000	1.436667	1.336667	1.363333	1.363333	11943000	-0.0271
2010-08-06	1.340000	1.344000	1.301333	1.306000	1.306000	11128500	-0.0271
2010-08-09	1.326667	1.332000	1.296667	1.306667	1.306667	12190500	-0.0271
2010-08-10	1.310000	1.310000	1.254667	1.268667	1.268667	19219500	-0.0271
2010-08-11	1.246000	1.258667	1.190000	1.193333	1.193333	11964000	-0.0271
...	...	...	...	...	...	...	...
2022-12-23	126.370003	128.619995	121.019997	123.150002	123.150002	166989700	1.0500
2022-12-27	117.500000	119.669998	108.760002	109.099998	109.099998	208643400	1.0500
2022-12-28	110.349998	116.269997	108.239998	112.709999	112.709999	221070500	1.0500
2022-12-29	120.389999	123.570000	117.500000	121.820000	121.820000	221923300	1.0500
2022-12-30	119.949997	124.480003	119.750000	123.180000	123.180000	157777300	1.0500

3124 rows × 15 columns

In [56]:
```
roc(df_TSLA)
```

Out[56]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2010-08-05	1.436000	1.436667	1.336667	1.363333	1.363333	11943000	-0.0271
2010-08-06	1.340000	1.344000	1.301333	1.306000	1.306000	11128500	-0.0271
2010-08-09	1.326667	1.332000	1.296667	1.306667	1.306667	12190500	-0.0271
2010-08-10	1.310000	1.310000	1.254667	1.268667	1.268667	19219500	-0.0271
2010-08-11	1.246000	1.258667	1.190000	1.193333	1.193333	11964000	-0.0271
...	...	...	...	...	...	...	...
2022-12-23	126.370003	128.619995	121.019997	123.150002	123.150002	166989700	1.0500
2022-12-27	117.500000	119.669998	108.760002	109.099998	109.099998	208643400	1.0500
2022-12-28	110.349998	116.269997	108.239998	112.709999	112.709999	221070500	1.0500
2022-12-29	120.389999	123.570000	117.500000	121.820000	121.820000	221923300	1.0500
2022-12-30	119.949997	124.480003	119.750000	123.180000	123.180000	157777300	1.0500

3124 rows × 16 columns

In [57]:
```
obv(df_TSLA)
```

```
/home/sergio/anaconda3/lib/python3.9/site-packages/pandas/core/indexing.py:1732: Set
tingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  self._setitem_single_block(indexer, value, name)
```

Out[57]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2010-08-05	1.436000	1.436667	1.336667	1.363333	1.363333	11943000	-0.0271
2010-08-06	1.340000	1.344000	1.301333	1.306000	1.306000	11128500	-0.0271
2010-08-09	1.326667	1.332000	1.296667	1.306667	1.306667	12190500	-0.0271
2010-08-10	1.310000	1.310000	1.254667	1.268667	1.268667	19219500	-0.0271
2010-08-11	1.246000	1.258667	1.190000	1.193333	1.193333	11964000	-0.0271
...	...	...	...	...	...	...	...
2022-12-23	126.370003	128.619995	121.019997	123.150002	123.150002	166989700	1.0500
2022-12-27	117.500000	119.669998	108.760002	109.099998	109.099998	208643400	1.0500
2022-12-28	110.349998	116.269997	108.239998	112.709999	112.709999	221070500	1.0500
2022-12-29	120.389999	123.570000	117.500000	121.820000	121.820000	221923300	1.0500
2022-12-30	119.949997	124.480003	119.750000	123.180000	123.180000	157777300	1.0500

3124 rows × 17 columns

In [58]:
```
pe(df_TSLA)
```

Out[58]:

Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2010-08-05	1.436000	1.436667	1.336667	1.363333	1.363333	11943000	-0.0271
2010-08-06	1.340000	1.344000	1.301333	1.306000	1.306000	11128500	-0.0271
2010-08-09	1.326667	1.332000	1.296667	1.306667	1.306667	12190500	-0.0271
2010-08-10	1.310000	1.310000	1.254667	1.268667	1.268667	19219500	-0.0271
2010-08-11	1.246000	1.258667	1.190000	1.193333	1.193333	11964000	-0.0271
...	...	...	...	...	...	...	...
2022-12-23	126.370003	128.619995	121.019997	123.150002	123.150002	166989700	1.0500
2022-12-27	117.500000	119.669998	108.760002	109.099998	109.099998	208643400	1.0500
2022-12-28	110.349998	116.269997	108.239998	112.709999	112.709999	221070500	1.0500
2022-12-29	120.389999	123.570000	117.500000	121.820000	121.820000	221923300	1.0500
2022-12-30	119.949997	124.480003	119.750000	123.180000	123.180000	157777300	1.0500

3124 rows × 18 columns

We shall not drop the rows with NaN values.

In [59]:
```
# Drop rows with NaN values
df_TSLA = df_TSLA.dropna()

df_TSLA
```

Out[59]:

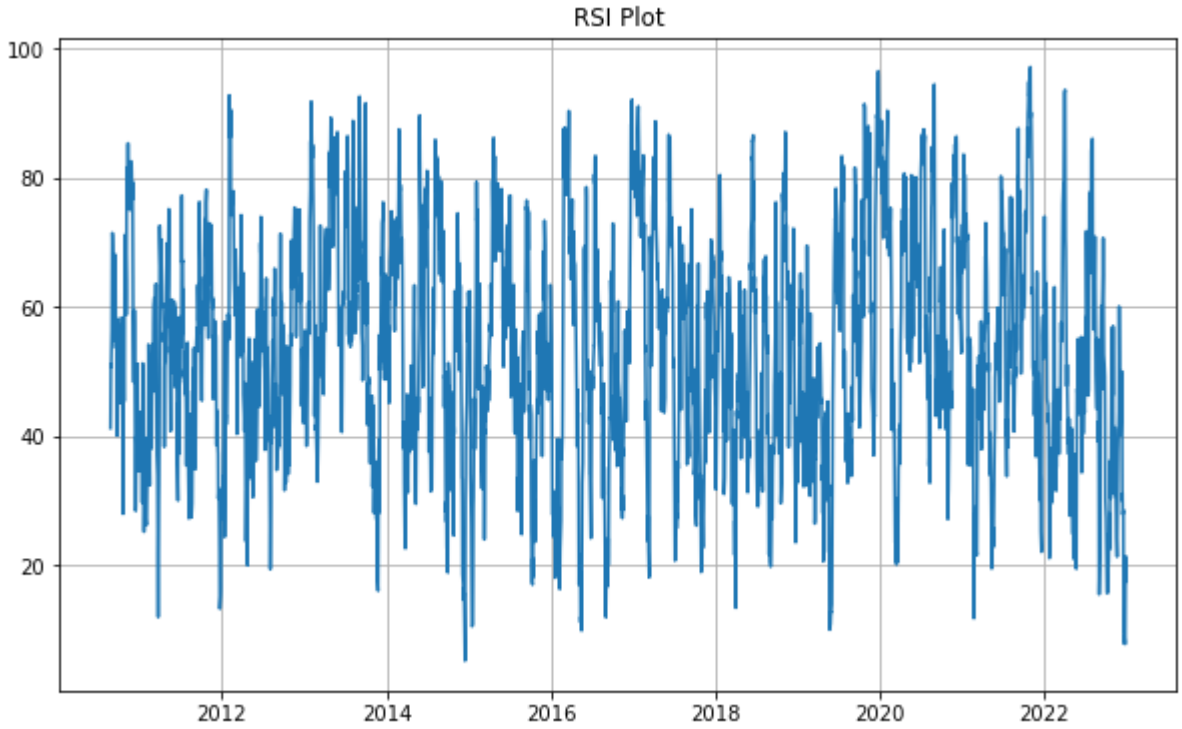
Date	Open	High	Low	Close	Adj Close	Volume	Reported EPS
2010-08-24	1.283333	1.314000	1.263333	1.280000	1.280000	10096500	-0.0271
2010-08-25	1.277333	1.332000	1.237333	1.326667	1.326667	7549500	-0.0271
2010-08-26	1.326000	1.351333	1.306667	1.316667	1.316667	6507000	-0.0271
2010-08-27	1.316667	1.324667	1.300000	1.313333	1.313333	5694000	-0.0271
2010-08-30	1.313333	1.346000	1.307333	1.324667	1.324667	10992000	-0.0271
...	...	...	...	...	...	...	...
2022-12-23	126.370003	128.619995	121.019997	123.150002	123.150002	166989700	1.0500
2022-12-27	117.500000	119.669998	108.760002	109.099998	109.099998	208643400	1.0500
2022-12-28	110.349998	116.269997	108.239998	112.709999	112.709999	221070500	1.0500
2022-12-29	120.389999	123.570000	117.500000	121.820000	121.820000	221923300	1.0500
2022-12-30	119.949997	124.480003	119.750000	123.180000	123.180000	157777300	1.0500

3111 rows × 18 columns

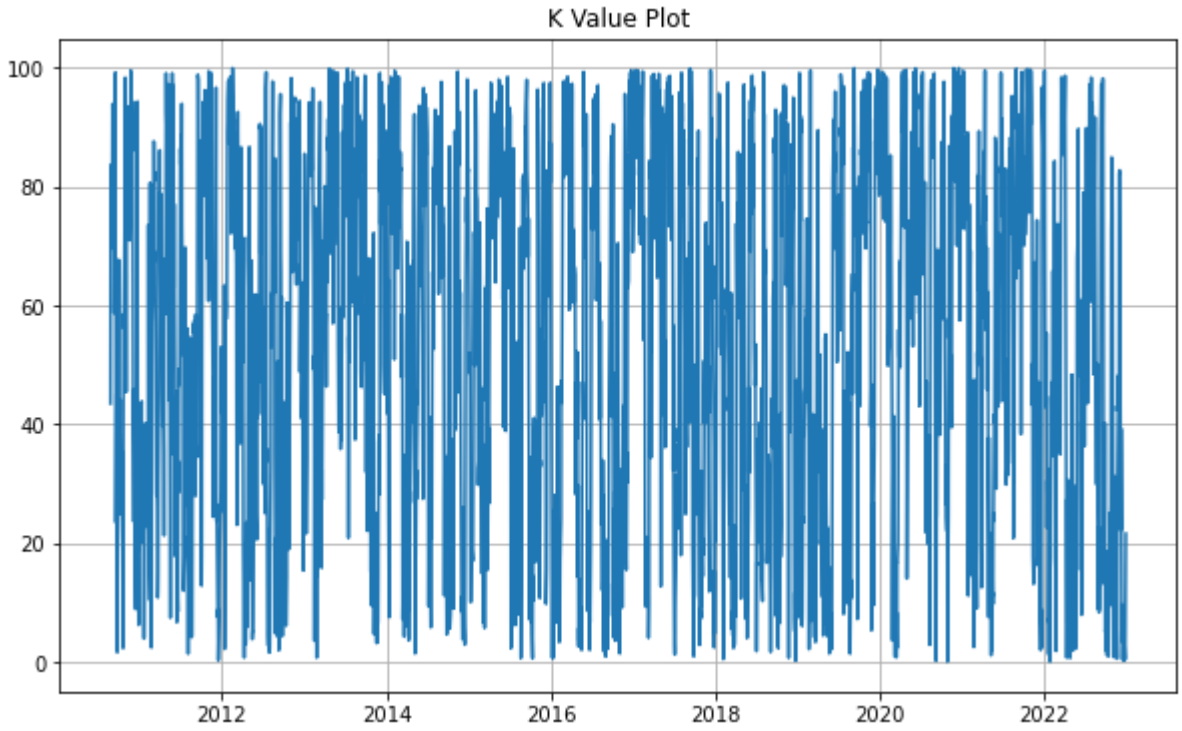
In [60]:
```
df_TSLA.to_csv('df_TSLA_.csv')
```

Plots

In [61]:
```
plot_rsi(df_TSLA)
```

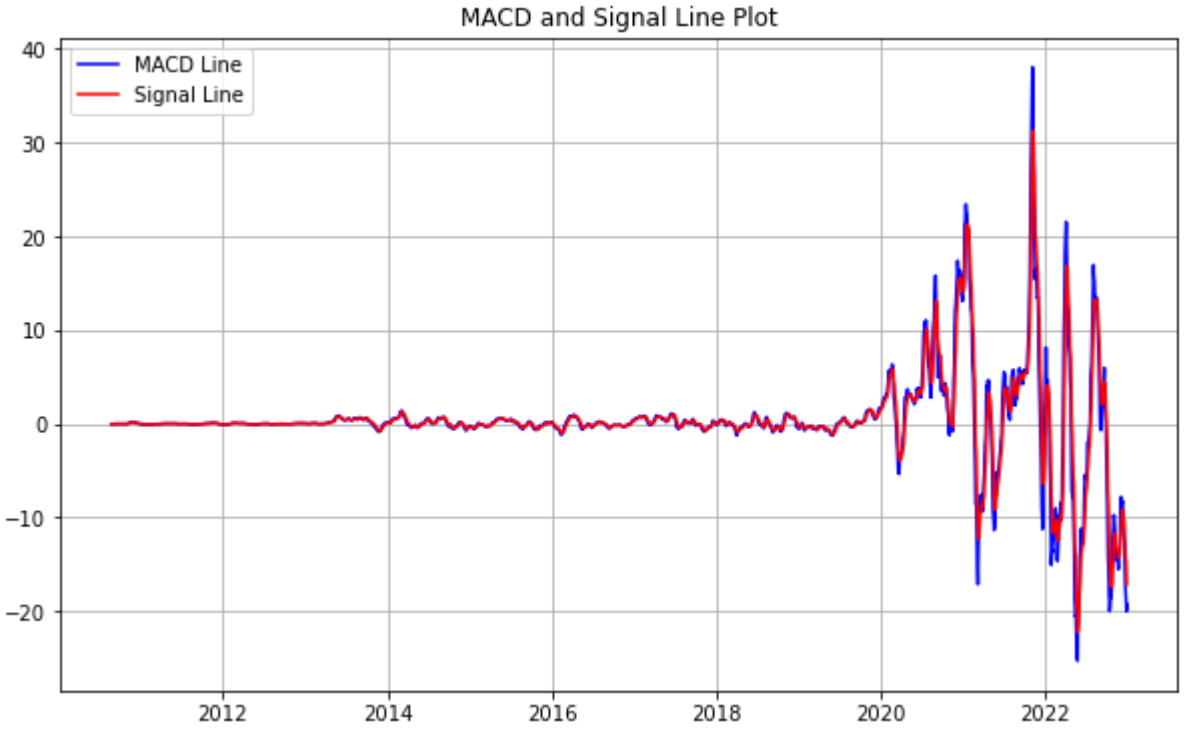


In [62]:
```
plot_K(df_TSLA)
```



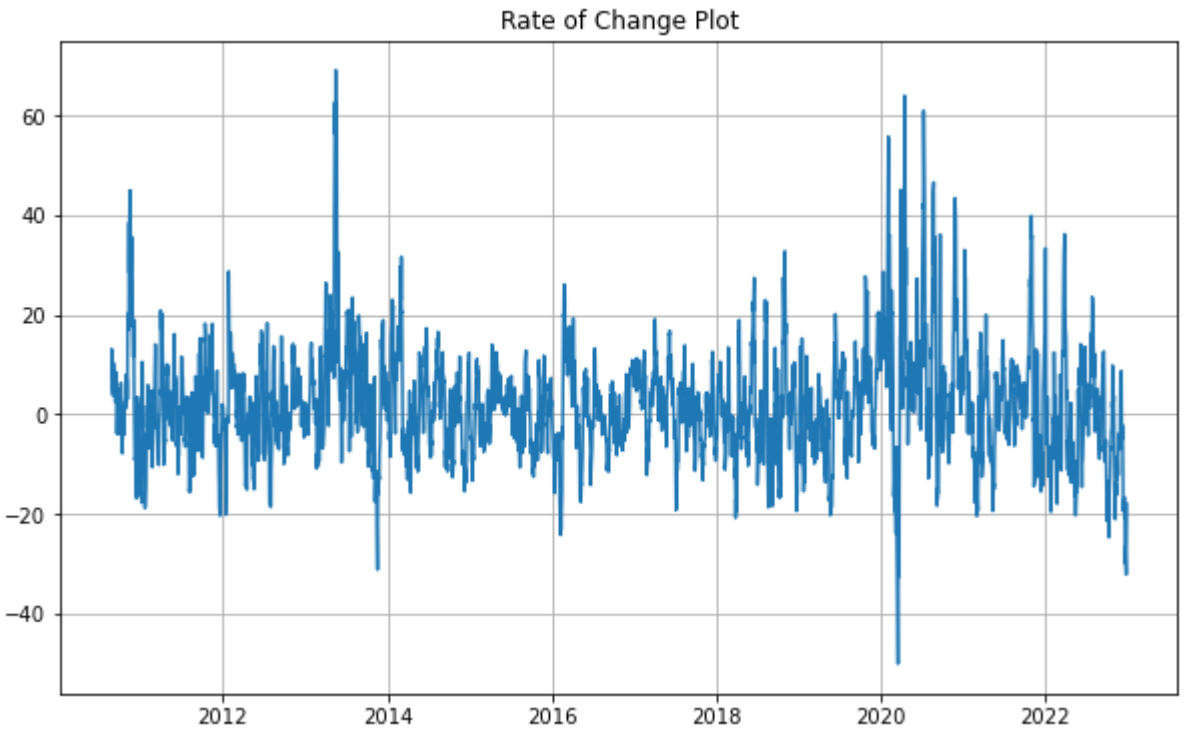
In this case the RSI and K value plots seems to cross more frequently their respective 'overbought' threshold. This is consistent with the history of TESLA of being frequently overbought and overpriced.

In [63]:
```
plot_MACD_and_Signal(df_TSLA)
```



The MACD plot indicates an almost alternating history of bullish and bearish signals since the company starts experiencing an exponential growth in 2020, which is consistent with the stock's high volatility.

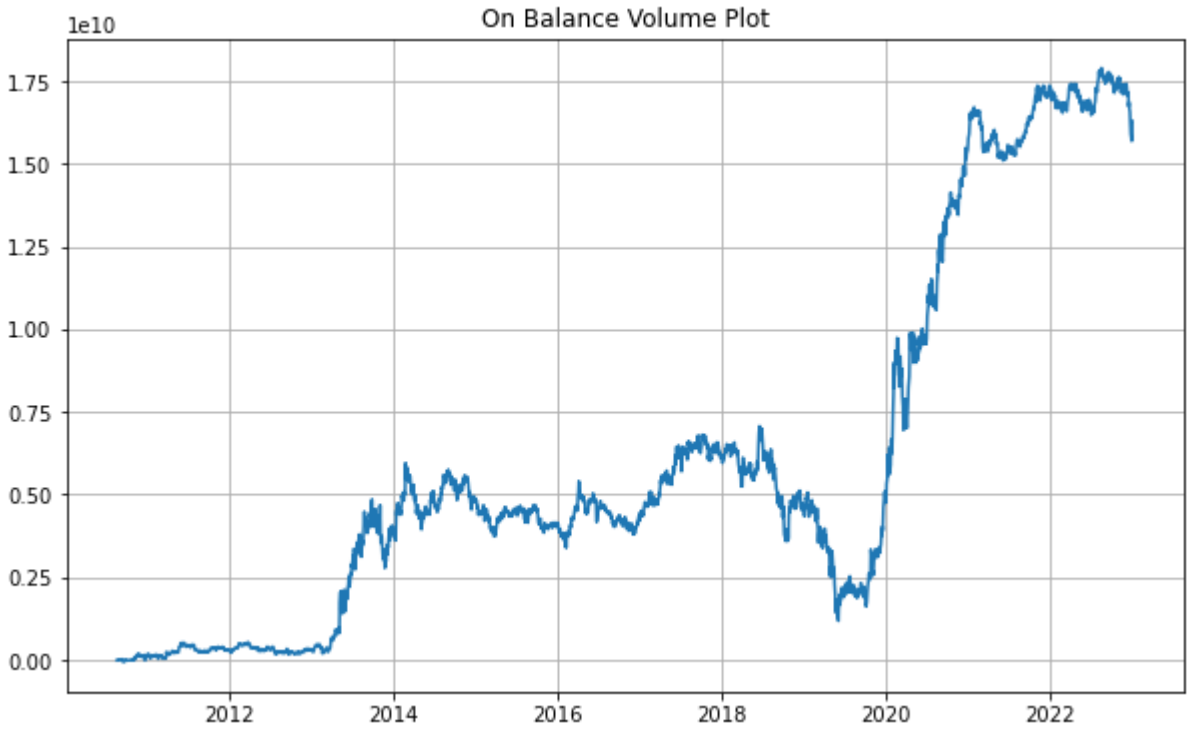
In [64]:
```
plot_ROC(df_TSLA)
```



The ROC plot generally oscillates between +20 and -20, which are already high values indicating respectively bullish and bearish signs. There are quite a few points with more extreme values, such as the one above +60 in 2013, which we can check in the Close Price

plot that was followed by a trend of slight growth and can be related to the company reporting higher-than-expected demand for its Model S electric sedan. Or the very negative value below -40 in 2020 related to the onset of the COVID pandemic. We also observe that from 2020 the peaks are more frequent, which can be related to the cascade of news about the company that resulted in the stock's high volatility.

In [65]:
```
plot_OBV(df_TSLA)
```



We observe a period between 2018 and late 2019 where the OBV plot fell down while the Close Price remained ŕather constant. This suggests that volume on down days is outweighing volume on up days, and could be a sign that selling pressure is increasing, which could precede a drop in the price. In this case, this scenario could have resulted from the frequent negative EPS reported by the company over that period, which could have influenced the investors' behavior. Contrarily to the expectations, the company started a period of exponential growth just in 2020.

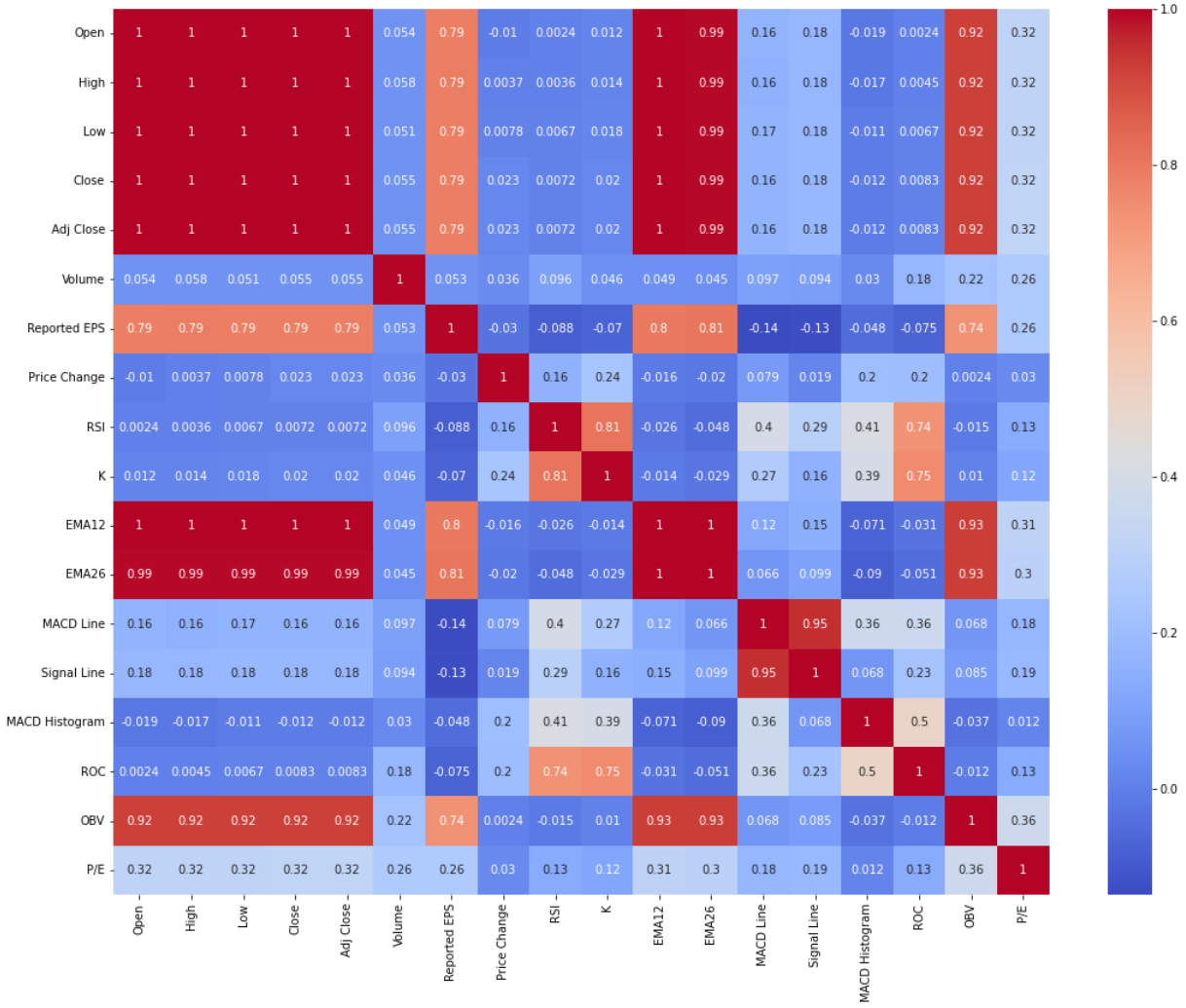
In [66]:
```
plot_PE(df_TSLA)
```

Correlations

Let's have a look at the correlation heatmap of the columns in `df_TSLA`:

In [67]:
```
# Plot a correlation heatmap of all columns in 'df_TSLA'
correlation_matrix = df_TSLA.corr()
plt.figure(figsize=(18, 14))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
```

Out[67]:
```
<AxesSubplot:>
```



We observe a similar pattern to the one described for the KO stock: a strong correlation between prices, EMA12 and EMA26 and OBV.

Train-test split

We can use the function `train_test_split` defined above to perform a train-test split of the TSLA dataset.

In [68]:
```
X_train, X_test, y_train, y_test = train_test_split(df_TSLA)
```

In [ ]:
```
```

Rescaling

We follow the same steps for the rescaling of numerical features as above:

In [69]:
```
# Initialize the scaler
scaler_features = RobustScaler()
scaler_target = RobustScaler()
```

```
# Fit the scaler on the training data
scaler_features.fit(X_train)
scaler_target.fit(y_train)

# Save the scaler object to a file
joblib.dump(scaler_target, 'robust_scaler2.pkl')

# Transform the training and testing data
X_train_scaled_np = scaler_features.transform(X_train)
X_test_scaled_np = scaler_features.transform(X_test)

# Transform the training and testing data
y_train_scaled_np = scaler_target.transform(y_train)
y_test_scaled_np = scaler_target.transform(y_test)
```

We can transform the numpy objects back to DataFrame:

In [70]:
```
X_train_scaled = pd.DataFrame(X_train_scaled_np, columns=X_train.columns, index=X_t
```

In [71]:
```
X_test_scaled = pd.DataFrame(X_test_scaled_np, columns=X_test.columns, index=X_test
```

In [72]:
```
y_train_scaled = pd.DataFrame(y_train_scaled_np, columns=y_train.columns, index=X_t
```

In [73]:
```
y_test_scaled = pd.DataFrame(y_test_scaled_np, columns=y_test.columns, index=X_test
```

Including temporal categorial features

We include the temporal categorical features `Day`, `Month` and `Year` to our dataframes:

In [74]:
```
# Add 'Day', 'Month' and 'Year' columns to X_train_scaled dataset
X_train_scaled['Day'] = X_train_scaled.index.day
X_train_scaled['Month'] = X_train_scaled.index.month
X_train_scaled['Year'] = X_train_scaled.index.year
```

In [75]:
```
# Add 'Day', 'Month' and 'Year' columns to X_test_scaled dataset
X_test_scaled['Day'] = X_test_scaled.index.day
X_test_scaled['Month'] = X_test_scaled.index.month
X_test_scaled['Year'] = X_test_scaled.index.year
```

Also for the unscaled ones:

In [76]:
```
# Add 'Day', 'Month' and 'Year' columns to X_train dataset
X_train['Day'] = X_train.index.day
X_train['Month'] = X_train.index.month
X_train['Year'] = X_train.index.year
```

In [77]:
```
# Add 'Day', 'Month' and 'Year' columns to X_test dataset
X_test['Day'] = X_test.index.day
X_test['Month'] = X_test.index.month
X_test['Year'] = X_test.index.year
```

We finish by saving our train-test split into csv files:

In [78]:
```
X_train_scaled.to_csv('X_train_TSLA_scaled.csv')
X_test_scaled.to_csv('X_test_TSLA_scaled.csv')
```

In [79]:
```
X_train.to_csv('X_train_TSLA.csv')
X_test.to_csv('X_test_TSLA.csv')
```

In [80]:
```
y_train_scaled.to_csv('y_train_TSLA_scaled.csv')
y_test_scaled.to_csv('y_test_TSLA_scaled.csv')
```

In [81]:
```
y_train.to_csv('y_train_TSLA.csv')
y_test.to_csv('y_test_TSLA.csv')
```

Section 4: Modeling Random Forests and LSTM neural networks

The purpose of this section is to train two machine learning models -Random Forests and LSTM neural network- to predict the `Close` price of our test dataset, and assess their performance. In both cases we will investigate how the size of the test set affects the model's performance.

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import mean_squared_error
```

Random Forest

Random Forests are an ensemble learning method (namely, a method that combines the predictions of multiple individual models --often called "base models" or "learners"-- to improve the overall performance) used for both classification and regression. They operate by constructing multiple decision trees during training and outputting the class that is the mode (for classification) or the mean prediction (for regression) of the individual trees.

Random Forests work on the following three principles:

1. Bagging: Bagging (Bootstrap Aggregating) is an ensemble technique lying at the core of Random Forests. Each tree in the forest is built on a different subset of the data (drawn with replacement), which means that the individual trees are trained on diverse samples. This diversity reduces the chances of overfitting.

2. Feature Randomness: In addition to bagging, Random Forests also select a random subset of features at each split. This introduces more diversity into the tree-building process and further reduces the risk of overfitting.

3. Voting/Averaging: The final prediction of the Random Forest is an average (for regression) or a majority vote (for classification) of the predictions of all the trees. This aggregation helps in smoothing out the noise and anomalies of individual trees.

As generic features of Random Forests, we can mention:

1. Due to the averaging nature of Random Forests, they are less likely to overfit compared to individual decision trees. Nevertheless, they can overfit noisy data if not properly tuned.

2. They naturally consider nonlinearities and interactions between features.

3. They can handle large dataset with higher dimensionality, though they may become computationally intensive.
4. Importance Scoring: Random Forests can rank the importance of input variables.
5. They are insensitive to rescaling features. The splitting criterion in decision trees is based on purity scores, such as the Gini impurity or information gain. These criteria are not affected by the scale of the data.
6. Flexibility: They can handle missing values and are not sensitive to outliers.
7. Parallelizable: Tree constructions can be done in parallel, making them suited for large datasets.
8. Less Interpretable: While an individual decision tree can be visualized and interpreted, a Random Forest is more of a "black box," making it harder to interpret how decisions are made.

We can argue that some of the features above make Random Forests an attractive model for stock price forecasting, as they can handle complex non-linearities of the stock market and capture interactions betwen features, and reduce overfitting (a common issue in stock price prediction) if properly tuned. Moreover, the ensemble nature of random forests can help in reducing variance and potentially provide a more stable prediction. On the other hand, the typical non-stationary of financial time-series (meaning their statistical properties change over time) presents a challenge for random forest models, as these do not take into account the temporal structure of time series data and assume that the underlying process generating the data does not change over time. Nevertheless, it is possible to engineer lagged features (values from previous time steps) to provide some context of the temporal structure. Depending on the behaviour of the stock at hand, random forest models might be more suited for understanding relationships and factors influencing stock prices rather than for making trading decisions.

First step is to import the Random Forest Regressor model:

In [2]:
```
# Import the RandomForestRegressor model
from sklearn.ensemble import RandomForestRegressor
```

1. Coca-Cola stock

Loading the data

We need to load the training and test features and target datasets we saved in the previous notebook. We are using the unscaled datasets as Random Forests are insensitive to scaling.

In [3]:
```
# Load the X_train, y_train, X_test and y_test datasets
```

```
X_train = pd.read_csv('X_train_KO.csv', parse_dates=True, index_col=0)
y_train = pd.read_csv('y_train_KO.csv', parse_dates=True, index_col=0)

X_test = pd.read_csv('X_test_KO.csv', parse_dates=True, index_col=0)
y_test = pd.read_csv('y_test_KO.csv', parse_dates=True, index_col=0)
```

In [4]:
```
X_train
```

Out[4]:

Date	Open	High	Low	Adj Close	Volume	Reported EPS	Price Change	R
1996-05-06	19.937500	20.218750	19.750000	10.236874	7170400	0.14	0.281250	54.16666
1996-05-07	20.218750	20.406250	20.156250	10.300159	6702800	0.14	0.125000	56.57894
1996-05-08	20.343750	20.687500	20.062500	10.474213	8292800	0.14	0.343750	65.85365
1996-05-09	20.687500	20.937500	20.593750	10.474213	4820400	0.14	0.000000	71.05263
1996-05-10	20.718750	20.968750	20.718750	10.616608	4942800	0.14	0.281250	68.57142
...	...	...	...	...	...	...	...	
2022-06-13	60.750000	62.290001	60.669998	59.126160	23064200	0.64	-0.070000	39.50276
2022-06-14	60.730000	60.889999	58.660000	57.504799	24620000	0.64	-2.110001	20.75706
2022-06-15	60.000000	60.639999	58.970001	57.931984	19684700	0.64	0.439999	24.35897
2022-06-16	58.639999	59.520000	58.250000	57.349461	15053800	0.64	-0.599998	20.78209
2022-06-17	59.700001	60.130001	59.130001	57.698975	34781900	0.64	0.360001	20.60470

6576 rows × 20 columns

We save the features names for future use:

In [5]:
```
features = X_train.columns
```

Training the model

We now proceed to train our Random Forest model and make predictions on the testing dataset. We choose to use a Random Forest Regressor model with 5 trees (n_estimators=5)

and set a random seed for reproducibility of our results (random_state=42).

In [6]:
```
# Initialize the random forest regressor
rf_model = RandomForestRegressor(n_estimators=5, random_state=42, n_jobs=-1)

# Fit the model on the training data
rf_model.fit(X_train, y_train.values.ravel())

# Make predictions on the testing data
y_pred = rf_model.predict(X_test)
```

The performance of the model will be assessed by using the mean squared error (MSE):

In [7]:
```
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

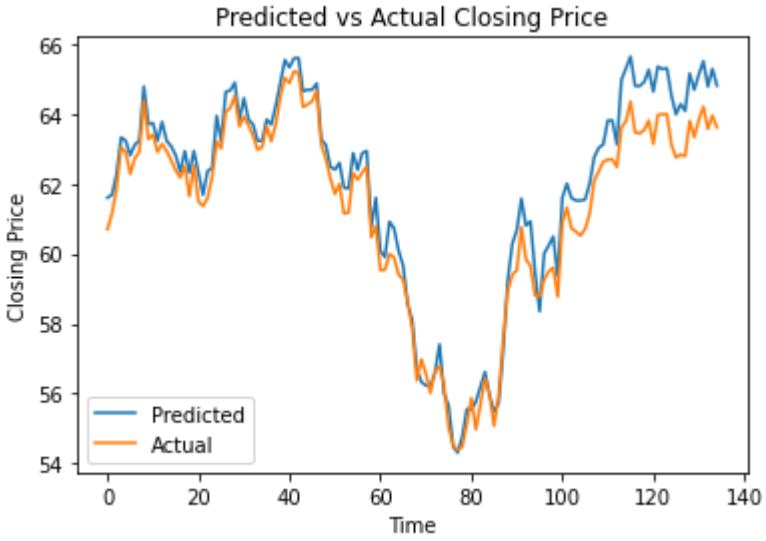
```
Mean Squared Error: 0.566241967822413
```

This means that that on average, our predictions are off by roughly 0.75 dollars from the true value, which we may consider a rather small error, especially for the reduced number of trees used.

In order to compare the predicted values of the target variable and its actual values we plot the corresponding series:

In [8]:
```
def plot_predictions(y_pred, y_test):
    """
    Plot the predicted values against the true values.

    Parameters:
    - y_pred: list or array-like of predicted values.
    - y_test: list or array-like of true values.
    """

    # Create a range of indices for the x-axis
    x = range(len(y_pred))

    # Plot the predicted values (y_pred) and the true values (y_test)
    plt.plot(x, y_pred, label='Predicted')
    plt.plot(x, y_test, label='Actual')

    # Set labels and title
    plt.ylabel('Closing Price')
    plt.xlabel('Time')
    plt.title('Predicted vs Actual Closing Price')

    # Add legend
    plt.legend()

    # Show the plot
    plt.show()
```

In [9]:
```
plot_predictions(y_pred, y_test)
```



We observe a quite good match between the predicted and actual Closing price distributions. Although the curves tend to slightly spread out from each other past around 90 timesteps, their shapes remain practically identical.

The size of the previous test set was quite small, only 2% of the total dataset. We are interested in investigating the performance of random forests in learning larger test sets. To that end, we are going to consider different test set sizes: 10%, 15% and 20%.

We first need to load the whole dataset:

In [10]:
```
df_KO = pd.read_csv('df_KO_.csv', parse_dates=True, index_col=0)
```

In [ ]:
```
```

In [11]:
```
def split_data(df, test_size=0.02, target='Close'):
    '''
    Perform train-test split with respect to time series structure.

    Parameters:
    df: DataFrame, the input pandas DataFrame.
    test_size: float, the proportion of the dataset to include in the test split.
    target: str, the target variable to predict. Default is 'Close'.

    Returns:
    X_train, X_test, y_train, y_test : series, split datasets.
    '''

    # Ensure df is a pandas DataFrame
    if not isinstance(df, pd.DataFrame):
        raise TypeError("Data should be a pandas DataFrame.")

    # Calculate the index at which to split the DataFrame into training and testing
    split_index = int(len(df) * (1 - test_size))

    # Split the DataFrame into training and testing data
    train_data = df.iloc[:split_index]
```

```
    test_data = df.iloc[split_index:]

    # Separate the features and the target variable in the training and testing set
    X_train = train_data.drop([target], axis=1)
    y_train = train_data[target].to_frame()
    X_test = test_data.drop([target], axis=1)
    y_test = test_data[target].to_frame()

    X_train['Day'] = X_train.index.day
    X_train['Month'] = X_train.index.month
    X_train['Year'] = X_train.index.year

    X_test['Day'] = X_test.index.day
    X_test['Month'] = X_test.index.month
    X_test['Year'] = X_test.index.year


    return X_train, X_test, y_train, y_test
```

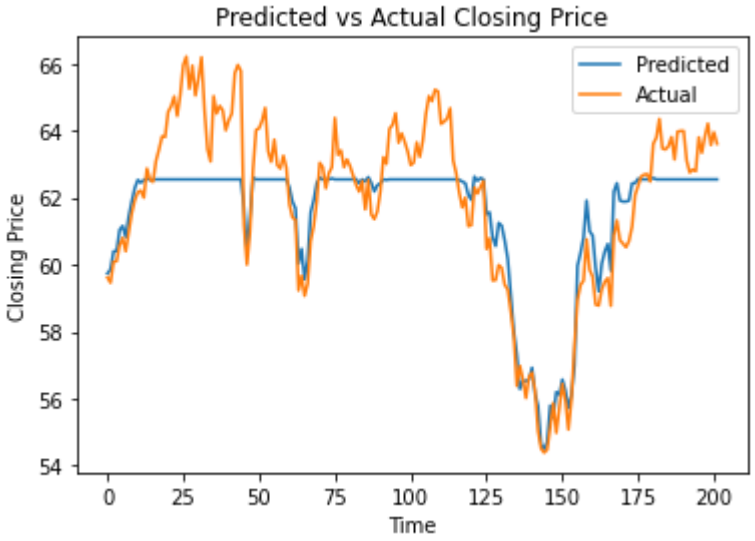
In [ ]:

In [12]:
```
# Perform a data split with test size 3%
X_train1, X_test1, y_train1, y_test1 = split_data(df_KO, test_size=0.03)

X_train1 = X_train1[['Open','High', 'Low','Adj Close']]
X_test1 = X_test1[['Open','High', 'Low','Adj Close']]

rf_model1 = RandomForestRegressor(n_estimators=5,random_state=42, n_jobs=-1)

# Fit the model on the training data
rf_model1.fit(X_train1, y_train1.values.ravel())

# Make predictions on the testing data
y_pred = rf_model1.predict(X_test1)

# Compute and print MSE
mse = mean_squared_error(y_test1, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 1.634829830013316
```

In [ ]:

In [13]:
```
plot_predictions(y_pred, y_test1)
```



We already observe a pattern that will be present for larger test set sizes: the presence of segments where the prediction flattens to a noisy horizontal line, alternating with segments where the prediction adjusts relatively well to the actual data. Changing the values of the hyperparameters does not solve this issue.

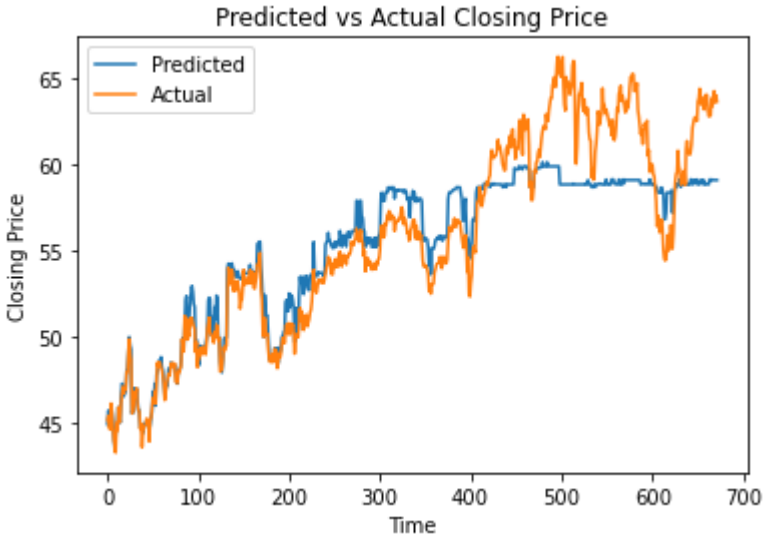
In [ ]:

In [14]:
```
X_train2, X_test2, y_train2, y_test2 = split_data(df_KO, test_size=0.1)
```

In [15]:
```
# Fit the model on the training data
rf_model1.fit(X_train2, y_train2.values.ravel())

# Make predictions on the testing data
y_pred = rf_model1.predict(X_test2)

# Compute and print MSE
mse = mean_squared_error(y_test2, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 5.677182937718932
```

In [16]:
```
plot_predictions(y_pred, y_test2)
```



In [ ]:

In [ ]:

In [17]:
```
X_train3, X_test3, y_train3, y_test3 = split_data(df_KO, test_size=0.15)
```

In [18]:
```
# Fit the model on the training data
rf_model1.fit(X_train3, y_train3.values.ravel())

# Make predictions on the testing data
y_pred = rf_model1.predict(X_test3)

# Compute and print MSE
mse = mean_squared_error(y_test3, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 45.256783625915396
```

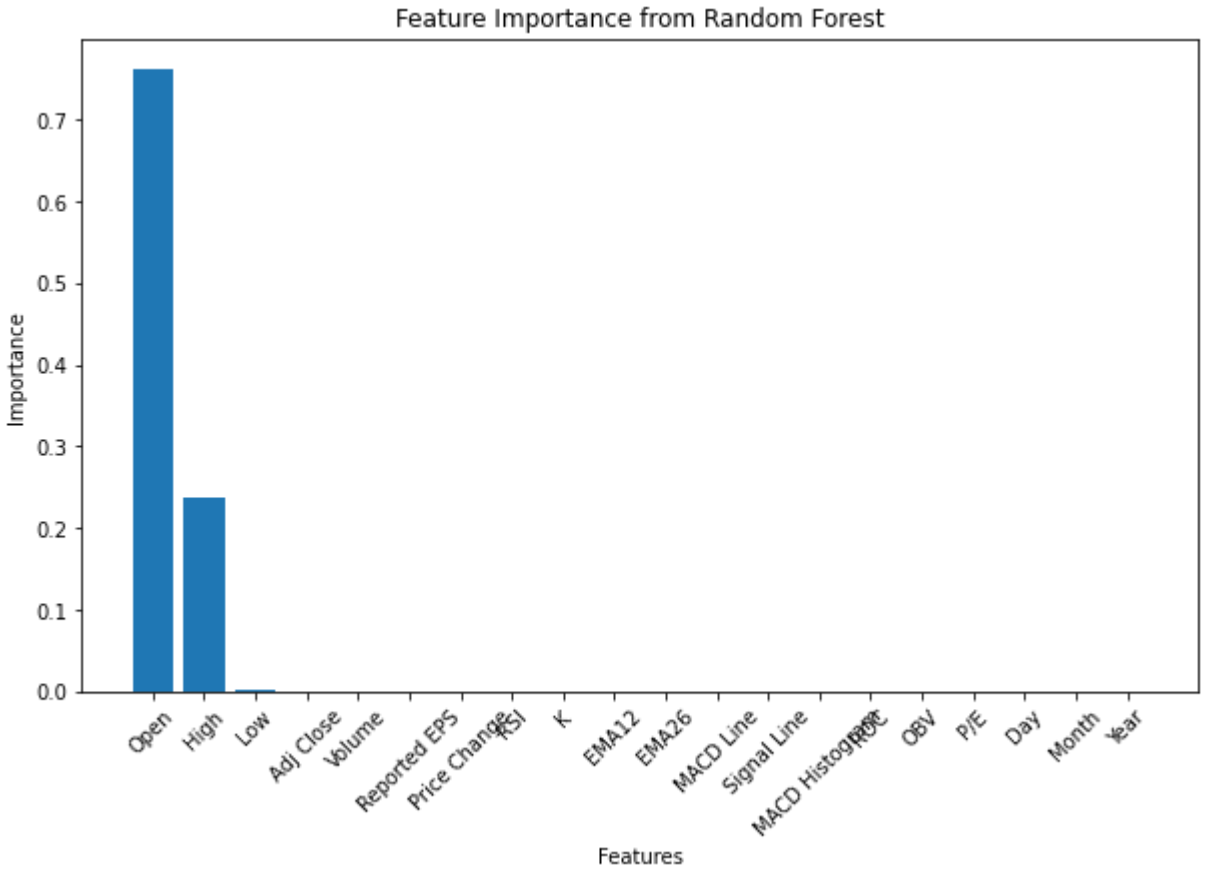
In [19]:
```
plot_predictions(y_pred, y_test3)
```

In general, the larger the size of the test set, the larger is the MSE due to larger regions having a constant prediction. From now on we restricti ourselves to the case of test set size 2%: this might seem small a priori, but it covers around 140 days of prediction, which is satisfactory in the context of short-term market forecasting.

Variable importance

We shall now perform a feature importance analysis, using the `feature_importances_` attribute, which provides a score for each feature of the data, with higher scores indicating more importance.

In [20]:
```
importances = rf_model.feature_importances_

sorted_idx = importances.argsort()[::-1]

plt.figure(figsize=(10,6))
plt.bar(features, importances[sorted_idx])
plt.xlabel('Features')
plt.ylabel('Importance')
plt.title('Feature Importance from Random Forest')
plt.xticks(rotation=45)
plt.show()
```



It turns out that the feature importance in this model is essentially split between the `Open` and `High` features, with respective importances around 0.75 and 0.25. We know that

additiona features can add extra variance and worsen the performance of a random forest, as well as producing a dilution of important features at each split in a tree, which can make the model less accurate. It is worth then to check if there is a subset of features that minimizes the MSE.

In this case, we have checked that for `test_size=0.02` the minimal subset of features is composed by `Open`, `High` and `Low`.

In [21]:
```
X_train = X_train[['Open','High', 'Low']]
X_test = X_test[['Open','High', 'Low']]

# Fit the model on the training data
rf_model.fit(X_train, y_train.values.ravel())

# Make predictions on the testing data
y_pred = rf_model.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 0.1661110703294957
```

Note that the MSE is reduced from 0.566 to 0.166 (around 70% reduction) only by selecting the most important features.

On the other hand, the presence of plateaus for higher values of the test set size is not solved by restricting the training set to the most important features.

From now on we focus on the model trained with only `Open`, `High` and `Low` features.

Cross-validaton

It is time now to perform model validation using cross-validation.

When dealing with time series data, you can't use traditional k-fold cross-validation because it shuffles the data, which can result in data leakage: information from the future might be used to predict the past. Instead, you use time series cross-validation where data is split in a way that respects the order of the data. This is achieved using the `tscv.split(X)` method. Essentially, it is used to obtain indices for train/test splits suitable for time series data, ensuring that the chronological order of the data is maintained and preventing any information leakage.

In [22]:
```
from sklearn.model_selection import TimeSeriesSplit

# Number of splits. For example, if n_splits=5, it will produce 5 train/test sets.
tscv = TimeSeriesSplit(n_splits=5)

# Initialize the Random Forest Regressor
model = RandomForestRegressor()
```

```
# Define a list for the MSE scores
mse_scores = []

# Perform the time series cross-validation
for train_index, test_index in tscv.split(X_train):
    X_train_cv, X_test_cv = X_train.iloc[train_index], X_train.iloc[test_index]
    y_train_cv, y_test_cv = y_train.iloc[train_index], y_train.iloc[test_index]


    model.fit(X_train, y_train.values.ravel())
    y_pred = model.predict(X_test)

    mse = mean_squared_error(y_test, y_pred)
    mse_scores.append(mse)

# Calculate the mean and standard deviation of MSE across folds
mean_mse = np.mean(mse_scores)
std_mse = np.std(mse_scores)

print(f"Mean MSE: {mean_mse:.4f} (+/- {std_mse:.4f})")
```

```
Mean MSE: 0.1329 (+/- 0.0029)
```

In [ ]:

The mean MSE obtained in the cross-validation process is very similar to the MSE obtained in our test set, with a very small standard deviation. This gives us confidence that the model is showing consistent performance across different subsets of the training data (as observed during cross-validation) and on unseen data (test set). This is a good indication that the model is not overfitting to specific portions of the training data.

Hyperparameter tuning

Although our previous model performs quite well on the test set with a small MSE, we can still further adjust the hyperparameters to check if a better performance is possible. We should then weight how much the MSE improves versus how complicated the model gets.

For this purpose, we will use `RandomizedSearchCV`. Unlike `GridSearchCV`, which evaluates the model for every combination of hyperparameters provided, `RandomizedSearchCV` randomly selects a subset of hyperparameter combinations. The number of combinations to try is specified by the `n_iter` parameter. Here we consider 100 random combinations of hyperparameter values and select the one among these that has a minimal MSE. We then check whether this MSE is less than the one of the model with the hyperparameter values previously used.

In [ ]:

In [23]:
```
from sklearn.model_selection import RandomizedSearchCV

# Define the parameter grid
param_dist = {
```

```
    'n_estimators': np.arange(3, 100),
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth': [None] + list(np.arange(1, 30)),
    'min_samples_split': np.arange(2, 50),
    'min_samples_leaf': np.arange(1, 50),
    'bootstrap': [True, False]
}

# Initialize RandomForestRegressor instance
rf = RandomForestRegressor()

# Perform RandomizedSearchCV
random_search = RandomizedSearchCV(
    rf, param_distributions=param_dist, n_iter=100,
    scoring='neg_mean_squared_error', cv=tscv,
    verbose=1, n_jobs=-1, random_state=42
)

# Fit the model
random_search.fit(X_train, y_train.values.ravel())

# Print the best parameters
print(random_search.best_params_)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
{'n_estimators': 63, 'min_samples_split': 8, 'min_samples_leaf': 3, 'max_features':
'sqrt', 'max_depth': 9, 'bootstrap': True}
```

In [24]:
```
# Initialize the random forest regressor
rf_model = RandomForestRegressor(n_estimators=63, min_samples_split=8, min_samples_

# Fit the model on the training data
rf_model.fit(X_train, y_train.values.ravel())

# Make predictions on the testing data
y_pred = rf_model.predict(X_test)
```

In [25]:
```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 0.15678970258562255
```

We have found a different choice of hyperparameters that slightly reduces the MSE from 0.166 to 0.156. Nevertheless, the new model is way more complicated: the number of trees has grown from 5 to 63, and other hyperparameters have also been increased. It does not seem reasonable to complicate our model so much for such a little improvement in MSE.

We should keep in mind that RandomSearchCV does not sweep over all the parameter grid we have defined, it only consider random points of the grid. So it is still possible that some choice of parameters reduces the MSE still rendering the model simple. In any case, an MSE of 0.166 is quite satisfactory, especially given the results of the cross-validation study.

2. Tesla stock

Loading the data

In [26]:
```
# Load the X_train, y_train, X_test and y_test datasets

X_train = pd.read_csv('X_train_TSLA.csv', parse_dates=True, index_col=0)
y_train = pd.read_csv('y_train_TSLA.csv', parse_dates=True, index_col=0)

X_test = pd.read_csv('X_test_TSLA.csv', parse_dates=True, index_col=0)
y_test = pd.read_csv('y_test_TSLA.csv', parse_dates=True, index_col=0)
```

In [27]:
```
X_train
```

Out[27]:

Date	Open	High	Low	Adj Close	Volume	Reported EPS	Price Change	
2010-08-24	1.283333	1.314000	1.263333	1.280000	10096500	-0.0271	-0.062000	41
2010-08-25	1.277333	1.332000	1.237333	1.326667	7549500	-0.0271	0.046667	46
2010-08-26	1.326000	1.351333	1.306667	1.316667	6507000	-0.0271	-0.010000	51
2010-08-27	1.316667	1.324667	1.300000	1.313333	5694000	-0.0271	-0.003334	50
2010-08-30	1.313333	1.346000	1.307333	1.324667	10992000	-0.0271	0.011334	56
...	...	...	...	...	...	...	...	
2022-09-26	271.829987	284.089996	270.309998	276.010010	58076900	0.7600	0.680023	50
2022-09-27	283.839996	288.670013	277.510010	282.940002	61925200	0.7600	6.929993	49
2022-09-28	283.079987	289.000000	277.570007	287.809998	54664800	0.7600	4.869995	49
2022-09-29	282.760010	283.649994	265.779999	268.209991	77620600	0.7600	-19.600006	34
2022-09-30	266.149994	275.570007	262.470001	265.250000	67726600	0.7600	-2.959991	30

3048 rows × 20 columns

In [28]:
```
features = X_train.columns
```

Training the model

In [29]:
```
# Initialize the random forest regressor
rf_model2 = RandomForestRegressor(n_estimators=17, random_state=42, n_jobs=-1)

# Fit the model on the training data
rf_model2.fit(X_train, y_train.values.ravel())

# Make predictions on the testing data
y_pred = rf_model2.predict(X_test)
```

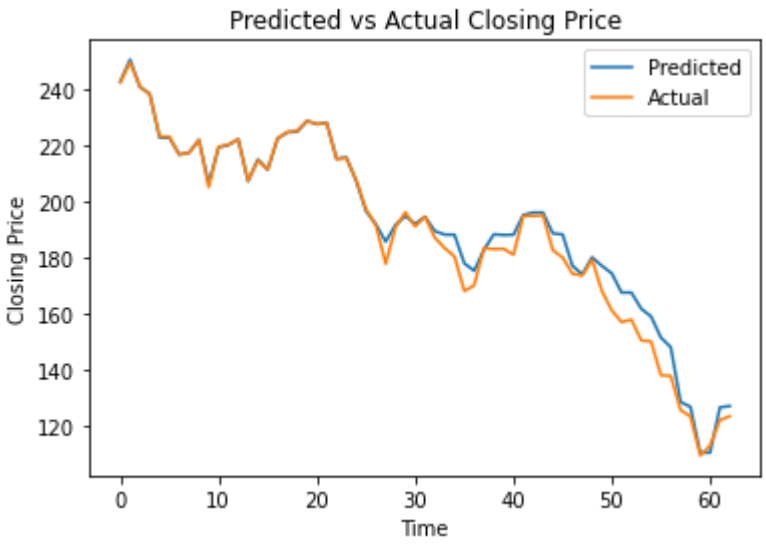
In [ ]:
```
```

In [30]:
```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 24.10017321535006
```

On average, the model's predictions deviate from the actual prices by about 4.91 dollars. This is about 6.55 times higher than in the case of the Coca-Cola stock (0.75 dollars); however the scale of prices is also higher from 2020, reaching up to 400 dollars with high volatility (which is more than x6 times the highest value of the Coca-Cola stock price, around 60 dollars). Thus, it seems that this second random forest model is performing comparably to the first one, especially taking into account the larg volatility of TSLA stock.

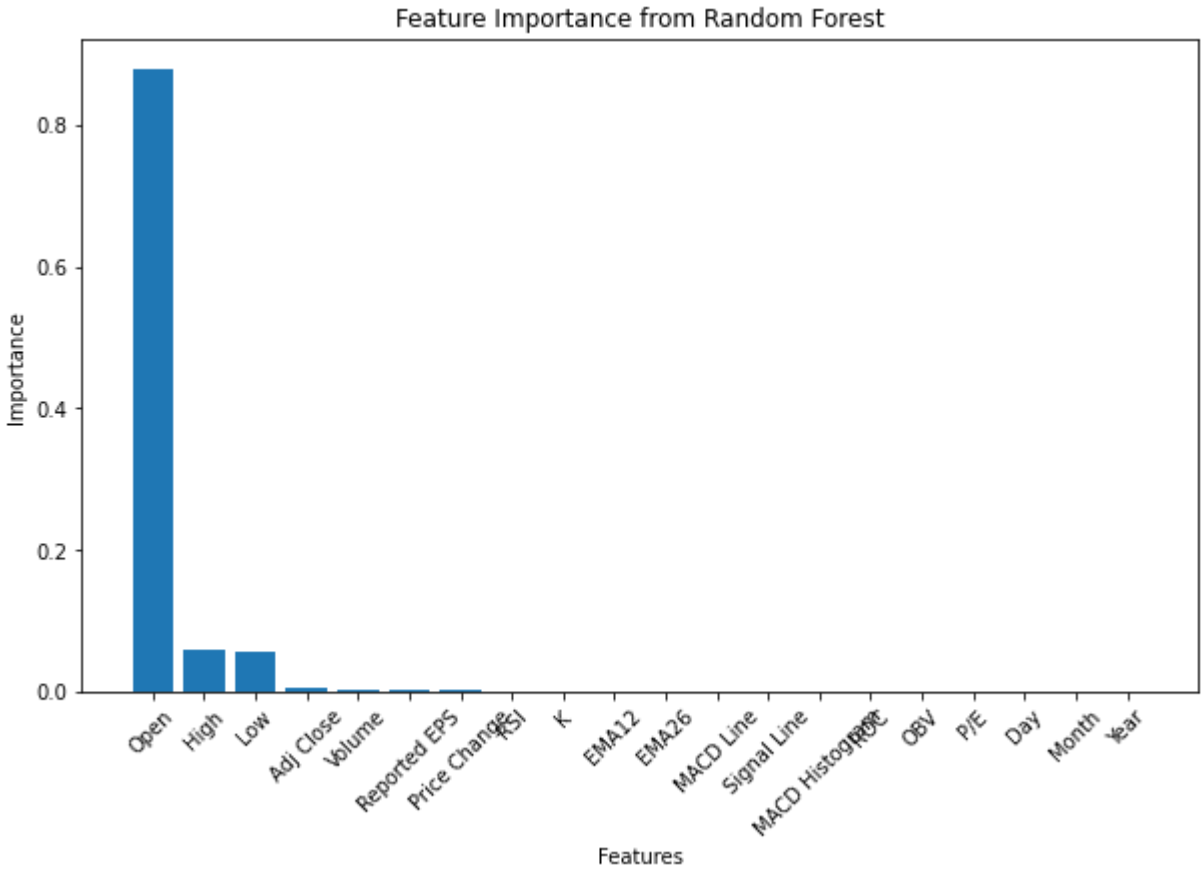
In [31]:
```
plot_predictions(y_pred, y_test)
```



We observe a perfect match up to day 25. Afterwards the predicted time series captures the general contracting trend, but showing values between 5% and 10% higher then the actual data over certain intermediate periods of time. Nevertheless, both predicted and actual time series end up convegenging during the last days within the test set.

Variable importance

In [32]:
```
importances = rf_model2.feature_importances_

sorted_idx = importances.argsort()[::-1]

plt.figure(figsize=(10,6))
plt.bar(features, importances[sorted_idx])
plt.xlabel('Features')
plt.ylabel('Importance')
plt.title('Feature Importance from Random Forest')
plt.xticks(rotation=45)
plt.show()
```



In this case most of the feature importance is concentrated in the `Open` price (0.86), followed by `High` and `Low` (with around 0.7). The rest of features have essentially insignificant importance.

We find that, if we only consider the subset of features `Open`, `High`, `Low` and `Adj Close` the MSE is drastically minimized from 24.10 to 6.70.

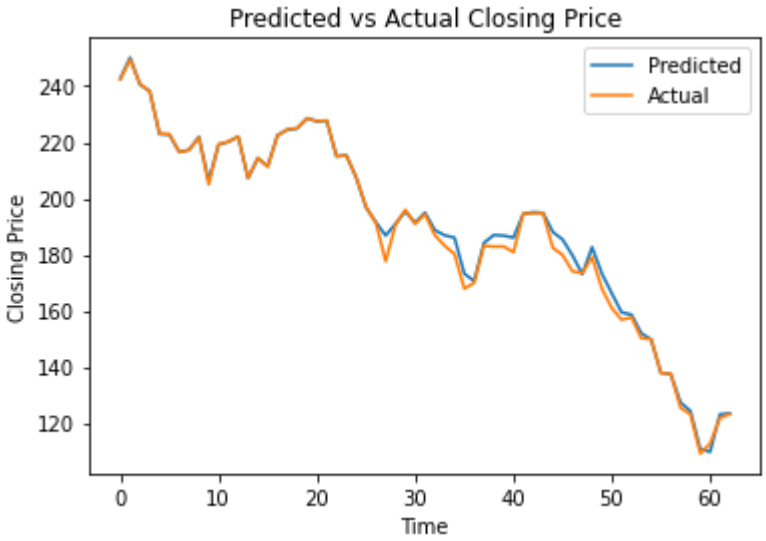
In [33]:
```
X_train = X_train[['Open','High', 'Low','Adj Close']]
X_test = X_test[['Open','High', 'Low','Adj Close']]

# Fit the model on the training data
rf_model2.fit(X_train, y_train.values.ravel())
```

```
# Make predictions on the testing data
y_pred = rf_model2.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 6.700998439567296
```

Therefore we focus in the sequel on this model with truncated number of features.

In [34]:
```
plot_predictions(y_pred, y_test)
```



Cross-validation

In [35]:
```
# Number of splits. For example, if n_splits=5, it will produce 5 train/test sets.
tscv = TimeSeriesSplit(n_splits=5)

# Initialize the Random Forest Regressor
model = RandomForestRegressor()

mse_scores = []

# Perform the time series cross-validation
for train_index, test_index in tscv.split(X_train):
    X_train_cv, X_test_cv = X_train.iloc[train_index], X_train.iloc[test_index]
    y_train_cv, y_test_cv = y_train.iloc[train_index], y_train.iloc[test_index]


    model.fit(X_train, y_train.values.ravel())
    y_pred = model.predict(X_test)

    mse = mean_squared_error(y_test, y_pred)
    mse_scores.append(mse)

# Calculate the mean and standard deviation of MSE across folds
mean_mse = np.mean(mse_scores)
std_mse = np.std(mse_scores)
```

```
print(f"Mean MSE: {mean_mse:.4f} (+/- {std_mse:.4f})")
```

```
Mean MSE: 6.5567 (+/- 0.2536)
```

The MSE from the test set is quite close to the mean MSE from cross-validation, with a standard devition of around 8% of the mean MSE. This suggests that the model performs consistently on different subsets of the data and generalizes fairly well to unseen data. On average, the model has a squared error of about 6.38 units, and this error can vary by about 0.53 units across different data splits.

Hyperparameter tuning

In this case we can expect the possibility of improving the MSE of our model by hyperparameter tuning up to at least 5.95. Again, we should assess if the complexity of the new model is worth the reduction of MSE.

Let us perform an exhaustive `RandomizedSearchCV` picking 5000 random combinations of hyperparameters values.

In [36]:
```
param_dist = {
    'n_estimators': np.arange(3, 100),
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth': [None] + list(np.arange(1, 30)),
    'min_samples_split': np.arange(2, 50),
    'min_samples_leaf': np.arange(1, 50)
}

#  RandomForestRegressor instance
rf = RandomForestRegressor()

# RandomizedSearchCV
random_search = RandomizedSearchCV(
    rf, param_distributions=param_dist, n_iter=5000,
    scoring='neg_mean_squared_error', cv=tscv,
    verbose=1, n_jobs=-1, random_state=42
)

# Fit the model
random_search.fit(X_train, y_train.values.ravel())

# Print the best parameters
print(random_search.best_params_)
```

```
Fitting 5 folds for each of 5000 candidates, totalling 25000 fits
{'n_estimators': 24, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features':
'sqrt', 'max_depth': 22}
```

In [ ]:
```
```

In [37]:
```
# Initialize the random forest regressor
rf_model = RandomForestRegressor(n_estimators=18, min_samples_split=5, min_samples_
```

```
# Fit the model on the training data
rf_model.fit(X_train, y_train.values.ravel())

# Make predictions on the testing data
y_pred = rf_model.predict(X_test)
```

In [38]:
```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 4.914689431382644
```

We find that without further introducing much complexity we can reduce the MSE to 4.91. This only requires adding one more tree to the forest, increasing `min_samples_split` from 2 to 5 and imposing a reasonable maximal depth of 11.

Long-Short-Term-Memory neural network

LSTM networks are a type of Recurrent Neural Network (RNN) that are designed to remember patterns in sequential data over long-term time. Introduced by Hochreiter and Schmidhuber in 1997, LSTMs were designed to overcome the limitations of traditional RNNs, namely the issues related to the vanishing and exploding gradient problems. These challenges made it difficult for RNNs to capture long-term dependencies in sequence data, a problem that LSTMs solve.

The key innovation in LSTMs is the introduction of memory cells and three multiplicative gates: the input, forget, and output gates. These components allow LSTMs to regulate the flow of information, determining what to retain, discard, or pass on to the next time step. As a result, LSTMs can effectively learn and remember patterns over long sequences, making them especially suitable for tasks involving sequential data such as time series forecasting, natural language processing, and speech recognition.

Here we apply LSTMs to the problem of stock price forecasting. LSTMs can model intricate patterns and dependencies in historical stock price data, which might be overlooked or deemed infeasible by traditional models. By leveraging their proficiency in handling sequential data and effectively capture long-term dependencies, LSTMs have proved to be an excellent tool in forecasting future stock prices with high degree of accuracy.

Concretely, our goal is to make predictions for the `Close` column in our dataset. Due to the nature of LSTM, past values of the `Close` variable (appropriately preprocessed) will be considered as input features for making the prediction. This means that the `Close` variable will act both as a feature and the target, in contradistinction with typical machine learning scenarios where features and the target variable are clearly separated. This requires a further preprocessing of the data based on creating sequences of a certain number of past observations to make the prediction, which we will explain later on.

We will consider two scenarios: feeding the LSTM only just past values of `Close` (univariate) and the itnroduction of additional features (multivariate).

Univariate LSTM

Since the performance of LSTMs are sensitive to feature scaling, the first step is to perform a train/test split and rescale our dataset. We define a function that takes as input our dataset filename and returns the dataframe, the scaled and unscaled versions of the train and test sets constructed out of just the `Close` column (using a 80-20 split) and the scaler, that we will need to unscale our predictions in the end.

In [2]:
```
from sklearn.preprocessing import MinMaxScaler
import pandas as pd
import numpy as np

# This function reads our csv file and returns the dataframe, train set, test set,

def scale_dataframe(filename):
    # Load data into DataFrame
    df = pd.read_csv(filename)

    # Compute the number of data points for 80% of the data
    train_size = int(0.8 * len(df))

    # Split into train and test sets
    train = df[['Close']][:train_size]
    test = df[['Close']][train_size:]

    # Scale the features and target together
    scaler = MinMaxScaler(feature_range=(0,1))

    # Fit the scaler using only the training data and transform
    train_scaled = pd.DataFrame(scaler.fit_transform(train), columns=['Close'])

    # Scale the test data
    test_scaled = pd.DataFrame(scaler.transform(test), columns=['Close'])

    return df, train, test, train_scaled, test_scaled, scaler
```

In [3]:
```
df, train, test, train_scaled, test_scaled, scaler = scale_dataframe('KO.csv')
```

In order to train our LSTM network, we need a further preprocessing of our data that will render it into a suitable format that can be fed to the network. In time series forecasting with LSTM, it is common to use a certain number of previous steps to predict the next step. This is often referred to as a 'sliding window' approach.

We will define a function `create_dataset` that creates this sliding window, taking a time series dataset and a parameter `look_back` as inputs. The `look_back` parameter is the number of previous time steps to use as input variables for the prediction. The function will return two outputs: a 2D numpy array `X` where each row corresponds to a sequence of

`look_back` number of previous values from the dataset, and a 1D numpy array `Y` where each value corresponds to the next value in the dataset that follows the sequence in `X`. `X` contains the input features for our LSTM model, whereas `Y` is the target variable that our LSTM model will aim to predict.

In [3]:
```
def create_sequences(data, seq_length):
    xs = []
    ys = []

    for i in range(len(data)-seq_length-1):
        x = data.iloc[i:(i+seq_length)].values
        y = data.iloc[i+seq_length]  # removed .values
        xs.append(x)
        ys.append(y)

    return np.array(xs), np.array(ys)

# Specify the sequence length
seq_length = 60
```

Now we use the previosuly defined function to create the actual sequences from the train and test sets:

In [5]:
```
# Create sequences from train and test data
X_train, y_train = create_sequences(train_scaled['Close'], seq_length)
X_test, y_test = create_sequences(test_scaled['Close'], seq_length)
```

There is yet another subtlety we should take care of. The LSTM layer in Keras expects an input in the form of a 3D array, where the dimensions represent:

1. batch_size: the number of samples in a batch.
2. time_steps: the number of time steps in a single input sequence. For time series data, this would be the length of a sequence.
3. features: the number of features at each time step.

We must ensure that our data is in the right shape for the LSTM to process it.

In [6]:
```
# Reshape the inputs to be suitable for LSTM
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))
```

By reshaping with (X_train.shape[0], X_train.shape[1], 1), we are essentially adding an extra dimension to the data to indicate that there is just one feature at every time step.

Now we are ready to define our LSTM model. In order to do so we need to import the relevant TensorFlow and Keras libraries.

In [4]:
```
import tensorflow as tf

# Set a random seed for reproducibility of results
```

```
np.random.seed(42)
tf.random.set_seed(42)

from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout
from pandas.tseries.offsets import DateOffset
```

```
2023-08-17 21:07:17.861009: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find
cuda drivers on your machine, GPU will not be used.
2023-08-17 21:07:17.940710: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find
cuda drivers on your machine, GPU will not be used.
2023-08-17 21:07:17.942641: I tensorflow/core/platform/cpu_feature_guard.cc:182] Thi
s TensorFlow binary is optimized to use available CPU instructions in performance-cr
itical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorF
low with the appropriate compiler flags.
2023-08-17 21:07:19.884664: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38]
TF-TRT Warning: Could not find TensorRT
```

Our LSTM model architecture will consist of three LSTM layers of 50 units, each of them followed by a dropout layer to prevent overfitting. The final layer is a dense layer that outputs into a single continuous value, which represent the prediction in a forecasting task (recall we are doing a one-day-ahead prediction). Having in the dense layer a single unit and no activation function defined makes the model suitable for regression, as this is what produces a single continuous value as output.

We will define a class LSTMModel that encapsulates the architecture of our LSTM neural network, as we will need to use several instances of the model.

In [5]:
```
class LSTMModel:
    # Call the '__init__' method to perform initializations and set up attributes
    def __init__(self, input_shape=(None,1)):
        self.input_shape = input_shape

        # Call the '_build_model' method to construct the actual LSTM and store it
        self.model = self._build_model()

    # Define the architecture of the LSTM neural network
    def _build_model(self):
        model = Sequential()

        model.add(LSTM(units=50, return_sequences=True, input_shape=self.input_shap
        model.add(Dropout(0.2))

        model.add(LSTM(units=50, return_sequences=True))
        model.add(Dropout(0.2))

        model.add(LSTM(units=50))
        model.add(Dropout(0.2))

        model.add(Dense(units=1))

        return model
```

Using the `summary()' method we can display the architecture of our model. It provides a quick overview of all the layers that make up our model, along with the shape and number of parameters (weights and biases) in each layer.

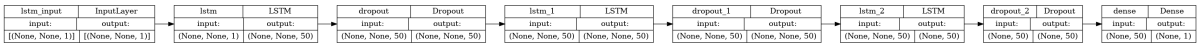
In [46]:
```
# Print out a summary of the model
lstm_model = LSTMModel((None, 1))
lstm_model.model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, None, 50)          10400

 dropout (Dropout)           (None, None, 50)          0

 lstm_1 (LSTM)               (None, None, 50)          20200

 dropout_1 (Dropout)         (None, None, 50)          0

 lstm_2 (LSTM)               (None, 50)                20200

 dropout_2 (Dropout)         (None, 50)                0

 dense (Dense)               (None, 1)                 51

=================================================================
Total params: 50851 (198.64 KB)
Trainable params: 50851 (198.64 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

We can plot the graphical representation of this network using `plot_model`:

In [47]:
```
from tensorflow.keras.utils import plot_model
plot_model(lstm_model.model, to_file='model.png', show_shapes=True, rankdir='LR')
```

Out[47]:



Now we need to configure our model for training using the `compile` method. We choose to work with the Adam optimizer and to minimize the Mean Squared Error.

In [48]:
```
lstm_model.model.compile(optimizer='adam', loss='mean_squared_error')
```

We will sue the `ModelCheckpoint` callback as a way to save the model at different points during training. The checkpointer in this code will monitor the model's validation loss during the training, and whenever the validation loss improves (namely, decreases), it saves the current weights of the model to the file 'weights_best.hdf5'.

In [6]:
```
from tensorflow.keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(
```

```
    filepath = 'weights_best.hdf5',
    verbose = 2,
    save_best_only = True
)
```

Finally we proceed to the training phase.

In [50]:
```
lstm_model.model.fit(
    X_train,
    y_train,
    epochs=25,
    batch_size = 32,
    callbacks = [checkpointer]
)
```

```
Epoch 1/25
167/167 [==============================] - ETA: 0s - loss: 0.0130WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 19s 79ms/step - loss: 0.0130
Epoch 2/25
167/167 [==============================] - ETA: 0s - loss: 0.0038WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0038
Epoch 3/25
167/167 [==============================] - ETA: 0s - loss: 0.0033WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0033
Epoch 4/25
167/167 [==============================] - ETA: 0s - loss: 0.0029WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0029
Epoch 5/25
167/167 [==============================] - ETA: 0s - loss: 0.0029WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0029
Epoch 6/25
167/167 [==============================] - ETA: 0s - loss: 0.0025WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 81ms/step - loss: 0.0025
Epoch 7/25
167/167 [==============================] - ETA: 0s - loss: 0.0023WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 81ms/step - loss: 0.0023
Epoch 8/25
167/167 [==============================] - ETA: 0s - loss: 0.0023WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0023
Epoch 9/25
167/167 [==============================] - ETA: 0s - loss: 0.0021WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0021
Epoch 10/25
167/167 [==============================] - ETA: 0s - loss: 0.0019WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 81ms/step - loss: 0.0019
Epoch 11/25
167/167 [==============================] - ETA: 0s - loss: 0.0019WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0019
Epoch 12/25
167/167 [==============================] - ETA: 0s - loss: 0.0017WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0017
Epoch 13/25
167/167 [==============================] - ETA: 0s - loss: 0.0019WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0019
Epoch 14/25
167/167 [==============================] - ETA: 0s - loss: 0.0016WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0016
```

```
Epoch 15/25
167/167 [==============================] - ETA: 0s - loss: 0.0015WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 14s 83ms/step - loss: 0.0015
Epoch 16/25
167/167 [==============================] - ETA: 0s - loss: 0.0015WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 14s 83ms/step - loss: 0.0015
Epoch 17/25
167/167 [==============================] - ETA: 0s - loss: 0.0014WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 14s 83ms/step - loss: 0.0014
Epoch 18/25
167/167 [==============================] - ETA: 0s - loss: 0.0013WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 14s 81ms/step - loss: 0.0013
Epoch 19/25
167/167 [==============================] - ETA: 0s - loss: 0.0013WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 81ms/step - loss: 0.0013
Epoch 20/25
167/167 [==============================] - ETA: 0s - loss: 0.0013WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 81ms/step - loss: 0.0013
Epoch 21/25
167/167 [==============================] - ETA: 0s - loss: 0.0012WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0012
Epoch 22/25
167/167 [==============================] - ETA: 0s - loss: 0.0011WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 14s 82ms/step - loss: 0.0011
Epoch 23/25
167/167 [==============================] - ETA: 0s - loss: 0.0011WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 14s 81ms/step - loss: 0.0011
Epoch 24/25
167/167 [==============================] - ETA: 0s - loss: 0.0011WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 14s 81ms/step - loss: 0.0011
Epoch 25/25
167/167 [==============================] - ETA: 0s - loss: 0.0011WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
167/167 [==============================] - 13s 80ms/step - loss: 0.0011
```

Out[50]: `<keras.src.callbacks.History at 0x7fc9c9689c70>`

We can now make the pediction on our test set and rescale it back to the scale of the original dataset.

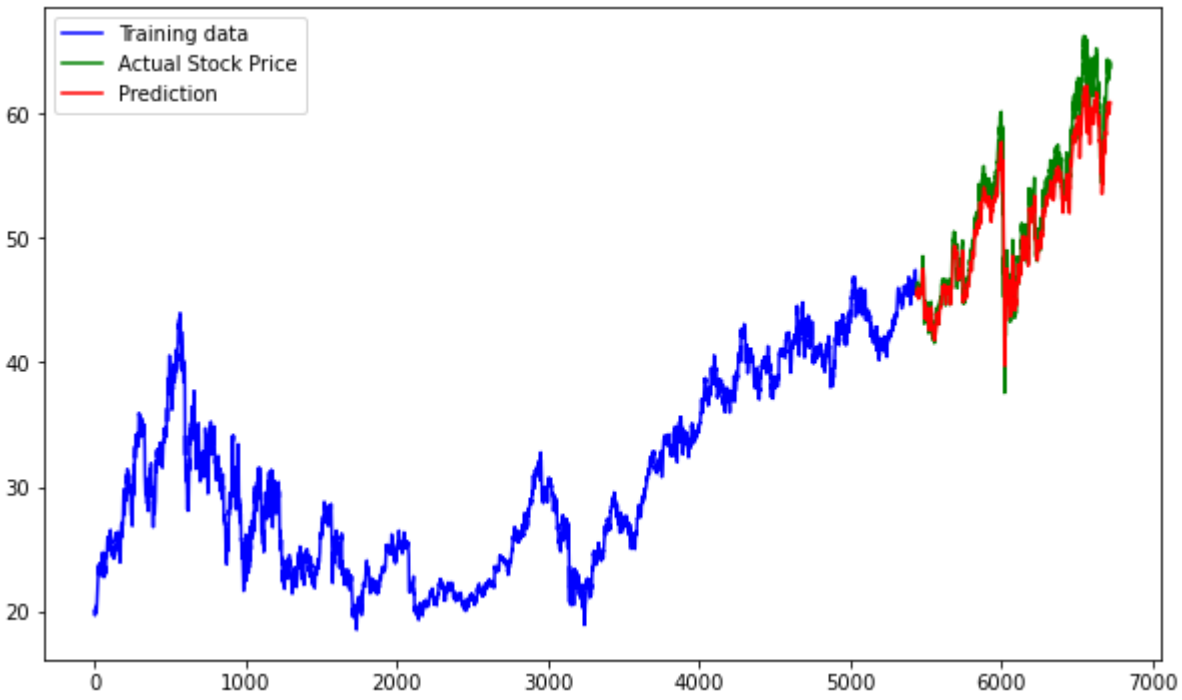
In [51]:
```
# Compute the predicted values on the test set
test_predict = lstm_model.model.predict(X_test)
# Undo the scaling transformation on the predicted values 'test_predict' and the ac
test_predict = scaler.inverse_transform(test_predict)
```

```
41/41 [==============================] - 3s 31ms/step
```

In [52]:
```
y_test_reshaped = y_test.reshape(-1, 1)
y_test = scaler.inverse_transform(y_test_reshaped)
```

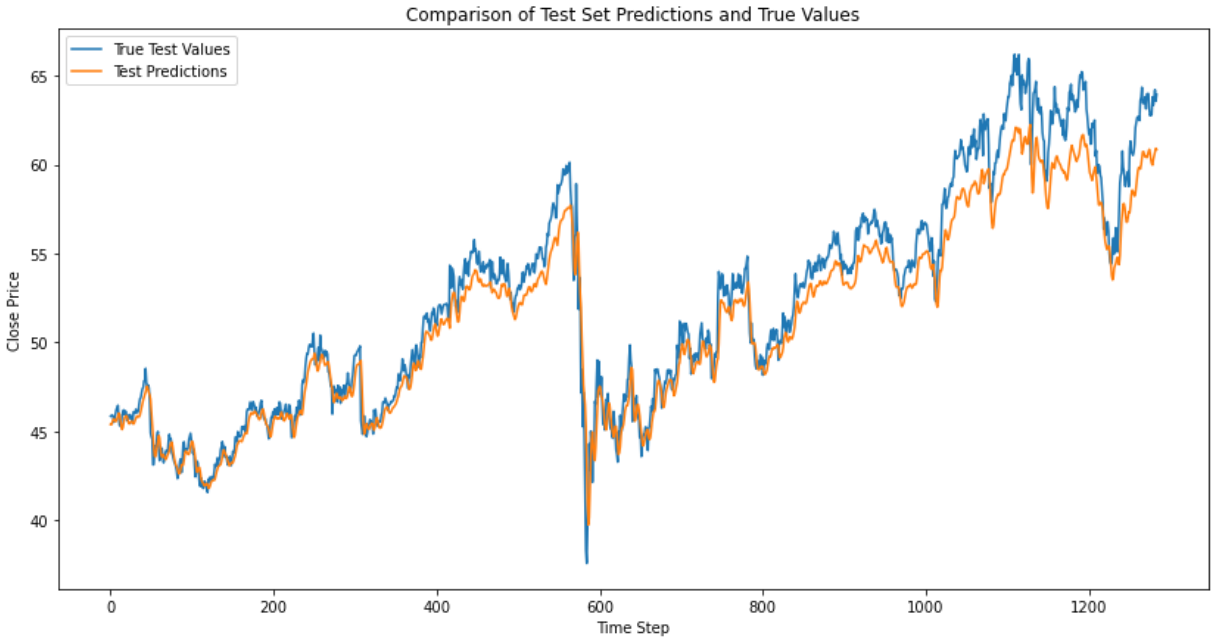
Finally we plot our prediction versus the actual time-series and compute the MSE.

In [53]:
```
plt.figure(figsize=(10,6))
plt.plot(df.index[:len(train)+seq_length], df['Close'][:len(train)+seq_length], col
plt.plot(df.index[len(train)+seq_length:-1], df['Close'][len(train)+seq_length:-1],
plt.plot(df.index[len(train)+seq_length:-1], test_predict, color='red', label='Pred
plt.legend()
plt.show()
```



Focusing on the test set:

In [54]:
```
# Plot test predictions and actual values
plt.figure(figsize=(14,7))
plt.plot(y_test, label='True Test Values')
plt.plot(test_predict, label='Test Predictions')
plt.title('Comparison of Test Set Predictions and True Values')
plt.xlabel('Time Step')
plt.ylabel('Close Price')
plt.legend()
plt.show()
```



Visually we already notice a remarkably good match. Let us compute the Mean Square Error:

In [55]:
```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, test_predict)
print(mse)
```

```
2.6559097200586157
```

This is a quite small error, especially for the timeframe we are considering (more than 1200 days, in other words). Recall that our Random Forest model yielded a similar error but for a test set 10 times smaller, which clearly indicates the superiority of the LSTM model in learning stock price time series.

There is yet one caveat: despite this great ability to learn time series data, it might be that our model is suffering from overfitting. We will perform a cross-validation analysis to study possible overfitting issues.

Cross validation

The following function performs a time series cross validation of our LSTM model

In [56]:
```
from sklearn.model_selection import TimeSeriesSplit

def train_lstm_cv(data, seq_length, model_function, n_splits=5):
    """
    Perform TimeSeries cross-validation on the LSTM model.

    Args:
    - data (pd.Series): The entire time series data.
    - seq_length (int): Length of input sequences.
    - model_function (function): A function to initialize and compile the LSTM mode
    - n_splits (int): Number of splits for cross-validation.
```

```
    Returns:
    - history_list (list): List of training histories.
    - mse_values (list): List of mean squared errors for each fold.
    """
    tscv = TimeSeriesSplit(n_splits=n_splits)
    history_list = []
    mse_values = []

    for train_index, test_index in tscv.split(data):

        # Split the data into train and test sets for this fold
        train_data, test_data = data.iloc[train_index], data.iloc[test_index]

        # Scale the training data for this fold and transform the test data
        scaler = MinMaxScaler(feature_range=(0,1))
        train_scaled = scaler.fit_transform(train_data.values.reshape(-1, 1))
        test_scaled = scaler.transform(test_data.values.reshape(-1, 1))

        # Create sequences for this fold's scaled data
        X_train, y_train = create_sequences(pd.DataFrame(train_scaled), seq_length)
        X_test, y_test = create_sequences(pd.DataFrame(test_scaled), seq_length)

        # Adjust the shape for LSTM input
        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))

        # Use the provided model_function to get the model
        lstm_model = model_function((X_train.shape[1], 1))
        lstm_model.model.compile(optimizer='adam', loss='mean_squared_error')

        checkpointer = ModelCheckpoint(
            filepath='weights_best_cv.hdf5',
            verbose=2,
            save_best_only=True
        )

        # Train the model
        history = lstm_model.model.fit(
            X_train,
            y_train,
            validation_data=(X_test, y_test),
            epochs=25,
            batch_size=32,
            callbacks=[checkpointer]
        )

        history_list.append(history)

        # Use the trained model to predict on the test set
        predictions = lstm_model.model.predict(X_test)

        # Inverse transform the predictions and actual values
        y_test_unscaled = scaler.inverse_transform(y_test.reshape(-1, 1))
        predictions_unscaled = scaler.inverse_transform(predictions)

        # Calculate MSE for this fold using unscaled data
```

```
        mse = mean_squared_error(y_test_unscaled, predictions_unscaled)
        mse_values.append(mse)

    return history_list, mse_values


# Each item in 'history_list' contains the training history for one fold.
# We can analyze these histories to assess the performance of the LSTM across diffe
```

Let us perform the cross validation analysis on the train set:

In [57]:
```
history_list, mse_values = train_lstm_cv(train['Close'], 60, model_function=LSTMMod
```

```
Epoch 1/25
27/27 [==============================] - ETA: 0s - loss: 0.0519
Epoch 1: val_loss improved from inf to 0.01902, saving model to weights_best.hdf5
27/27 [==============================] - 9s 153ms/step - loss: 0.0519 - val_loss: 0.
0190
Epoch 2/25
 1/27 [>.............................] - ETA: 2s - loss: 0.0171
/home/sergio/anaconda3/lib/python3.9/site-packages/keras/src/engine/training.py:300
0: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This f
ile format is considered legacy. We recommend using instead the native Keras format,
e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
27/27 [==============================] - ETA: 0s - loss: 0.0101
Epoch 2: val_loss improved from 0.01902 to 0.00562, saving model to weights_best.hdf
5
27/27 [==============================] - 3s 111ms/step - loss: 0.0101 - val_loss: 0.
0056
Epoch 3/25
27/27 [==============================] - ETA: 0s - loss: 0.0073
Epoch 3: val_loss improved from 0.00562 to 0.00302, saving model to weights_best.hdf
5
27/27 [==============================] - 3s 111ms/step - loss: 0.0073 - val_loss: 0.
0030
Epoch 4/25
27/27 [==============================] - ETA: 0s - loss: 0.0066
Epoch 4: val_loss improved from 0.00302 to 0.00281, saving model to weights_best.hdf
5
27/27 [==============================] - 3s 112ms/step - loss: 0.0066 - val_loss: 0.
0028
Epoch 5/25
27/27 [==============================] - ETA: 0s - loss: 0.0074
Epoch 5: val_loss did not improve from 0.00281
27/27 [==============================] - 3s 110ms/step - loss: 0.0074 - val_loss: 0.
0033
Epoch 6/25
27/27 [==============================] - ETA: 0s - loss: 0.0066
Epoch 6: val_loss improved from 0.00281 to 0.00228, saving model to weights_best.hdf
5
27/27 [==============================] - 3s 114ms/step - loss: 0.0066 - val_loss: 0.
0023
Epoch 7/25
27/27 [==============================] - ETA: 0s - loss: 0.0061
Epoch 7: val_loss did not improve from 0.00228
27/27 [==============================] - 3s 111ms/step - loss: 0.0061 - val_loss: 0.
0030
Epoch 8/25
27/27 [==============================] - ETA: 0s - loss: 0.0063
Epoch 8: val_loss did not improve from 0.00228
27/27 [==============================] - 3s 114ms/step - loss: 0.0063 - val_loss: 0.
0038
Epoch 9/25
27/27 [==============================] - ETA: 0s - loss: 0.0055
Epoch 9: val_loss did not improve from 0.00228
27/27 [==============================] - 3s 110ms/step - loss: 0.0055 - val_loss: 0.
0034
Epoch 10/25
27/27 [==============================] - ETA: 0s - loss: 0.0056
Epoch 10: val_loss did not improve from 0.00228
27/27 [==============================] - 3s 111ms/step - loss: 0.0056 - val_loss: 0.
0024
Epoch 11/25
27/27 [==============================] - ETA: 0s - loss: 0.0058
Epoch 11: val_loss did not improve from 0.00228
27/27 [==============================] - 3s 111ms/step - loss: 0.0058 - val_loss: 0.
0027
Epoch 12/25
27/27 [==============================] - ETA: 0s - loss: 0.0050
Epoch 12: val_loss improved from 0.00228 to 0.00220, saving model to weights_best.hd
```

```
f5
27/27 [==============================] - 3s 112ms/step - loss: 0.0050 - val_loss: 0.
0022
Epoch 13/25
27/27 [==============================] - ETA: 0s - loss: 0.0044
Epoch 13: val_loss improved from 0.00220 to 0.00193, saving model to weights_best.hd
f5
27/27 [==============================] - 3s 113ms/step - loss: 0.0044 - val_loss: 0.
0019
Epoch 14/25
27/27 [==============================] - ETA: 0s - loss: 0.0054
Epoch 14: val_loss did not improve from 0.00193
27/27 [==============================] - 3s 110ms/step - loss: 0.0054 - val_loss: 0.
0020
Epoch 15/25
27/27 [==============================] - ETA: 0s - loss: 0.0049
Epoch 15: val_loss did not improve from 0.00193
27/27 [==============================] - 3s 111ms/step - loss: 0.0049 - val_loss: 0.
0028
Epoch 16/25
27/27 [==============================] - ETA: 0s - loss: 0.0050
Epoch 16: val_loss did not improve from 0.00193
27/27 [==============================] - 3s 112ms/step - loss: 0.0050 - val_loss: 0.
0020
Epoch 17/25
27/27 [==============================] - ETA: 0s - loss: 0.0046
Epoch 17: val_loss improved from 0.00193 to 0.00174, saving model to weights_best.hd
f5
27/27 [==============================] - 3s 112ms/step - loss: 0.0046 - val_loss: 0.
0017
Epoch 18/25
27/27 [==============================] - ETA: 0s - loss: 0.0043
Epoch 18: val_loss did not improve from 0.00174
27/27 [==============================] - 3s 110ms/step - loss: 0.0043 - val_loss: 0.
0029
Epoch 19/25
27/27 [==============================] - ETA: 0s - loss: 0.0045
Epoch 19: val_loss did not improve from 0.00174
27/27 [==============================] - 3s 110ms/step - loss: 0.0045 - val_loss: 0.
0020
Epoch 20/25
27/27 [==============================] - ETA: 0s - loss: 0.0043
Epoch 20: val_loss did not improve from 0.00174
27/27 [==============================] - 3s 111ms/step - loss: 0.0043 - val_loss: 0.
0019
Epoch 21/25
27/27 [==============================] - ETA: 0s - loss: 0.0043
Epoch 21: val_loss did not improve from 0.00174
27/27 [==============================] - 3s 109ms/step - loss: 0.0043 - val_loss: 0.
0017
Epoch 22/25
27/27 [==============================] - ETA: 0s - loss: 0.0041
Epoch 22: val_loss did not improve from 0.00174
27/27 [==============================] - 3s 110ms/step - loss: 0.0041 - val_loss: 0.
0024
Epoch 23/25
```

```
27/27 [==============================] - ETA: 0s - loss: 0.0042
Epoch 23: val_loss improved from 0.00174 to 0.00165, saving model to weights_best.hd
f5
27/27 [==============================] - 3s 113ms/step - loss: 0.0042 - val_loss: 0.
0017
Epoch 24/25
27/27 [==============================] - ETA: 0s - loss: 0.0041
Epoch 24: val_loss did not improve from 0.00165
27/27 [==============================] - 3s 110ms/step - loss: 0.0041 - val_loss: 0.
0017
Epoch 25/25
27/27 [==============================] - ETA: 0s - loss: 0.0042
Epoch 25: val_loss improved from 0.00165 to 0.00161, saving model to weights_best.hd
f5
27/27 [==============================] - 3s 113ms/step - loss: 0.0042 - val_loss: 0.
0016
27/27 [==============================] - 2s 30ms/step
Epoch 1/25
55/55 [==============================] - ETA: 0s - loss: 0.0208
Epoch 1: val_loss improved from inf to 0.00063, saving model to weights_best.hdf5
55/55 [==============================] - 11s 116ms/step - loss: 0.0208 - val_loss:
6.3271e-04
Epoch 2/25
 1/55 [..............................] - ETA: 4s - loss: 0.0101
/home/sergio/anaconda3/lib/python3.9/site-packages/keras/src/engine/training.py:300
0: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This f
ile format is considered legacy. We recommend using instead the native Keras format,
e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
55/55 [==============================] - ETA: 0s - loss: 0.0060
Epoch 2: val_loss improved from 0.00063 to 0.00061, saving model to weights_best.hdf
5
55/55 [==============================] - 5s 96ms/step - loss: 0.0060 - val_loss: 6.0
738e-04
Epoch 3/25
55/55 [==============================] - ETA: 0s - loss: 0.0061
Epoch 3: val_loss improved from 0.00061 to 0.00048, saving model to weights_best.hdf
5
55/55 [==============================] - 5s 97ms/step - loss: 0.0061 - val_loss: 4.8
266e-04
Epoch 4/25
55/55 [==============================] - ETA: 0s - loss: 0.0048
Epoch 4: val_loss did not improve from 0.00048
55/55 [==============================] - 5s 95ms/step - loss: 0.0048 - val_loss: 5.4
701e-04
Epoch 5/25
55/55 [==============================] - ETA: 0s - loss: 0.0042
Epoch 5: val_loss improved from 0.00048 to 0.00044, saving model to weights_best.hdf
5
55/55 [==============================] - 5s 97ms/step - loss: 0.0042 - val_loss: 4.3
752e-04
Epoch 6/25
55/55 [==============================] - ETA: 0s - loss: 0.0041
Epoch 6: val_loss did not improve from 0.00044
55/55 [==============================] - 5s 95ms/step - loss: 0.0041 - val_loss: 5.6
461e-04
Epoch 7/25
55/55 [==============================] - ETA: 0s - loss: 0.0043
Epoch 7: val_loss improved from 0.00044 to 0.00041, saving model to weights_best.hdf
5
55/55 [==============================] - 5s 96ms/step - loss: 0.0043 - val_loss: 4.1
123e-04
Epoch 8/25
55/55 [==============================] - ETA: 0s - loss: 0.0036
Epoch 8: val_loss did not improve from 0.00041
55/55 [==============================] - 5s 96ms/step - loss: 0.0036 - val_loss: 6.1
931e-04
Epoch 9/25
55/55 [==============================] - ETA: 0s - loss: 0.0038
Epoch 9: val_loss improved from 0.00041 to 0.00036, saving model to weights_best.hdf
5
55/55 [==============================] - 5s 97ms/step - loss: 0.0038 - val_loss: 3.5
764e-04
Epoch 10/25
55/55 [==============================] - ETA: 0s - loss: 0.0038
Epoch 10: val_loss improved from 0.00036 to 0.00033, saving model to weights_best.hd
f5
55/55 [==============================] - 5s 97ms/step - loss: 0.0038 - val_loss: 3.3
279e-04
Epoch 11/25
55/55 [==============================] - ETA: 0s - loss: 0.0034
Epoch 11: val_loss did not improve from 0.00033
55/55 [==============================] - 5s 96ms/step - loss: 0.0034 - val_loss: 0.0
010
Epoch 12/25
```

```
55/55 [==============================] - ETA: 0s - loss: 0.0034
Epoch 12: val_loss did not improve from 0.00033
55/55 [==============================] - 5s 95ms/step - loss: 0.0034 - val_loss: 3.6
143e-04
Epoch 13/25
55/55 [==============================] - ETA: 0s - loss: 0.0032
Epoch 13: val_loss improved from 0.00033 to 0.00028, saving model to weights_best.hd
f5
55/55 [==============================] - 5s 97ms/step - loss: 0.0032 - val_loss: 2.8
387e-04
Epoch 14/25
55/55 [==============================] - ETA: 0s - loss: 0.0028
Epoch 14: val_loss improved from 0.00028 to 0.00026, saving model to weights_best.hd
f5
55/55 [==============================] - 5s 96ms/step - loss: 0.0028 - val_loss: 2.6
311e-04
Epoch 15/25
55/55 [==============================] - ETA: 0s - loss: 0.0030
Epoch 15: val_loss did not improve from 0.00026
55/55 [==============================] - 5s 96ms/step - loss: 0.0030 - val_loss: 6.9
558e-04
Epoch 16/25
55/55 [==============================] - ETA: 0s - loss: 0.0029
Epoch 16: val_loss did not improve from 0.00026
55/55 [==============================] - 5s 96ms/step - loss: 0.0029 - val_loss: 2.8
191e-04
Epoch 17/25
55/55 [==============================] - ETA: 0s - loss: 0.0026
Epoch 17: val_loss did not improve from 0.00026
55/55 [==============================] - 5s 95ms/step - loss: 0.0026 - val_loss: 2.6
830e-04
Epoch 18/25
55/55 [==============================] - ETA: 0s - loss: 0.0026
Epoch 18: val_loss did not improve from 0.00026
55/55 [==============================] - 5s 95ms/step - loss: 0.0026 - val_loss: 4.2
454e-04
Epoch 19/25
55/55 [==============================] - ETA: 0s - loss: 0.0026
Epoch 19: val_loss improved from 0.00026 to 0.00024, saving model to weights_best.hd
f5
55/55 [==============================] - 5s 97ms/step - loss: 0.0026 - val_loss: 2.3
701e-04
Epoch 20/25
55/55 [==============================] - ETA: 0s - loss: 0.0026
Epoch 20: val_loss improved from 0.00024 to 0.00022, saving model to weights_best.hd
f5
55/55 [==============================] - 5s 96ms/step - loss: 0.0026 - val_loss: 2.1
774e-04
Epoch 21/25
55/55 [==============================] - ETA: 0s - loss: 0.0026
Epoch 21: val_loss did not improve from 0.00022
55/55 [==============================] - 5s 96ms/step - loss: 0.0026 - val_loss: 2.6
239e-04
Epoch 22/25
55/55 [==============================] - ETA: 0s - loss: 0.0025
Epoch 22: val_loss improved from 0.00022 to 0.00022, saving model to weights_best.hd
```

```
f5
55/55 [==============================] - 5s 96ms/step - loss: 0.0025 - val_loss: 2.1
753e-04
Epoch 23/25
55/55 [==============================] - ETA: 0s - loss: 0.0025
Epoch 23: val_loss did not improve from 0.00022
55/55 [==============================] - 5s 96ms/step - loss: 0.0025 - val_loss: 2.3
067e-04
Epoch 24/25
55/55 [==============================] - ETA: 0s - loss: 0.0022
Epoch 24: val_loss did not improve from 0.00022
55/55 [==============================] - 5s 96ms/step - loss: 0.0022 - val_loss: 2.2
301e-04
Epoch 25/25
55/55 [==============================] - ETA: 0s - loss: 0.0022
Epoch 25: val_loss improved from 0.00022 to 0.00020, saving model to weights_best.hd
f5
55/55 [==============================] - 5s 97ms/step - loss: 0.0022 - val_loss: 2.0
090e-04
27/27 [==============================] - 2s 31ms/step
Epoch 1/25
83/83 [==============================] - ETA: 0s - loss: 0.0094
Epoch 1: val_loss improved from inf to 0.00108, saving model to weights_best.hdf5
83/83 [==============================] - 14s 105ms/step - loss: 0.0094 - val_loss:
0.0011
Epoch 2/25
 1/83 [..............................] - ETA: 7s - loss: 0.0085
/home/sergio/anaconda3/lib/python3.9/site-packages/keras/src/engine/training.py:300
0: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This f
ile format is considered legacy. We recommend using instead the native Keras format,
e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
83/83 [==============================] - ETA: 0s - loss: 0.0036
Epoch 2: val_loss improved from 0.00108 to 0.00096, saving model to weights_best.hdf
5
83/83 [==============================] - 8s 91ms/step - loss: 0.0036 - val_loss: 9.5
721e-04
Epoch 3/25
83/83 [==============================] - ETA: 0s - loss: 0.0034
Epoch 3: val_loss did not improve from 0.00096
83/83 [==============================] - 8s 91ms/step - loss: 0.0034 - val_loss: 0.0
011
Epoch 4/25
83/83 [==============================] - ETA: 0s - loss: 0.0030
Epoch 4: val_loss did not improve from 0.00096
83/83 [==============================] - 8s 91ms/step - loss: 0.0030 - val_loss: 0.0
014
Epoch 5/25
83/83 [==============================] - ETA: 0s - loss: 0.0027
Epoch 5: val_loss did not improve from 0.00096
83/83 [==============================] - 8s 91ms/step - loss: 0.0027 - val_loss: 0.0
013
Epoch 6/25
83/83 [==============================] - ETA: 0s - loss: 0.0026
Epoch 6: val_loss improved from 0.00096 to 0.00071, saving model to weights_best.hdf
5
83/83 [==============================] - 8s 91ms/step - loss: 0.0026 - val_loss: 7.1
482e-04
Epoch 7/25
83/83 [==============================] - ETA: 0s - loss: 0.0024
Epoch 7: val_loss did not improve from 0.00071
83/83 [==============================] - 8s 91ms/step - loss: 0.0024 - val_loss: 8.0
348e-04
Epoch 8/25
83/83 [==============================] - ETA: 0s - loss: 0.0022
Epoch 8: val_loss did not improve from 0.00071
83/83 [==============================] - 8s 91ms/step - loss: 0.0022 - val_loss: 0.0
016
Epoch 9/25
83/83 [==============================] - ETA: 0s - loss: 0.0023
Epoch 9: val_loss did not improve from 0.00071
83/83 [==============================] - 8s 91ms/step - loss: 0.0023 - val_loss: 0.0
012
Epoch 10/25
83/83 [==============================] - ETA: 0s - loss: 0.0021
Epoch 10: val_loss improved from 0.00071 to 0.00066, saving model to weights_best.hd
f5
83/83 [==============================] - 8s 92ms/step - loss: 0.0021 - val_loss: 6.5
976e-04
Epoch 11/25
83/83 [==============================] - ETA: 0s - loss: 0.0019
Epoch 11: val_loss did not improve from 0.00066
83/83 [==============================] - 8s 91ms/step - loss: 0.0019 - val_loss: 6.6
757e-04
Epoch 12/25
83/83 [==============================] - ETA: 0s - loss: 0.0020
Epoch 12: val_loss did not improve from 0.00066
83/83 [==============================] - 8s 91ms/step - loss: 0.0020 - val_loss: 8.0
```

```
615e-04
Epoch 13/25
83/83 [==============================] - ETA: 0s - loss: 0.0017
Epoch 13: val_loss improved from 0.00066 to 0.00058, saving model to weights_best.hd
f5
83/83 [==============================] - 8s 92ms/step - loss: 0.0017 - val_loss: 5.8
074e-04
Epoch 14/25
83/83 [==============================] - ETA: 0s - loss: 0.0018
Epoch 14: val_loss did not improve from 0.00058
83/83 [==============================] - 8s 91ms/step - loss: 0.0018 - val_loss: 8.7
626e-04
Epoch 15/25
83/83 [==============================] - ETA: 0s - loss: 0.0017
Epoch 15: val_loss improved from 0.00058 to 0.00052, saving model to weights_best.hd
f5
83/83 [==============================] - 8s 92ms/step - loss: 0.0017 - val_loss: 5.1
707e-04
Epoch 16/25
83/83 [==============================] - ETA: 0s - loss: 0.0015
Epoch 16: val_loss did not improve from 0.00052
83/83 [==============================] - 8s 91ms/step - loss: 0.0015 - val_loss: 5.3
927e-04
Epoch 17/25
83/83 [==============================] - ETA: 0s - loss: 0.0016
Epoch 17: val_loss improved from 0.00052 to 0.00050, saving model to weights_best.hd
f5
83/83 [==============================] - 8s 92ms/step - loss: 0.0016 - val_loss: 4.9
516e-04
Epoch 18/25
83/83 [==============================] - ETA: 0s - loss: 0.0015
Epoch 18: val_loss did not improve from 0.00050
83/83 [==============================] - 8s 92ms/step - loss: 0.0015 - val_loss: 5.0
249e-04
Epoch 19/25
83/83 [==============================] - ETA: 0s - loss: 0.0014
Epoch 19: val_loss improved from 0.00050 to 0.00046, saving model to weights_best.hd
f5
83/83 [==============================] - 8s 92ms/step - loss: 0.0014 - val_loss: 4.5
737e-04
Epoch 20/25
83/83 [==============================] - ETA: 0s - loss: 0.0014
Epoch 20: val_loss improved from 0.00046 to 0.00045, saving model to weights_best.hd
f5
83/83 [==============================] - 8s 92ms/step - loss: 0.0014 - val_loss: 4.4
851e-04
Epoch 21/25
83/83 [==============================] - ETA: 0s - loss: 0.0014
Epoch 21: val_loss did not improve from 0.00045
83/83 [==============================] - 8s 92ms/step - loss: 0.0014 - val_loss: 5.4
836e-04
Epoch 22/25
83/83 [==============================] - ETA: 0s - loss: 0.0013
Epoch 22: val_loss improved from 0.00045 to 0.00042, saving model to weights_best.hd
f5
83/83 [==============================] - 8s 91ms/step - loss: 0.0013 - val_loss: 4.2
```

```
384e-04
Epoch 23/25
83/83 [==============================] - ETA: 0s - loss: 0.0013
Epoch 23: val_loss did not improve from 0.00042
83/83 [==============================] - 8s 91ms/step - loss: 0.0013 - val_loss: 4.3
866e-04
Epoch 24/25
83/83 [==============================] - ETA: 0s - loss: 0.0013
Epoch 24: val_loss improved from 0.00042 to 0.00039, saving model to weights_best.hd
f5
83/83 [==============================] - 8s 92ms/step - loss: 0.0013 - val_loss: 3.9
421e-04
Epoch 25/25
83/83 [==============================] - ETA: 0s - loss: 0.0012
Epoch 25: val_loss improved from 0.00039 to 0.00039, saving model to weights_best.hd
f5
83/83 [==============================] - 8s 92ms/step - loss: 0.0012 - val_loss: 3.8
510e-04
27/27 [==============================] - 2s 32ms/step
Epoch 1/25
111/111 [==============================] - ETA: 0s - loss: 0.0098
Epoch 1: val_loss improved from inf to 0.00092, saving model to weights_best.hdf5
111/111 [==============================] - 16s 99ms/step - loss: 0.0098 - val_loss:
9.1708e-04
Epoch 2/25
  1/111 [..............................] - ETA: 9s - loss: 0.0040
/home/sergio/anaconda3/lib/python3.9/site-packages/keras/src/engine/training.py:300
0: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This f
ile format is considered legacy. We recommend using instead the native Keras format,
e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
111/111 [==============================] - ETA: 0s - loss: 0.0036
Epoch 2: val_loss did not improve from 0.00092
111/111 [==============================] - 10s 89ms/step - loss: 0.0036 - val_loss:
9.2193e-04
Epoch 3/25
111/111 [==============================] - ETA: 0s - loss: 0.0027
Epoch 3: val_loss improved from 0.00092 to 0.00070, saving model to weights_best.hdf
5
111/111 [==============================] - 10s 89ms/step - loss: 0.0027 - val_loss:
7.0282e-04
Epoch 4/25
111/111 [==============================] - ETA: 0s - loss: 0.0026
Epoch 4: val_loss did not improve from 0.00070
111/111 [==============================] - 10s 89ms/step - loss: 0.0026 - val_loss:
8.3425e-04
Epoch 5/25
111/111 [==============================] - ETA: 0s - loss: 0.0023
Epoch 5: val_loss did not improve from 0.00070
111/111 [==============================] - 10s 88ms/step - loss: 0.0023 - val_loss:
0.0018
Epoch 6/25
111/111 [==============================] - ETA: 0s - loss: 0.0020
Epoch 6: val_loss did not improve from 0.00070
111/111 [==============================] - 10s 89ms/step - loss: 0.0020 - val_loss:
0.0014
Epoch 7/25
111/111 [==============================] - ETA: 0s - loss: 0.0022
Epoch 7: val_loss did not improve from 0.00070
111/111 [==============================] - 10s 89ms/step - loss: 0.0022 - val_loss:
0.0035
Epoch 8/25
111/111 [==============================] - ETA: 0s - loss: 0.0019
Epoch 8: val_loss did not improve from 0.00070
111/111 [==============================] - 10s 89ms/step - loss: 0.0019 - val_loss:
9.3803e-04
Epoch 9/25
111/111 [==============================] - ETA: 0s - loss: 0.0018
Epoch 9: val_loss improved from 0.00070 to 0.00065, saving model to weights_best.hdf
5
111/111 [==============================] - 10s 89ms/step - loss: 0.0018 - val_loss:
6.5156e-04
Epoch 10/25
111/111 [==============================] - ETA: 0s - loss: 0.0018
Epoch 10: val_loss did not improve from 0.00065
111/111 [==============================] - 10s 89ms/step - loss: 0.0018 - val_loss:
9.1158e-04
Epoch 11/25
111/111 [==============================] - ETA: 0s - loss: 0.0017
Epoch 11: val_loss improved from 0.00065 to 0.00053, saving model to weights_best.hd
f5
111/111 [==============================] - 10s 89ms/step - loss: 0.0017 - val_loss:
5.3218e-04
Epoch 12/25
111/111 [==============================] - ETA: 0s - loss: 0.0016
Epoch 12: val_loss did not improve from 0.00053
111/111 [==============================] - 10s 88ms/step - loss: 0.0016 - val_loss:
```

```
0.0016
Epoch 13/25
111/111 [==============================] - ETA: 0s - loss: 0.0016
Epoch 13: val_loss did not improve from 0.00053
111/111 [==============================] - 10s 88ms/step - loss: 0.0016 - val_loss:
0.0022
Epoch 14/25
111/111 [==============================] - ETA: 0s - loss: 0.0014
Epoch 14: val_loss did not improve from 0.00053
111/111 [==============================] - 10s 89ms/step - loss: 0.0014 - val_loss:
0.0015
Epoch 15/25
111/111 [==============================] - ETA: 0s - loss: 0.0014
Epoch 15: val_loss did not improve from 0.00053
111/111 [==============================] - 10s 88ms/step - loss: 0.0014 - val_loss:
0.0013
Epoch 16/25
111/111 [==============================] - ETA: 0s - loss: 0.0013
Epoch 16: val_loss improved from 0.00053 to 0.00051, saving model to weights_best.hd
f5
111/111 [==============================] - 10s 89ms/step - loss: 0.0013 - val_loss:
5.1009e-04
Epoch 17/25
111/111 [==============================] - ETA: 0s - loss: 0.0013
Epoch 17: val_loss improved from 0.00051 to 0.00034, saving model to weights_best.hd
f5
111/111 [==============================] - 10s 89ms/step - loss: 0.0013 - val_loss:
3.4199e-04
Epoch 18/25
111/111 [==============================] - ETA: 0s - loss: 0.0012
Epoch 18: val_loss did not improve from 0.00034
111/111 [==============================] - 10s 89ms/step - loss: 0.0012 - val_loss:
3.7455e-04
Epoch 19/25
111/111 [==============================] - ETA: 0s - loss: 0.0012
Epoch 19: val_loss improved from 0.00034 to 0.00031, saving model to weights_best.hd
f5
111/111 [==============================] - 10s 89ms/step - loss: 0.0012 - val_loss:
3.0723e-04
Epoch 20/25
111/111 [==============================] - ETA: 0s - loss: 0.0012
Epoch 20: val_loss did not improve from 0.00031
111/111 [==============================] - 10s 89ms/step - loss: 0.0012 - val_loss:
4.5886e-04
Epoch 21/25
111/111 [==============================] - ETA: 0s - loss: 0.0011
Epoch 21: val_loss improved from 0.00031 to 0.00029, saving model to weights_best.hd
f5
111/111 [==============================] - 10s 90ms/step - loss: 0.0011 - val_loss:
2.8791e-04
Epoch 22/25
111/111 [==============================] - ETA: 0s - loss: 0.0010
Epoch 22: val_loss did not improve from 0.00029
111/111 [==============================] - 10s 88ms/step - loss: 0.0010 - val_loss:
7.6942e-04
Epoch 23/25
```

```
111/111 [==============================] - ETA: 0s - loss: 0.0010
Epoch 23: val_loss improved from 0.00029 to 0.00027, saving model to weights_best.hd
f5
111/111 [==============================] - 10s 89ms/step - loss: 0.0010 - val_loss:
2.7363e-04
Epoch 24/25
111/111 [==============================] - ETA: 0s - loss: 0.0010
Epoch 24: val_loss did not improve from 0.00027
111/111 [==============================] - 10s 88ms/step - loss: 0.0010 - val_loss:
8.6285e-04
Epoch 25/25
111/111 [==============================] - ETA: 0s - loss: 0.0010
Epoch 25: val_loss did not improve from 0.00027
111/111 [==============================] - 10s 89ms/step - loss: 0.0010 - val_loss:
6.4344e-04
27/27 [==============================] - 2s 31ms/step
Epoch 1/25
139/139 [==============================] - ETA: 0s - loss: 0.0087
Epoch 1: val_loss improved from inf to 0.00138, saving model to weights_best.hdf5
139/139 [==============================] - 19s 95ms/step - loss: 0.0087 - val_loss:
0.0014
Epoch 2/25
  1/139 [..............................] - ETA: 11s - loss: 0.0041
/home/sergio/anaconda3/lib/python3.9/site-packages/keras/src/engine/training.py:300
0: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This f
ile format is considered legacy. We recommend using instead the native Keras format,
e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
139/139 [==============================] - ETA: 0s - loss: 0.0034
Epoch 2: val_loss did not improve from 0.00138
139/139 [==============================] - 12s 87ms/step - loss: 0.0034 - val_loss:
0.0014
Epoch 3/25
139/139 [==============================] - ETA: 0s - loss: 0.0037
Epoch 3: val_loss did not improve from 0.00138
139/139 [==============================] - 12s 87ms/step - loss: 0.0037 - val_loss:
0.0014
Epoch 4/25
139/139 [==============================] - ETA: 0s - loss: 0.0029
Epoch 4: val_loss did not improve from 0.00138
139/139 [==============================] - 12s 87ms/step - loss: 0.0029 - val_loss:
0.0028
Epoch 5/25
139/139 [==============================] - ETA: 0s - loss: 0.0026
Epoch 5: val_loss did not improve from 0.00138
139/139 [==============================] - 12s 87ms/step - loss: 0.0026 - val_loss:
0.0024
Epoch 6/25
139/139 [==============================] - ETA: 0s - loss: 0.0023
Epoch 6: val_loss did not improve from 0.00138
139/139 [==============================] - 12s 87ms/step - loss: 0.0023 - val_loss:
0.0053
Epoch 7/25
139/139 [==============================] - ETA: 0s - loss: 0.0022
Epoch 7: val_loss did not improve from 0.00138
139/139 [==============================] - 12s 87ms/step - loss: 0.0022 - val_loss:
0.0047
Epoch 8/25
139/139 [==============================] - ETA: 0s - loss: 0.0023
Epoch 8: val_loss improved from 0.00138 to 0.00062, saving model to weights_best.hdf
5
139/139 [==============================] - 12s 88ms/step - loss: 0.0023 - val_loss:
6.2428e-04
Epoch 9/25
139/139 [==============================] - ETA: 0s - loss: 0.0019
Epoch 9: val_loss did not improve from 0.00062
139/139 [==============================] - 12s 87ms/step - loss: 0.0019 - val_loss:
9.6949e-04
Epoch 10/25
139/139 [==============================] - ETA: 0s - loss: 0.0020
Epoch 10: val_loss did not improve from 0.00062
139/139 [==============================] - 12s 87ms/step - loss: 0.0020 - val_loss:
0.0011
Epoch 11/25
139/139 [==============================] - ETA: 0s - loss: 0.0018
Epoch 11: val_loss did not improve from 0.00062
139/139 [==============================] - 12s 87ms/step - loss: 0.0018 - val_loss:
7.7698e-04
Epoch 12/25
139/139 [==============================] - ETA: 0s - loss: 0.0017
Epoch 12: val_loss did not improve from 0.00062
139/139 [==============================] - 12s 87ms/step - loss: 0.0017 - val_loss:
0.0039
Epoch 13/25
```

```
139/139 [==============================] - ETA: 0s - loss: 0.0016
Epoch 13: val_loss did not improve from 0.00062
139/139 [==============================] - 12s 88ms/step - loss: 0.0016 - val_loss:
0.0011
Epoch 14/25
139/139 [==============================] - ETA: 0s - loss: 0.0016
Epoch 14: val_loss did not improve from 0.00062
139/139 [==============================] - 12s 87ms/step - loss: 0.0016 - val_loss:
0.0011
Epoch 15/25
139/139 [==============================] - ETA: 0s - loss: 0.0015
Epoch 15: val_loss did not improve from 0.00062
139/139 [==============================] - 14s 103ms/step - loss: 0.0015 - val_loss:
0.0089
Epoch 16/25
139/139 [==============================] - ETA: 0s - loss: 0.0015
Epoch 16: val_loss did not improve from 0.00062
139/139 [==============================] - 15s 105ms/step - loss: 0.0015 - val_loss:
0.0011
Epoch 17/25
139/139 [==============================] - ETA: 0s - loss: 0.0013
Epoch 17: val_loss did not improve from 0.00062
139/139 [==============================] - 14s 98ms/step - loss: 0.0013 - val_loss:
8.1328e-04
Epoch 18/25
139/139 [==============================] - ETA: 0s - loss: 0.0013
Epoch 18: val_loss did not improve from 0.00062
139/139 [==============================] - 14s 100ms/step - loss: 0.0013 - val_loss:
6.4186e-04
Epoch 19/25
139/139 [==============================] - ETA: 0s - loss: 0.0013
Epoch 19: val_loss improved from 0.00062 to 0.00057, saving model to weights_best.hd
f5
139/139 [==============================] - 13s 93ms/step - loss: 0.0013 - val_loss:
5.7362e-04
Epoch 20/25
139/139 [==============================] - ETA: 0s - loss: 0.0013
Epoch 20: val_loss improved from 0.00057 to 0.00045, saving model to weights_best.hd
f5
139/139 [==============================] - 13s 92ms/step - loss: 0.0013 - val_loss:
4.5165e-04
Epoch 21/25
139/139 [==============================] - ETA: 0s - loss: 0.0012
Epoch 21: val_loss improved from 0.00045 to 0.00033, saving model to weights_best.hd
f5
139/139 [==============================] - 14s 103ms/step - loss: 0.0012 - val_loss:
3.2818e-04
Epoch 22/25
139/139 [==============================] - ETA: 0s - loss: 0.0012
Epoch 22: val_loss did not improve from 0.00033
139/139 [==============================] - 13s 94ms/step - loss: 0.0012 - val_loss:
4.5591e-04
Epoch 23/25
139/139 [==============================] - ETA: 0s - loss: 0.0011
Epoch 23: val_loss did not improve from 0.00033
139/139 [==============================] - 13s 93ms/step - loss: 0.0011 - val_loss:
```

```
4.7855e-04
Epoch 24/25
139/139 [==============================] - ETA: 0s - loss: 0.0010
Epoch 24: val_loss did not improve from 0.00033
139/139 [==============================] - 13s 94ms/step - loss: 0.0010 - val_loss:
8.0253e-04
Epoch 25/25
139/139 [==============================] - ETA: 0s - loss: 0.0011
Epoch 25: val_loss did not improve from 0.00033
139/139 [==============================] - 14s 99ms/step - loss: 0.0011 - val_loss:
6.9393e-04
27/27 [==============================] - 2s 34ms/step
```

Let's compute the average MSE and standard deviation:

In [58]:
```
# Compute average MSE and its standard deviation
average_mse = np.mean(mse_values)
mse_std = np.std(mse_values)

print(f"Average MSE across all folds: {average_mse:.4f}")
print(f"MSE standard deviation: {mse_std:.4f}")
```

```
Average MSE across all folds: 0.4390
MSE standard deviation: 0.2807
```

From these values, and recalling that the MSE on the test set is around 2.66, we can conclude the following:

1. The LSTM model, on average, performed better during cross-validation compared to its performance on the test set.

2. There is a relatively moderate variability in the performance across the different folds, as indicated by the standard deviation of the MSE values being similar to the mean MSE of cross validation. This suggests that the model might be sensitive to the specific portions of the dataset it is trained on, which might be a sign of the model slightly overfitting to particular patterns in specific subsets of the data.

3. The higher MSE on the test set compared to the average cross-validation MSE might indicate overfitting, or it might reflect that the test set has some characteristics or patterns that were not present in the training data. This could be due to the drastic drop due to the onset of the COVID pandemic.

We can have a look at the mean on the different folds:

In [59]:
```
mse_values
```

Out[59]:
```
[0.9507002120255099,
 0.12995407001454698,
 0.24911492146949116,
 0.41622265535807135,
 0.44888451762600645]
```

Indeed we observe a moderate variability in the performance on different folds. The MSE appears to follow a declining trend from the first fold (higher MSE of 0.95) to the second fold (lowest MSE of 0.12) with the highest MSE being approximately 8.5 times the lowest MSE, and then gradually increases from the second fold to the fifth.

A higher MSE in the first fold could be indicating underfitting. Underfitting is especially plausible to happen in the first fold of a time series cross-validation for several reasons. One reason is limited data: in the first fold of a time series cross-validation, the training dataset is the smallest. If the initial segment of data is not sufficiently diverse, the model might not learn the underlying temporal structures effectively. Other reason is changing dynamics: if the time series data has non-stationary characteristics as in our case, the initial segments might not represent the later segments well. In such cases, a model trained on early data may underperform when tested on later segments.

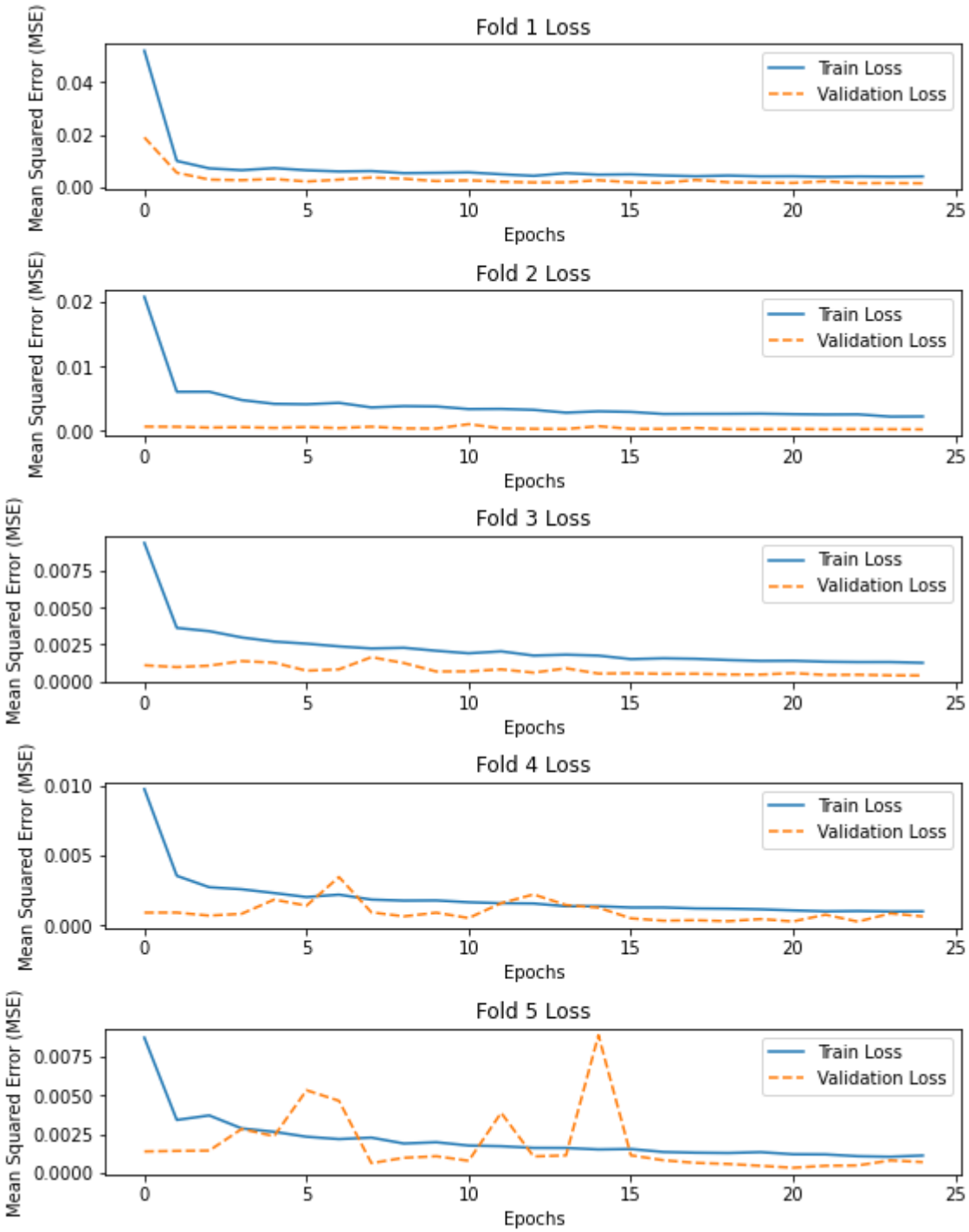
The second fold (MSE: 0.1299) shows a significant drop in error. This indicates that the model performs much better on this segment. The error starts to increase gradually from the third fold onwards. The increasing error trend could be due to overfitting: the model is trained cumulatively on previous folds, so it might start memorizing the training data, thus reducing its ability to generalize on new data. Another reason could be a data behavior shift: the later parts of the data might contain different or more complex patterns than the earlier parts which the model has not seen before, increasing prediction errors.

The high test set MSE relative to the cross-validation MSEs might raise concerns about the model's generalization capabilities. This could be due to the test set containing different, more complex patterns, or it could be a sign that the model might be overfitting to the training data, hence performing poorly on unseen data.

Comparing the plots of the MSE on the train and validation sets for the different folds as a function of the epoch may shed light on this issue:

In [60]:
```
def plot_history(history_list):
    """
    Plot the training and validation loss across multiple histories.

    Args:
    - history_list (list): List of training histories.
    """
    plt.figure(figsize=(15, 10))

    # Iterate over each history item
    for index, history in enumerate(history_list, 1):
        plt.subplot(len(history_list), 2, 2*index - 1)
        plt.plot(history.history['loss'], label='Train Loss')
        plt.plot(history.history['val_loss'], label='Validation Loss', linestyle='-
        plt.title(f'Fold {index} Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Mean Squared Error (MSE)')
        plt.legend()
```

```
    plt.tight_layout()
    plt.show()

# Call the function
plot_history(history_list)
```



The higher value of MSE on the first fold along with the higher values of training and validation losses compared to the other folds suggests underfitting in this early step, at least in comparison with the later steps.

Train and validation losses decrease subsequently until the fourth fold, where the validation loss curve has for the first time segments above the train loss curve. In the last fold train and validation losses have increased to the level of the third fold, and the validation loss curve has a few peaks above the train loss curve, with one value significantly higher. This succession suggests overfitting: the model is getting better at predicting the training data but worse at generalizing to new, unseen data.

As a conclusion of our cross-validation analysis we can expect our model to show slight overfitting. In order to take care of this issue, an additional step of further tuning the architecture and hyperparameters of the model would be needed, which we are not taking here.

Addressing overfitting

A modification of our previous LSTM model that reduces overfitting is introduced here. We implement L2 regularization for each layer and reduce the slide window (the number of past data points we use to create the sequences that are fed to the model) from 60 to 20.

In [61]:
```
# Create sequences from train and test data
X_train, y_train = create_sequences(train_scaled['Close'], seq_length=20)
X_test, y_test = create_sequences(test_scaled['Close'], seq_length=20)

# Reshape the inputs to be suitable for LSTM
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))
```

In [62]:
```
from keras.regularizers import l2

class LSTMModel2:
    def __init__(self, input_shape, l2_reg_strength=0.05):
        self.input_shape = input_shape
        self.l2_reg_strength = l2_reg_strength

        # Call the '_build_model' method to construct the actual LSTM and store it
        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()

        # Using instance variable for regularization strength
        model.add(LSTM(units=50, return_sequences=True, input_shape=self.input_shap
                       kernel_regularizer=l2(self.l2_reg_strength),
                       recurrent_regularizer=l2(self.l2_reg_strength)))
        model.add(Dropout(0.2))

        model.add(LSTM(units=50,
                       kernel_regularizer=l2(self.l2_reg_strength),
                       recurrent_regularizer=l2(self.l2_reg_strength)))
        model.add(Dropout(0.2))
```

```
        model.add(Dense(1))
        return model
```

This second LSTM model architecture now looks as follows:

In [63]:
```
# Print out a summary of the model
lstm_model = LSTMModel2((None, 1),l2_reg_strength=0.01)
lstm_model.model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_18 (LSTM)              (None, None, 50)          10400

 dropout_18 (Dropout)        (None, None, 50)          0

 lstm_19 (LSTM)              (None, 50)                20200

 dropout_19 (Dropout)        (None, 50)                0

 dense_6 (Dense)             (None, 1)                 51

=================================================================
Total params: 30651 (119.73 KB)
Trainable params: 30651 (119.73 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Compiling and training the LSTMModel2

In [64]:
```
lstm_model.model.compile(optimizer='adam', loss='mean_squared_error')
```

In [65]:
```
checkpointer = ModelCheckpoint(
    filepath = 'weights_best2.hdf5',
    verbose = 2,
    save_best_only = True
)
```

In [66]:
```
lstm_model.model.fit(
    X_train,
    y_train,
    epochs=25,
    batch_size = 32,
    callbacks = [checkpointer]
)
```

```
Epoch 1/25
167/168 [============================>.] - ETA: 0s - loss: 0.5497WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 9s 30ms/step - loss: 0.5484
Epoch 2/25
167/168 [============================>.] - ETA: 0s - loss: 0.0276WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 5s 27ms/step - loss: 0.0275
Epoch 3/25
166/168 [============================>.] - ETA: 0s - loss: 0.0107WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 23ms/step - loss: 0.0107
Epoch 4/25
167/168 [============================>.] - ETA: 0s - loss: 0.0092WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 22ms/step - loss: 0.0092
Epoch 5/25
166/168 [============================>.] - ETA: 0s - loss: 0.0080WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 22ms/step - loss: 0.0080
Epoch 6/25
166/168 [============================>.] - ETA: 0s - loss: 0.0073WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 23ms/step - loss: 0.0073
Epoch 7/25
167/168 [============================>.] - ETA: 0s - loss: 0.0064WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 23ms/step - loss: 0.0064
Epoch 8/25
166/168 [============================>.] - ETA: 0s - loss: 0.0058WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 23ms/step - loss: 0.0058
Epoch 9/25
166/168 [============================>.] - ETA: 0s - loss: 0.0055WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 22ms/step - loss: 0.0055
Epoch 10/25
167/168 [============================>.] - ETA: 0s - loss: 0.0050WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 23ms/step - loss: 0.0050
Epoch 11/25
167/168 [============================>.] - ETA: 0s - loss: 0.0047WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 23ms/step - loss: 0.0048
Epoch 12/25
168/168 [==============================] - ETA: 0s - loss: 0.0046WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 22ms/step - loss: 0.0046
Epoch 13/25
168/168 [==============================] - ETA: 0s - loss: 0.0044WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 23ms/step - loss: 0.0044
Epoch 14/25
168/168 [==============================] - ETA: 0s - loss: 0.0045WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 23ms/step - loss: 0.0045
```

```
Epoch 15/25
166/168 [============================>.] - ETA: 0s - loss: 0.0042WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 23ms/step - loss: 0.0042
Epoch 16/25
166/168 [============================>.] - ETA: 0s - loss: 0.0039WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 23ms/step - loss: 0.0039
Epoch 17/25
167/168 [============================>.] - ETA: 0s - loss: 0.0040WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 25ms/step - loss: 0.0040
Epoch 18/25
168/168 [==============================] - ETA: 0s - loss: 0.0039WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 24ms/step - loss: 0.0039
Epoch 19/25
168/168 [==============================] - ETA: 0s - loss: 0.0039WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 24ms/step - loss: 0.0039
Epoch 20/25
166/168 [============================>.] - ETA: 0s - loss: 0.0037WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 4s 25ms/step - loss: 0.0037
Epoch 21/25
166/168 [============================>.] - ETA: 0s - loss: 0.0038WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 5s 27ms/step - loss: 0.0038
Epoch 22/25
167/168 [============================>.] - ETA: 0s - loss: 0.0037WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 6s 37ms/step - loss: 0.0037
Epoch 23/25
166/168 [============================>.] - ETA: 0s - loss: 0.0035WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 5s 31ms/step - loss: 0.0035
Epoch 24/25
168/168 [==============================] - ETA: 0s - loss: 0.0035WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 5s 30ms/step - loss: 0.0035
Epoch 25/25
167/168 [============================>.] - ETA: 0s - loss: 0.0037WARNING:tensorflow:
Can save best model only with val_loss available, skipping.
168/168 [==============================] - 5s 30ms/step - loss: 0.0037
```

Out[66]: <keras.src.callbacks.History at 0x7fc97b65df70>

In [67]:
```
# Compute the predicted values on the test set
test_predict = lstm_model.model.predict(X_test)
# Undo the scaling transformation on the predicted values 'test_predict' and the ac
test_predict = scaler.inverse_transform(test_predict)
```

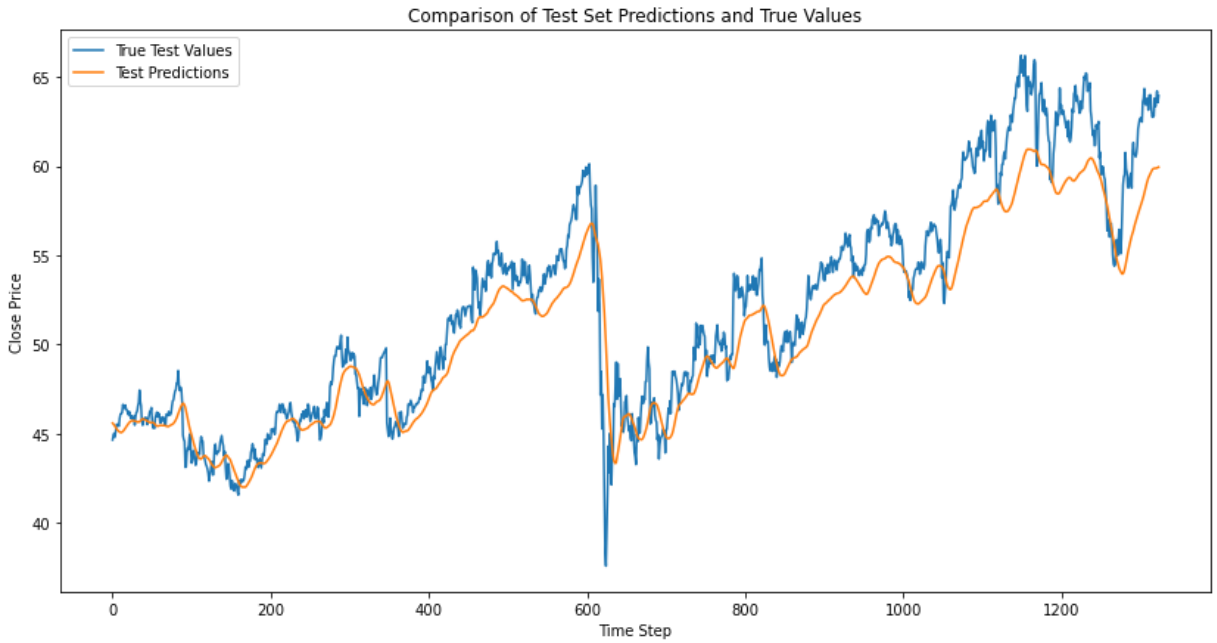
```
42/42 [==============================] - 2s 12ms/step
```

In [68]:
```
# Reshape y_test to a 2D array structure for input to scaler
y_test_reshaped = y_test.reshape(-1, 1)
```

```
y_test = scaler.inverse_transform(y_test_reshaped)
```

Let's plot our prediction versus actual values on the test set for our second LSTM model:

In [69]:
```
# Plot test predictions and actual values
plt.figure(figsize=(14,7))
plt.plot(y_test, label='True Test Values')
plt.plot(test_predict, label='Test Predictions')
plt.title('Comparison of Test Set Predictions and True Values')
plt.xlabel('Time Step')
plt.ylabel('Close Price')
plt.legend()
plt.show()
```



In [70]:
```
mse = mean_squared_error(y_test, test_predict)
print(mse)
```

```
5.595769797308592
```

We note that overall trends are captured but the MSE has increased. This is typically the price to pay for overfitting avoidance.

A cross-validation analysis will determine whether we have correctly taken care of overfitting.

In [71]:
```
history_list, mse_values = train_lstm_cv(train['Close'], 20, LSTMModel2)
```

```
Epoch 1/25
26/28 [==========================>...] - ETA: 0s - loss: 7.1890
Epoch 1: val_loss improved from inf to 5.03674, saving model to weights_best.hdf5
/home/sergio/anaconda3/lib/python3.9/site-packages/keras/src/engine/training.py:3000: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
28/28 [==============================] - 11s 106ms/step - loss: 7.0865 - val_loss:
5.0367
Epoch 2/25
28/28 [==============================] - ETA: 0s - loss: 3.8048
Epoch 2: val_loss improved from 5.03674 to 2.68551, saving model to weights_best.hdf
5
28/28 [==============================] - 1s 34ms/step - loss: 3.8048 - val_loss: 2.6
855
Epoch 3/25
28/28 [==============================] - ETA: 0s - loss: 2.0152
Epoch 3: val_loss improved from 2.68551 to 1.42417, saving model to weights_best.hdf
5
28/28 [==============================] - 1s 37ms/step - loss: 2.0152 - val_loss: 1.4
242
Epoch 4/25
26/28 [==========================>...] - ETA: 0s - loss: 1.0765
Epoch 4: val_loss improved from 1.42417 to 0.75531, saving model to weights_best.hdf
5
28/28 [==============================] - 1s 34ms/step - loss: 1.0605 - val_loss: 0.7
553
Epoch 5/25
27/28 [===========================>..] - ETA: 0s - loss: 0.5609
Epoch 5: val_loss improved from 0.75531 to 0.40854, saving model to weights_best.hdf
5
28/28 [==============================] - 1s 36ms/step - loss: 0.5583 - val_loss: 0.4
085
Epoch 6/25
28/28 [==============================] - ETA: 0s - loss: 0.2971
Epoch 6: val_loss improved from 0.40854 to 0.21972, saving model to weights_best.hdf
5
28/28 [==============================] - 1s 36ms/step - loss: 0.2971 - val_loss: 0.2
197
Epoch 7/25
28/28 [==============================] - ETA: 0s - loss: 0.1654
Epoch 7: val_loss improved from 0.21972 to 0.13167, saving model to weights_best.hdf
5
28/28 [==============================] - 1s 35ms/step - loss: 0.1654 - val_loss: 0.1
317
Epoch 8/25
26/28 [==========================>...] - ETA: 0s - loss: 0.0996
Epoch 8: val_loss improved from 0.13167 to 0.09663, saving model to weights_best.hdf
5
28/28 [==============================] - 1s 35ms/step - loss: 0.0987 - val_loss: 0.0
966
Epoch 9/25
28/28 [==============================] - ETA: 0s - loss: 0.0662
Epoch 9: val_loss improved from 0.09663 to 0.06815, saving model to weights_best.hdf
5
28/28 [==============================] - 1s 35ms/step - loss: 0.0662 - val_loss: 0.0
682
Epoch 10/25
28/28 [==============================] - ETA: 0s - loss: 0.0490
Epoch 10: val_loss improved from 0.06815 to 0.04927, saving model to weights_best.hd
f5
28/28 [==============================] - 1s 34ms/step - loss: 0.0490 - val_loss: 0.0
493
```

```
Epoch 11/25
26/28 [==========================>...] - ETA: 0s - loss: 0.0398
Epoch 11: val_loss improved from 0.04927 to 0.04332, saving model to weights_best.hd
f5
28/28 [==============================] - 1s 34ms/step - loss: 0.0394 - val_loss: 0.0
433
Epoch 12/25
27/28 [===========================>..] - ETA: 0s - loss: 0.0337
Epoch 12: val_loss improved from 0.04332 to 0.03302, saving model to weights_best.hd
f5
28/28 [==============================] - 1s 36ms/step - loss: 0.0337 - val_loss: 0.0
330
Epoch 13/25
27/28 [===========================>..] - ETA: 0s - loss: 0.0310
Epoch 13: val_loss did not improve from 0.03302
28/28 [==============================] - 1s 32ms/step - loss: 0.0310 - val_loss: 0.0
407
Epoch 14/25
27/28 [===========================>..] - ETA: 0s - loss: 0.0288
Epoch 14: val_loss improved from 0.03302 to 0.02841, saving model to weights_best.hd
f5
28/28 [==============================] - 1s 34ms/step - loss: 0.0288 - val_loss: 0.0
284
Epoch 15/25
28/28 [==============================] - ETA: 0s - loss: 0.0268
Epoch 15: val_loss improved from 0.02841 to 0.02509, saving model to weights_best.hd
f5
28/28 [==============================] - 1s 35ms/step - loss: 0.0268 - val_loss: 0.0
251
Epoch 16/25
28/28 [==============================] - ETA: 0s - loss: 0.0247
Epoch 16: val_loss improved from 0.02509 to 0.02204, saving model to weights_best.hd
f5
28/28 [==============================] - 1s 37ms/step - loss: 0.0247 - val_loss: 0.0
220
Epoch 17/25
27/28 [===========================>..] - ETA: 0s - loss: 0.0226
Epoch 17: val_loss improved from 0.02204 to 0.02099, saving model to weights_best.hd
f5
28/28 [==============================] - 1s 35ms/step - loss: 0.0226 - val_loss: 0.0
210
Epoch 18/25
26/28 [==========================>...] - ETA: 0s - loss: 0.0212
Epoch 18: val_loss improved from 0.02099 to 0.01859, saving model to weights_best.hd
f5
28/28 [==============================] - 1s 34ms/step - loss: 0.0210 - val_loss: 0.0
186
Epoch 19/25
27/28 [===========================>..] - ETA: 0s - loss: 0.0194
Epoch 19: val_loss did not improve from 0.01859
28/28 [==============================] - 1s 33ms/step - loss: 0.0195 - val_loss: 0.0
209
Epoch 20/25
28/28 [==============================] - ETA: 0s - loss: 0.0186
Epoch 20: val_loss improved from 0.01859 to 0.01532, saving model to weights_best.hd
f5
```

```
28/28 [==============================] - 1s 36ms/step - loss: 0.0186 - val_loss: 0.0
153
Epoch 21/25
27/28 [===========================>..] - ETA: 0s - loss: 0.0168
Epoch 21: val_loss did not improve from 0.01532
28/28 [==============================] - 1s 36ms/step - loss: 0.0168 - val_loss: 0.0
166
Epoch 22/25
28/28 [==============================] - ETA: 0s - loss: 0.0167
Epoch 22: val_loss improved from 0.01532 to 0.01449, saving model to weights_best.hd
f5
28/28 [==============================] - 1s 34ms/step - loss: 0.0167 - val_loss: 0.0
145
Epoch 23/25
26/28 [==========================>...] - ETA: 0s - loss: 0.0151
Epoch 23: val_loss improved from 0.01449 to 0.01198, saving model to weights_best.hd
f5
28/28 [==============================] - 1s 35ms/step - loss: 0.0150 - val_loss: 0.0
120
Epoch 24/25
26/28 [==========================>...] - ETA: 0s - loss: 0.0156
Epoch 24: val_loss did not improve from 0.01198
28/28 [==============================] - 1s 33ms/step - loss: 0.0153 - val_loss: 0.0
161
Epoch 25/25
26/28 [==========================>...] - ETA: 0s - loss: 0.0144
Epoch 25: val_loss did not improve from 0.01198
28/28 [==============================] - 1s 36ms/step - loss: 0.0144 - val_loss: 0.0
144
28/28 [==============================] - 1s 11ms/step
Epoch 1/25
54/56 [===========================>..] - ETA: 0s - loss: 5.3966
Epoch 1: val_loss improved from inf to 2.59930, saving model to weights_best.hdf5
56/56 [==============================] - 8s 52ms/step - loss: 5.3266 - val_loss: 2.5
993
Epoch 2/25
 5/56 [=>............................] - ETA: 1s - loss: 2.4767
/home/sergio/anaconda3/lib/python3.9/site-packages/keras/src/engine/training.py:300
0: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This f
ile format is considered legacy. We recommend using instead the native Keras format,
e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
56/56 [==============================] - ETA: 0s - loss: 1.4838
Epoch 2: val_loss improved from 2.59930 to 0.71746, saving model to weights_best.hdf
5
56/56 [==============================] - 2s 29ms/step - loss: 1.4838 - val_loss: 0.7
175
Epoch 3/25
55/56 [============================>.] - ETA: 0s - loss: 0.4143
Epoch 3: val_loss improved from 0.71746 to 0.22398, saving model to weights_best.hdf
5
56/56 [==============================] - 2s 29ms/step - loss: 0.4126 - val_loss: 0.2
240
Epoch 4/25
55/56 [============================>.] - ETA: 0s - loss: 0.1283
Epoch 4: val_loss improved from 0.22398 to 0.07348, saving model to weights_best.hdf
5
56/56 [==============================] - 2s 32ms/step - loss: 0.1278 - val_loss: 0.0
735
Epoch 5/25
55/56 [============================>.] - ETA: 0s - loss: 0.0549
Epoch 5: val_loss improved from 0.07348 to 0.03834, saving model to weights_best.hdf
5
56/56 [==============================] - 2s 31ms/step - loss: 0.0548 - val_loss: 0.0
383
Epoch 6/25
55/56 [============================>.] - ETA: 0s - loss: 0.0326
Epoch 6: val_loss improved from 0.03834 to 0.02183, saving model to weights_best.hdf
5
56/56 [==============================] - 2s 34ms/step - loss: 0.0326 - val_loss: 0.0
218
Epoch 7/25
55/56 [============================>.] - ETA: 0s - loss: 0.0238
Epoch 7: val_loss improved from 0.02183 to 0.01632, saving model to weights_best.hdf
5
56/56 [==============================] - 2s 33ms/step - loss: 0.0238 - val_loss: 0.0
163
Epoch 8/25
55/56 [============================>.] - ETA: 0s - loss: 0.0187
Epoch 8: val_loss did not improve from 0.01632
56/56 [==============================] - 2s 30ms/step - loss: 0.0187 - val_loss: 0.0
171
Epoch 9/25
54/56 [===========================>..] - ETA: 0s - loss: 0.0159
Epoch 9: val_loss improved from 0.01632 to 0.01116, saving model to weights_best.hdf
5
56/56 [==============================] - 2s 29ms/step - loss: 0.0159 - val_loss: 0.0
112
Epoch 10/25
56/56 [==============================] - ETA: 0s - loss: 0.0141
Epoch 10: val_loss improved from 0.01116 to 0.00991, saving model to weights_best.hd
f5
56/56 [==============================] - 2s 31ms/step - loss: 0.0141 - val_loss: 0.0
099
Epoch 11/25
54/56 [===========================>..] - ETA: 0s - loss: 0.0133
Epoch 11: val_loss improved from 0.00991 to 0.00859, saving model to weights_best.hd
f5
```

```
56/56 [==============================] - 2s 30ms/step - loss: 0.0133 - val_loss: 0.0
086
Epoch 12/25
54/56 [===========================>..] - ETA: 0s - loss: 0.0124
Epoch 12: val_loss improved from 0.00859 to 0.00765, saving model to weights_best.hd
f5
56/56 [==============================] - 2s 32ms/step - loss: 0.0124 - val_loss: 0.0
077
Epoch 13/25
54/56 [===========================>..] - ETA: 0s - loss: 0.0124
Epoch 13: val_loss did not improve from 0.00765
56/56 [==============================] - 2s 33ms/step - loss: 0.0123 - val_loss: 0.0
109
Epoch 14/25
55/56 [============================>.] - ETA: 0s - loss: 0.0115
Epoch 14: val_loss improved from 0.00765 to 0.00666, saving model to weights_best.hd
f5
56/56 [==============================] - 2s 33ms/step - loss: 0.0115 - val_loss: 0.0
067
Epoch 15/25
55/56 [============================>.] - ETA: 0s - loss: 0.0111
Epoch 15: val_loss did not improve from 0.00666
56/56 [==============================] - 2s 29ms/step - loss: 0.0111 - val_loss: 0.0
072
Epoch 16/25
56/56 [==============================] - ETA: 0s - loss: 0.0111
Epoch 16: val_loss did not improve from 0.00666
56/56 [==============================] - 2s 29ms/step - loss: 0.0111 - val_loss: 0.0
069
Epoch 17/25
55/56 [============================>.] - ETA: 0s - loss: 0.0108
Epoch 17: val_loss did not improve from 0.00666
56/56 [==============================] - 2s 29ms/step - loss: 0.0108 - val_loss: 0.0
105
Epoch 18/25
54/56 [===========================>..] - ETA: 0s - loss: 0.0109
Epoch 18: val_loss improved from 0.00666 to 0.00532, saving model to weights_best.hd
f5
56/56 [==============================] - 2s 29ms/step - loss: 0.0109 - val_loss: 0.0
053
Epoch 19/25
55/56 [============================>.] - ETA: 0s - loss: 0.0102
Epoch 19: val_loss did not improve from 0.00532
56/56 [==============================] - 2s 29ms/step - loss: 0.0102 - val_loss: 0.0
060
Epoch 20/25
54/56 [===========================>..] - ETA: 0s - loss: 0.0100
Epoch 20: val_loss improved from 0.00532 to 0.00523, saving model to weights_best.hd
f5
56/56 [==============================] - 2s 29ms/step - loss: 0.0100 - val_loss: 0.0
052
Epoch 21/25
56/56 [==============================] - ETA: 0s - loss: 0.0100
Epoch 21: val_loss improved from 0.00523 to 0.00503, saving model to weights_best.hd
f5
56/56 [==============================] - 2s 29ms/step - loss: 0.0100 - val_loss: 0.0
```

```
050
Epoch 22/25
55/56 [============================>.] - ETA: 0s - loss: 0.0094
Epoch 22: val_loss did not improve from 0.00503
56/56 [==============================] - 2s 28ms/step - loss: 0.0094 - val_loss: 0.0
054
Epoch 23/25
54/56 [============================>..] - ETA: 0s - loss: 0.0094
Epoch 23: val_loss improved from 0.00503 to 0.00482, saving model to weights_best.hd
f5
56/56 [==============================] - 2s 29ms/step - loss: 0.0094 - val_loss: 0.0
048
Epoch 24/25
56/56 [==============================] - ETA: 0s - loss: 0.0092
Epoch 24: val_loss did not improve from 0.00482
56/56 [==============================] - 2s 28ms/step - loss: 0.0092 - val_loss: 0.0
049
Epoch 25/25
56/56 [==============================] - ETA: 0s - loss: 0.0091
Epoch 25: val_loss improved from 0.00482 to 0.00470, saving model to weights_best.hd
f5
56/56 [==============================] - 2s 34ms/step - loss: 0.0091 - val_loss: 0.0
047
28/28 [==============================] - 1s 9ms/step
Epoch 1/25
83/84 [============================>.] - ETA: 0s - loss: 4.2261
Epoch 1: val_loss improved from inf to 1.35483, saving model to weights_best.hdf5
84/84 [==============================] - 7s 38ms/step - loss: 4.2112 - val_loss: 1.3
548
Epoch 2/25
 4/84 [>.............................] - ETA: 1s - loss: 1.3220
/home/sergio/anaconda3/lib/python3.9/site-packages/keras/src/engine/training.py:300
0: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This f
ile format is considered legacy. We recommend using instead the native Keras format,
e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
84/84 [==============================] - ETA: 0s - loss: 0.6171
Epoch 2: val_loss improved from 1.35483 to 0.19652, saving model to weights_best.hdf
5
84/84 [==============================] - 3s 31ms/step - loss: 0.6171 - val_loss: 0.1
965
Epoch 3/25
82/84 [============================>.] - ETA: 0s - loss: 0.1042
Epoch 3: val_loss improved from 0.19652 to 0.04297, saving model to weights_best.hdf
5
84/84 [==============================] - 3s 30ms/step - loss: 0.1032 - val_loss: 0.0
430
Epoch 4/25
83/84 [============================>.] - ETA: 0s - loss: 0.0335
Epoch 4: val_loss improved from 0.04297 to 0.02056, saving model to weights_best.hdf
5
84/84 [==============================] - 2s 28ms/step - loss: 0.0335 - val_loss: 0.0
206
Epoch 5/25
83/84 [============================>.] - ETA: 0s - loss: 0.0209
Epoch 5: val_loss improved from 0.02056 to 0.01491, saving model to weights_best.hdf
5
84/84 [==============================] - 2s 27ms/step - loss: 0.0209 - val_loss: 0.0
149
Epoch 6/25
82/84 [============================>.] - ETA: 0s - loss: 0.0157
Epoch 6: val_loss improved from 0.01491 to 0.01116, saving model to weights_best.hdf
5
84/84 [==============================] - 2s 29ms/step - loss: 0.0157 - val_loss: 0.0
112
Epoch 7/25
84/84 [==============================] - ETA: 0s - loss: 0.0130
Epoch 7: val_loss improved from 0.01116 to 0.00926, saving model to weights_best.hdf
5
84/84 [==============================] - 2s 27ms/step - loss: 0.0130 - val_loss: 0.0
093
Epoch 8/25
83/84 [============================>.] - ETA: 0s - loss: 0.0111
Epoch 8: val_loss improved from 0.00926 to 0.00780, saving model to weights_best.hdf
5
84/84 [==============================] - 3s 30ms/step - loss: 0.0112 - val_loss: 0.0
078
Epoch 9/25
83/84 [============================>.] - ETA: 0s - loss: 0.0105
Epoch 9: val_loss improved from 0.00780 to 0.00773, saving model to weights_best.hdf
5
84/84 [==============================] - 2s 28ms/step - loss: 0.0105 - val_loss: 0.0
077
Epoch 10/25
84/84 [==============================] - ETA: 0s - loss: 0.0095
Epoch 10: val_loss improved from 0.00773 to 0.00674, saving model to weights_best.hd
f5
84/84 [==============================] - 3s 30ms/step - loss: 0.0095 - val_loss: 0.0
067
Epoch 11/25
82/84 [============================>.] - ETA: 0s - loss: 0.0091
Epoch 11: val_loss improved from 0.00674 to 0.00610, saving model to weights_best.hd
```

```
f5
84/84 [==============================] - 2s 30ms/step - loss: 0.0091 - val_loss: 0.0
061
Epoch 12/25
82/84 [============================>.] - ETA: 0s - loss: 0.0087
Epoch 12: val_loss improved from 0.00610 to 0.00585, saving model to weights_best.hd
f5
84/84 [==============================] - 3s 30ms/step - loss: 0.0086 - val_loss: 0.0
058
Epoch 13/25
82/84 [============================>.] - ETA: 0s - loss: 0.0085
Epoch 13: val_loss improved from 0.00585 to 0.00558, saving model to weights_best.hd
f5
84/84 [==============================] - 2s 27ms/step - loss: 0.0085 - val_loss: 0.0
056
Epoch 14/25
82/84 [============================>.] - ETA: 0s - loss: 0.0084
Epoch 14: val_loss improved from 0.00558 to 0.00512, saving model to weights_best.hd
f5
84/84 [==============================] - 2s 28ms/step - loss: 0.0084 - val_loss: 0.0
051
Epoch 15/25
83/84 [============================>.] - ETA: 0s - loss: 0.0082
Epoch 15: val_loss did not improve from 0.00512
84/84 [==============================] - 2s 29ms/step - loss: 0.0082 - val_loss: 0.0
054
Epoch 16/25
84/84 [==============================] - ETA: 0s - loss: 0.0078
Epoch 16: val_loss improved from 0.00512 to 0.00483, saving model to weights_best.hd
f5
84/84 [==============================] - 2s 29ms/step - loss: 0.0078 - val_loss: 0.0
048
Epoch 17/25
82/84 [============================>.] - ETA: 0s - loss: 0.0077
Epoch 17: val_loss did not improve from 0.00483
84/84 [==============================] - 2s 28ms/step - loss: 0.0077 - val_loss: 0.0
052
Epoch 18/25
83/84 [============================>.] - ETA: 0s - loss: 0.0075
Epoch 18: val_loss improved from 0.00483 to 0.00449, saving model to weights_best.hd
f5
84/84 [==============================] - 2s 29ms/step - loss: 0.0074 - val_loss: 0.0
045
Epoch 19/25
84/84 [==============================] - ETA: 0s - loss: 0.0070
Epoch 19: val_loss improved from 0.00449 to 0.00425, saving model to weights_best.hd
f5
84/84 [==============================] - 2s 29ms/step - loss: 0.0070 - val_loss: 0.0
043
Epoch 20/25
83/84 [============================>.] - ETA: 0s - loss: 0.0068
Epoch 20: val_loss improved from 0.00425 to 0.00409, saving model to weights_best.hd
f5
84/84 [==============================] - 2s 27ms/step - loss: 0.0068 - val_loss: 0.0
041
Epoch 21/25
```

```
82/84 [============================>.] - ETA: 0s - loss: 0.0068
Epoch 21: val_loss improved from 0.00409 to 0.00394, saving model to weights_best.hd
f5
84/84 [==============================] - 2s 27ms/step - loss: 0.0068 - val_loss: 0.0
039
Epoch 22/25
83/84 [============================>.] - ETA: 0s - loss: 0.0067
Epoch 22: val_loss improved from 0.00394 to 0.00392, saving model to weights_best.hd
f5
84/84 [==============================] - 2s 27ms/step - loss: 0.0067 - val_loss: 0.0
039
Epoch 23/25
83/84 [============================>.] - ETA: 0s - loss: 0.0067
Epoch 23: val_loss improved from 0.00392 to 0.00392, saving model to weights_best.hd
f5
84/84 [==============================] - 3s 31ms/step - loss: 0.0067 - val_loss: 0.0
039
Epoch 24/25
83/84 [============================>.] - ETA: 0s - loss: 0.0066
Epoch 24: val_loss did not improve from 0.00392
84/84 [==============================] - 2s 29ms/step - loss: 0.0066 - val_loss: 0.0
049
Epoch 25/25
82/84 [============================>.] - ETA: 0s - loss: 0.0065
Epoch 25: val_loss improved from 0.00392 to 0.00368, saving model to weights_best.hd
f5
84/84 [==============================] - 3s 31ms/step - loss: 0.0064 - val_loss: 0.0
037
28/28 [==============================] - 2s 11ms/step
Epoch 1/25
111/112 [============================>.] - ETA: 0s - loss: 3.4417
Epoch 1: val_loss improved from inf to 0.79598, saving model to weights_best.hdf5
/home/sergio/anaconda3/lib/python3.9/site-packages/keras/src/engine/training.py:300
0: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This f
ile format is considered legacy. We recommend using instead the native Keras format,
e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
112/112 [==============================] - 13s 55ms/step - loss: 3.4311 - val_loss:
0.7960
Epoch 2/25
110/112 [============================>.] - ETA: 0s - loss: 0.2741
Epoch 2: val_loss improved from 0.79598 to 0.12092, saving model to weights_best.hdf
5
112/112 [==============================] - 4s 31ms/step - loss: 0.2717 - val_loss:
0.1209
Epoch 3/25
110/112 [============================>.] - ETA: 0s - loss: 0.0431
Epoch 3: val_loss improved from 0.12092 to 0.04445, saving model to weights_best.hdf
5
112/112 [==============================] - 3s 28ms/step - loss: 0.0428 - val_loss:
0.0444
Epoch 4/25
110/112 [============================>.] - ETA: 0s - loss: 0.0189
Epoch 4: val_loss improved from 0.04445 to 0.02194, saving model to weights_best.hdf
5
112/112 [==============================] - 3s 28ms/step - loss: 0.0188 - val_loss:
0.0219
Epoch 5/25
111/112 [============================>.] - ETA: 0s - loss: 0.0129
Epoch 5: val_loss improved from 0.02194 to 0.01521, saving model to weights_best.hdf
5
112/112 [==============================] - 3s 29ms/step - loss: 0.0129 - val_loss:
0.0152
Epoch 6/25
110/112 [============================>.] - ETA: 0s - loss: 0.0109
Epoch 6: val_loss improved from 0.01521 to 0.01414, saving model to weights_best.hdf
5
112/112 [==============================] - 3s 26ms/step - loss: 0.0109 - val_loss:
0.0141
Epoch 7/25
111/112 [============================>.] - ETA: 0s - loss: 0.0097
Epoch 7: val_loss did not improve from 0.01414
112/112 [==============================] - 3s 29ms/step - loss: 0.0097 - val_loss:
0.0159
Epoch 8/25
111/112 [============================>.] - ETA: 0s - loss: 0.0091
Epoch 8: val_loss improved from 0.01414 to 0.00709, saving model to weights_best.hdf
5
112/112 [==============================] - 4s 33ms/step - loss: 0.0091 - val_loss:
0.0071
Epoch 9/25
111/112 [============================>.] - ETA: 0s - loss: 0.0083
Epoch 9: val_loss did not improve from 0.00709
112/112 [==============================] - 4s 32ms/step - loss: 0.0083 - val_loss:
0.0106
Epoch 10/25
111/112 [============================>.] - ETA: 0s - loss: 0.0078
Epoch 10: val_loss improved from 0.00709 to 0.00548, saving model to weights_best.hd
f5
112/112 [==============================] - 4s 33ms/step - loss: 0.0078 - val_loss:
0.0055
Epoch 11/25
111/112 [============================>.] - ETA: 0s - loss: 0.0077
```

```
Epoch 11: val_loss did not improve from 0.00548
112/112 [==============================] - 3s 31ms/step - loss: 0.0077 - val_loss:
0.0055
Epoch 12/25
111/112 [============================>.] - ETA: 0s - loss: 0.0073
Epoch 12: val_loss did not improve from 0.00548
112/112 [==============================] - 4s 32ms/step - loss: 0.0073 - val_loss:
0.0064
Epoch 13/25
111/112 [============================>.] - ETA: 0s - loss: 0.0069
Epoch 13: val_loss improved from 0.00548 to 0.00481, saving model to weights_best.hd
f5
112/112 [==============================] - 4s 34ms/step - loss: 0.0069 - val_loss:
0.0048
Epoch 14/25
111/112 [============================>.] - ETA: 0s - loss: 0.0070
Epoch 14: val_loss did not improve from 0.00481
112/112 [==============================] - 4s 33ms/step - loss: 0.0070 - val_loss:
0.0062
Epoch 15/25
111/112 [============================>.] - ETA: 0s - loss: 0.0065
Epoch 15: val_loss did not improve from 0.00481
112/112 [==============================] - 4s 35ms/step - loss: 0.0065 - val_loss:
0.0114
Epoch 16/25
112/112 [==============================] - ETA: 0s - loss: 0.0065
Epoch 16: val_loss improved from 0.00481 to 0.00389, saving model to weights_best.hd
f5
112/112 [==============================] - 4s 35ms/step - loss: 0.0065 - val_loss:
0.0039
Epoch 17/25
110/112 [============================>.] - ETA: 0s - loss: 0.0062
Epoch 17: val_loss did not improve from 0.00389
112/112 [==============================] - 4s 33ms/step - loss: 0.0062 - val_loss:
0.0135
Epoch 18/25
112/112 [==============================] - ETA: 0s - loss: 0.0062
Epoch 18: val_loss improved from 0.00389 to 0.00357, saving model to weights_best.hd
f5
112/112 [==============================] - 4s 33ms/step - loss: 0.0062 - val_loss:
0.0036
Epoch 19/25
111/112 [============================>.] - ETA: 0s - loss: 0.0061
Epoch 19: val_loss did not improve from 0.00357
112/112 [==============================] - 5s 45ms/step - loss: 0.0061 - val_loss:
0.0050
Epoch 20/25
112/112 [==============================] - ETA: 0s - loss: 0.0061
Epoch 20: val_loss did not improve from 0.00357
112/112 [==============================] - 4s 32ms/step - loss: 0.0061 - val_loss:
0.0047
Epoch 21/25
110/112 [============================>.] - ETA: 0s - loss: 0.0058
Epoch 21: val_loss did not improve from 0.00357
112/112 [==============================] - 4s 33ms/step - loss: 0.0058 - val_loss:
0.0038
```

```
Epoch 22/25
112/112 [==============================] - ETA: 0s - loss: 0.0058
Epoch 22: val_loss did not improve from 0.00357
112/112 [==============================] - 4s 40ms/step - loss: 0.0058 - val_loss:
0.0042
Epoch 23/25
111/112 [============================>.] - ETA: 0s - loss: 0.0056
Epoch 23: val_loss did not improve from 0.00357
112/112 [==============================] - 44s 393ms/step - loss: 0.0056 - val_loss:
0.0039
Epoch 24/25
112/112 [==============================] - ETA: 0s - loss: 0.0055
Epoch 24: val_loss did not improve from 0.00357
112/112 [==============================] - 124s 1s/step - loss: 0.0055 - val_loss:
0.0062
Epoch 25/25
112/112 [==============================] - ETA: 0s - loss: 0.0056
Epoch 25: val_loss improved from 0.00357 to 0.00341, saving model to weights_best.hd
f5
112/112 [==============================] - 8s 66ms/step - loss: 0.0056 - val_loss:
0.0034
28/28 [==============================] - 16s 15ms/step
Epoch 1/25
139/140 [============================>.] - ETA: 3s - loss: 2.8530
Epoch 1: val_loss improved from inf to 0.40674, saving model to weights_best.hdf5
140/140 [==============================] - 615s 4s/step - loss: 2.8453 - val_loss:
0.4067
Epoch 2/25
  3/140 [..............................] - ETA: 4s - loss: 0.3720
/home/sergio/anaconda3/lib/python3.9/site-packages/keras/src/engine/training.py:300
0: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This f
ile format is considered legacy. We recommend using instead the native Keras format,
e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
139/140 [============================>.] - ETA: 0s - loss: 0.1274
Epoch 2: val_loss improved from 0.40674 to 0.04790, saving model to weights_best.hdf
5
140/140 [==============================] - 5s 38ms/step - loss: 0.1271 - val_loss:
0.0479
Epoch 3/25
139/140 [============================>.] - ETA: 0s - loss: 0.0244
Epoch 3: val_loss improved from 0.04790 to 0.02636, saving model to weights_best.hdf
5
140/140 [==============================] - 6s 40ms/step - loss: 0.0244 - val_loss:
0.0264
Epoch 4/25
139/140 [============================>.] - ETA: 0s - loss: 0.0153
Epoch 4: val_loss did not improve from 0.02636
140/140 [==============================] - 5s 33ms/step - loss: 0.0153 - val_loss:
0.0280
Epoch 5/25
139/140 [============================>.] - ETA: 0s - loss: 0.0119
Epoch 5: val_loss improved from 0.02636 to 0.01149, saving model to weights_best.hdf
5
140/140 [==============================] - 5s 33ms/step - loss: 0.0119 - val_loss:
0.0115
Epoch 6/25
139/140 [============================>.] - ETA: 0s - loss: 0.0097
Epoch 6: val_loss did not improve from 0.01149
140/140 [==============================] - 5s 33ms/step - loss: 0.0097 - val_loss:
0.0193
Epoch 7/25
139/140 [============================>.] - ETA: 0s - loss: 0.0087
Epoch 7: val_loss improved from 0.01149 to 0.00839, saving model to weights_best.hdf
5
140/140 [==============================] - 5s 33ms/step - loss: 0.0087 - val_loss:
0.0084
Epoch 8/25
139/140 [============================>.] - ETA: 0s - loss: 0.0084
Epoch 8: val_loss improved from 0.00839 to 0.00627, saving model to weights_best.hdf
5
140/140 [==============================] - 5s 33ms/step - loss: 0.0084 - val_loss:
0.0063
Epoch 9/25
140/140 [==============================] - ETA: 0s - loss: 0.0081
Epoch 9: val_loss did not improve from 0.00627
140/140 [==============================] - 4s 32ms/step - loss: 0.0081 - val_loss:
0.0110
Epoch 10/25
139/140 [============================>.] - ETA: 0s - loss: 0.0075
Epoch 10: val_loss did not improve from 0.00627
140/140 [==============================] - 5s 32ms/step - loss: 0.0075 - val_loss:
0.0107
Epoch 11/25
140/140 [==============================] - ETA: 0s - loss: 0.0072
Epoch 11: val_loss did not improve from 0.00627
140/140 [==============================] - 4s 32ms/step - loss: 0.0072 - val_loss:
0.0106
Epoch 12/25
140/140 [==============================] - ETA: 0s - loss: 0.0073
```

```
Epoch 12: val_loss did not improve from 0.00627
140/140 [==============================] - 5s 32ms/step - loss: 0.0073 - val_loss:
0.0093
Epoch 13/25
139/140 [============================>.] - ETA: 0s - loss: 0.0067
Epoch 13: val_loss did not improve from 0.00627
140/140 [==============================] - 4s 30ms/step - loss: 0.0067 - val_loss:
0.0064
Epoch 14/25
139/140 [============================>.] - ETA: 0s - loss: 0.0067
Epoch 14: val_loss improved from 0.00627 to 0.00442, saving model to weights_best.hd
f5
140/140 [==============================] - 4s 32ms/step - loss: 0.0067 - val_loss:
0.0044
Epoch 15/25
138/140 [============================>.] - ETA: 0s - loss: 0.0065
Epoch 15: val_loss did not improve from 0.00442
140/140 [==============================] - 4s 29ms/step - loss: 0.0065 - val_loss:
0.0103
Epoch 16/25
139/140 [============================>.] - ETA: 0s - loss: 0.0061
Epoch 16: val_loss did not improve from 0.00442
140/140 [==============================] - 4s 30ms/step - loss: 0.0061 - val_loss:
0.0095
Epoch 17/25
139/140 [============================>.] - ETA: 0s - loss: 0.0062
Epoch 17: val_loss did not improve from 0.00442
140/140 [==============================] - 4s 29ms/step - loss: 0.0062 - val_loss:
0.0208
Epoch 18/25
139/140 [============================>.] - ETA: 0s - loss: 0.0064
Epoch 18: val_loss did not improve from 0.00442
140/140 [==============================] - 4s 29ms/step - loss: 0.0064 - val_loss:
0.0089
Epoch 19/25
139/140 [============================>.] - ETA: 0s - loss: 0.0059
Epoch 19: val_loss did not improve from 0.00442
140/140 [==============================] - 4s 30ms/step - loss: 0.0059 - val_loss:
0.0110
Epoch 20/25
139/140 [============================>.] - ETA: 0s - loss: 0.0060
Epoch 20: val_loss improved from 0.00442 to 0.00438, saving model to weights_best.hd
f5
140/140 [==============================] - 4s 30ms/step - loss: 0.0060 - val_loss:
0.0044
Epoch 21/25
139/140 [============================>.] - ETA: 0s - loss: 0.0057
Epoch 21: val_loss improved from 0.00438 to 0.00420, saving model to weights_best.hd
f5
140/140 [==============================] - 4s 30ms/step - loss: 0.0057 - val_loss:
0.0042
Epoch 22/25
139/140 [============================>.] - ETA: 0s - loss: 0.0058
Epoch 22: val_loss did not improve from 0.00420
140/140 [==============================] - 4s 30ms/step - loss: 0.0058 - val_loss:
0.0067
```

```
Epoch 23/25
139/140 [============================>.] - ETA: 0s - loss: 0.0060
Epoch 23: val_loss did not improve from 0.00420
140/140 [==============================] - 4s 30ms/step - loss: 0.0060 - val_loss:
0.0045
Epoch 24/25
139/140 [============================>.] - ETA: 0s - loss: 0.0054
Epoch 24: val_loss improved from 0.00420 to 0.00335, saving model to weights_best.hd
f5
140/140 [==============================] - 4s 30ms/step - loss: 0.0054 - val_loss:
0.0033
Epoch 25/25
139/140 [============================>.] - ETA: 0s - loss: 0.0057
Epoch 25: val_loss did not improve from 0.00335
140/140 [==============================] - 4s 29ms/step - loss: 0.0057 - val_loss:
0.0084
28/28 [==============================] - 1s 11ms/step
```

The average MSE and standard deviation are:

In [72]:
```
# Compute average MSE and its standard deviation
average_mse = np.mean(mse_values)
mse_std = np.std(mse_values)

print(f"Average MSE across all folds: {average_mse:.4f}")
print(f"MSE standard deviation: {mse_std:.4f}")
```

```
Average MSE across all folds: 2.2745
MSE standard deviation: 1.7985
```

A relatively moderate variability in the performance across the different folds is still present, as the standard deviation of the MSE values is similar to the mean MSE (although less similar then in the previous model). This suggests that our new model is still sensitive to the specific portions of the dataset it is trained on, but this sensitivity has been slightly reduced.

In [73]:
```
mse_values
```

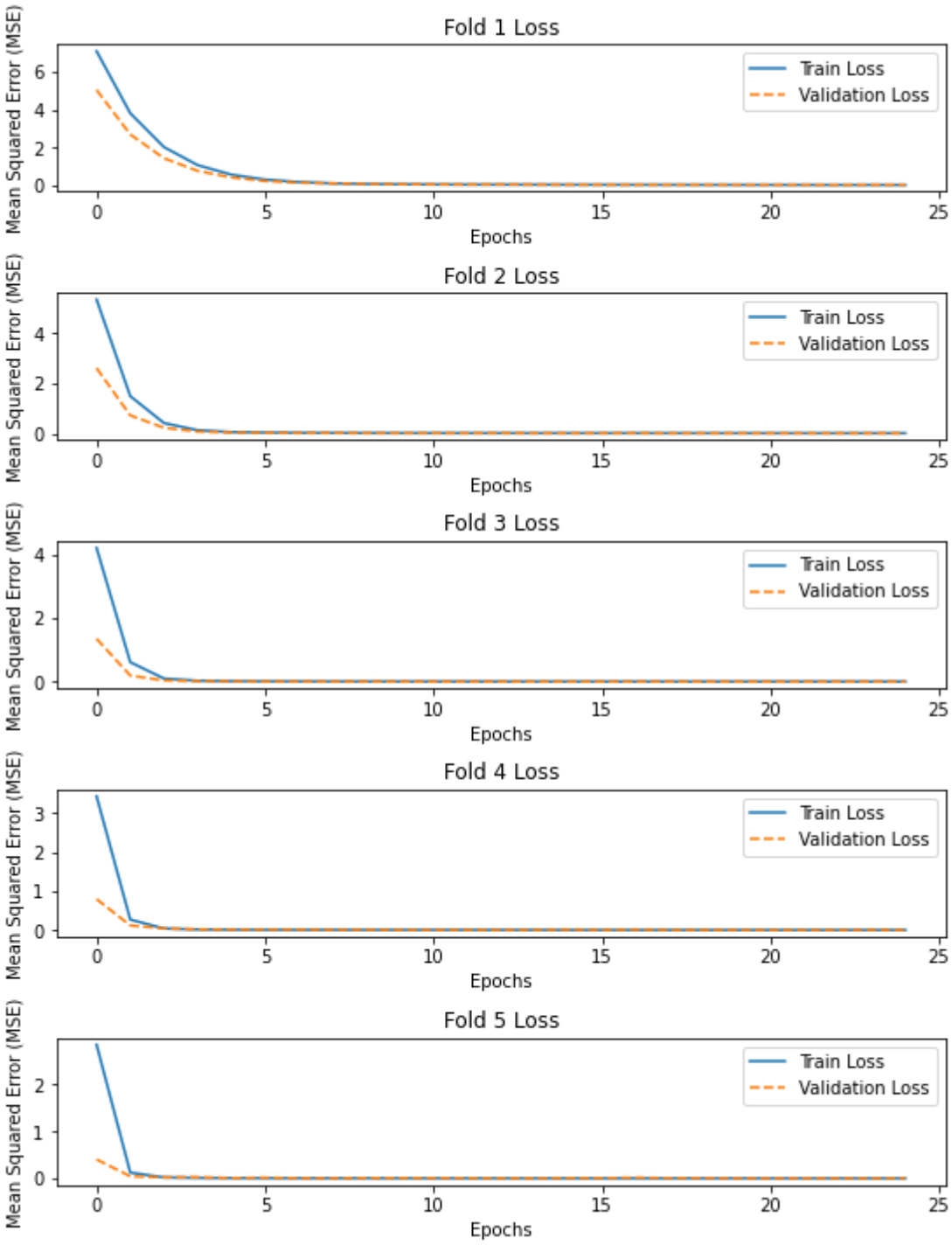
Out[73]:
```
[4.562612010871846,
 0.7350575217430886,
 0.7987386338447188,
 0.8879223753729432,
 4.388073760615863]
```

The individual values of the MSE for different folds seem more similar among themselves, with the largest one (4.56) being around 6 times the smallest one (0.73), which is also a reduction compared to the previous model.

Still, this variability might be due to overfitting. Again, a look at the train loss versus validation loss plots sheds light on the issue.

In [74]:
```
def plot_history(history_list):
    """
    Plot the training and validation loss across multiple histories.
```

```
    Args:
    - history_list (list): List of training histories.
    """
    plt.figure(figsize=(15, 10))

    # Iterate over each history item
    for index, history in enumerate(history_list, 1):
        plt.subplot(len(history_list), 2, 2*index - 1)
        plt.plot(history.history['loss'], label='Train Loss')
        plt.plot(history.history['val_loss'], label='Validation Loss', linestyle='-
        plt.title(f'Fold {index} Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Mean Squared Error (MSE)')
        plt.legend()

        # If you wish to also plot other metrics, for example accuracy (if used), y
        # plt.subplot(len(history_list), 2, 2*index)
        # plt.plot(history.history['accuracy'], label='Train Accuracy')
        # plt.plot(history.history['val_accuracy'], label='Validation Accuracy', li
        # plt.title(f'Fold {index} Accuracy')
        # plt.xlabel('Epochs')
        # plt.ylabel('Accuracy')
        # plt.legend()

    plt.tight_layout()
    plt.show()

# Call the function
plot_history(history_list)
```



The validation loss curve is always under the train loss curve. This demonstrates that our second model addresses overfitting satisfactorily. The variability on the different cross validation folds seems to be due to the time series data showing different behaviors, noise, and patterns that cannot be learned from previous folds. This is particularly likely in the case of a non-stationary time series as ours.

In general, whether to include or not corrections to prevent overfitting will depend on the specific application and time scale we have in mind. For instance, it seems reasonable to reduce overfitting for prediction over long periods of time. However, in the short term we might be more interested in obtaining a prediction that better captures noise.

Multivariate LSTM

We shall now explore the effect of adding extra features to our model. We must keep in mind that introducing additional features to an LSTM model can lead to a variety of outcomes, and the effect on the model's performance is not necessarily predictable. Namely, adding more features does not always guarantee improved performance.

The multivariate case requires some modifications to our previous code. We define a new function for the train/test split and scaling where a subset of features has been selected:

In [75]:
```
def scale_dataframe_m(filename):
    # Load data into DataFrame
    df = pd.read_csv(filename, parse_dates=True, index_col=0)

    df = df[['Close', 'Open', 'High', 'Reported EPS']]

    # Compute the number of data points for 80% of the data
    train_size = int(0.8 * len(df))

    # Split into train and test sets
    train = df[:train_size]
    test = df[train_size:]

    # Scale the entire dataset (all features, including 'Close')
    scaler = MinMaxScaler(feature_range=(0,1))
    train_scaled = pd.DataFrame(scaler.fit_transform(train), columns=train.columns,
    test_scaled = pd.DataFrame(scaler.transform(test), columns=test.columns, index=

    # Return all relevant data
    return df, train, test, train_scaled, test_scaled, scaler

df, train, test, train_scaled, test_scaled, scaler = scale_dataframe_m('df_KO_.csv'
```

The `create_sequences` function needs to be slightly modified so we are cerating sequences of the selected features, not just for `Close`. However, since we are only interested in predicting `Close`, the labels `ys` will only correspond to this variable.

In [76]:
```
def create_sequences_m(data, seq_length):
    xs = []
    ys = []

    for i in range(len(data)-seq_length-1):
        x = data.iloc[i:(i+seq_length)].values
        y = data.iloc[i+seq_length]["Close"]  # Target remains the "Close" value
        xs.append(x)
        ys.append(y)
```

```
    return np.array(xs), np.array(ys)
```

In [77]:
```
# Set the window size back to 60
seq_length = 60
```

In [78]:
```
# Create sequences from train and test data
X_train, y_train = create_sequences_m(train_scaled, seq_length)
X_test, y_test = create_sequences_m(test_scaled, seq_length)
```

Note that in the multivariate case, if the `create_sequences` function is handling multivariate data appropriately, there is no need to explicitly reshape the resulting `X_train` and `X_test` sequences: they should already be in the correct shape.

On the other hand, we will have to modify the code defining the class of our model to take into account the multiple number of features.

In [79]:
```
class LSTMModel3:
    def __init__(self, input_shape, l2_reg_strength=0.05):
        self.input_shape = input_shape
        self.l2_reg_strength = l2_reg_strength

        # Call the '_build_model' method to construct the actual LSTM and store it
        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()

        # Using instance variable for regularization strength
        model.add(LSTM(units=50, return_sequences=True, input_shape=self.input_shap
                       kernel_regularizer=l2(self.l2_reg_strength),
                       recurrent_regularizer=l2(self.l2_reg_strength)))
        model.add(Dropout(0.2))

        model.add(LSTM(units=50,
                       kernel_regularizer=l2(self.l2_reg_strength),
                       recurrent_regularizer=l2(self.l2_reg_strength)))
        model.add(Dropout(0.2))

        model.add(Dense(1))
        return model
```

`train_scaled.shape[1]` captures the number of features, and we shall pass it to the `input_shape`:

In [80]:
```
input_shape = (seq_length, train_scaled.shape[1])
```

In [81]:
```
lstm_model = LSTMModel3(input_shape=input_shape)
lstm_model.model.summary()
```

```
Model: "sequential_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_30 (LSTM)              (None, 60, 50)            11000

 dropout_30 (Dropout)        (None, 60, 50)            0

 lstm_31 (LSTM)              (None, 50)                20200

 dropout_31 (Dropout)        (None, 50)                0

 dense_12 (Dense)            (None, 1)                 51

=================================================================
Total params: 31251 (122.07 KB)
Trainable params: 31251 (122.07 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [82]:
```
lstm_model.model.compile(optimizer='adam', loss='mean_squared_error')
```

In [83]:
```
checkpointer = ModelCheckpoint(
    filepath = 'weights_best_m.hdf5',
    verbose = 2,
    save_best_only = True
)
```

In [84]:
```
lstm_model.model.fit(
    X_train,
    y_train,
    epochs=50,
    batch_size = 64,
    callbacks = [checkpointer]
)
```

```
Epoch 1/50
83/83 [==============================] - ETA: 0s - loss: 4.4989WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 15s 87ms/step - loss: 4.4989
Epoch 2/50
83/83 [==============================] - ETA: 0s - loss: 0.6955WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.6955
Epoch 3/50
83/83 [==============================] - ETA: 0s - loss: 0.1131WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.1131
Epoch 4/50
83/83 [==============================] - ETA: 0s - loss: 0.0325WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0325
Epoch 5/50
83/83 [==============================] - ETA: 0s - loss: 0.0189WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.0189
Epoch 6/50
83/83 [==============================] - ETA: 0s - loss: 0.0141WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.0141
Epoch 7/50
83/83 [==============================] - ETA: 0s - loss: 0.0115WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.0115
Epoch 8/50
83/83 [==============================] - ETA: 0s - loss: 0.0100WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0100
Epoch 9/50
83/83 [==============================] - ETA: 0s - loss: 0.0091WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.0091
Epoch 10/50
83/83 [==============================] - ETA: 0s - loss: 0.0084WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.0084
Epoch 11/50
83/83 [==============================] - ETA: 0s - loss: 0.0080WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0080
Epoch 12/50
83/83 [==============================] - ETA: 0s - loss: 0.0077WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0077
Epoch 13/50
83/83 [==============================] - ETA: 0s - loss: 0.0073WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.0073
Epoch 14/50
83/83 [==============================] - ETA: 0s - loss: 0.0072WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0072
```

```
Epoch 15/50
83/83 [==============================] - ETA: 0s - loss: 0.0069WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0069
Epoch 16/50
83/83 [==============================] - ETA: 0s - loss: 0.0067WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0067
Epoch 17/50
83/83 [==============================] - ETA: 0s - loss: 0.0066WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.0066
Epoch 18/50
83/83 [==============================] - ETA: 0s - loss: 0.0066WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0066
Epoch 19/50
83/83 [==============================] - ETA: 0s - loss: 0.0065WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0065
Epoch 20/50
83/83 [==============================] - ETA: 0s - loss: 0.0060WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.0060
Epoch 21/50
83/83 [==============================] - ETA: 0s - loss: 0.0058WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0058
Epoch 22/50
83/83 [==============================] - ETA: 0s - loss: 0.0059WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0059
Epoch 23/50
83/83 [==============================] - ETA: 0s - loss: 0.0056WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 87ms/step - loss: 0.0056
Epoch 24/50
83/83 [==============================] - ETA: 0s - loss: 0.0056WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0056
Epoch 25/50
83/83 [==============================] - ETA: 0s - loss: 0.0054WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 8s 92ms/step - loss: 0.0054
Epoch 26/50
83/83 [==============================] - ETA: 0s - loss: 0.0055WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 8s 98ms/step - loss: 0.0055
Epoch 27/50
83/83 [==============================] - ETA: 0s - loss: 0.0055WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 8s 91ms/step - loss: 0.0055
Epoch 28/50
83/83 [==============================] - ETA: 0s - loss: 0.0053WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0053
```

```
Epoch 29/50
83/83 [==============================] - ETA: 0s - loss: 0.0052WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 89ms/step - loss: 0.0052
Epoch 30/50
83/83 [==============================] - ETA: 0s - loss: 0.0051WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0051
Epoch 31/50
83/83 [==============================] - ETA: 0s - loss: 0.0052WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0052
Epoch 32/50
83/83 [==============================] - ETA: 0s - loss: 0.0051WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 8s 91ms/step - loss: 0.0051
Epoch 33/50
83/83 [==============================] - ETA: 0s - loss: 0.0050WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 89ms/step - loss: 0.0050
Epoch 34/50
83/83 [==============================] - ETA: 0s - loss: 0.0047WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0047
Epoch 35/50
83/83 [==============================] - ETA: 0s - loss: 0.0049WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0049
Epoch 36/50
83/83 [==============================] - ETA: 0s - loss: 0.0046WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 89ms/step - loss: 0.0046
Epoch 37/50
83/83 [==============================] - ETA: 0s - loss: 0.0047WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0047
Epoch 38/50
83/83 [==============================] - ETA: 0s - loss: 0.0048WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0048
Epoch 39/50
83/83 [==============================] - ETA: 0s - loss: 0.0047WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 89ms/step - loss: 0.0047
Epoch 40/50
83/83 [==============================] - ETA: 0s - loss: 0.0045WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0045
Epoch 41/50
83/83 [==============================] - ETA: 0s - loss: 0.0047WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0047
Epoch 42/50
83/83 [==============================] - ETA: 0s - loss: 0.0045WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 89ms/step - loss: 0.0045
```

```
Epoch 43/50
83/83 [==============================] - ETA: 0s - loss: 0.0045WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 89ms/step - loss: 0.0045
Epoch 44/50
83/83 [==============================] - ETA: 0s - loss: 0.0043WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0043
Epoch 45/50
83/83 [==============================] - ETA: 0s - loss: 0.0043WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0043
Epoch 46/50
83/83 [==============================] - ETA: 0s - loss: 0.0044WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 89ms/step - loss: 0.0044
Epoch 47/50
83/83 [==============================] - ETA: 0s - loss: 0.0044WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0044
Epoch 48/50
83/83 [==============================] - ETA: 0s - loss: 0.0043WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 90ms/step - loss: 0.0043
Epoch 49/50
83/83 [==============================] - ETA: 0s - loss: 0.0044WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 88ms/step - loss: 0.0044
Epoch 50/50
83/83 [==============================] - ETA: 0s - loss: 0.0043WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
83/83 [==============================] - 7s 89ms/step - loss: 0.0043
```

Out[84]: `<keras.src.callbacks.History at 0x7fc95a7c8d60>`

After training we make our prediction:

In [85]:
```
# Compute the predicted values on the test set
test_predict = lstm_model.model.predict(X_test)
```

```
41/41 [==============================] - 2s 25ms/step
```

In order to undo the scaling transformation on our predictions, we have to keep in mind that the scaling was done for multiple features, not just the `Close` column as in the univariate case. We need to have an array with the same shape as the original dataset and pass it to the scaler. So we will create a dummy array filled with zeros with the right shape, with the column corresponding to our predictions in the same position as the `Close` column in our dataset.

In [86]:
```
# Create a dummy array filled with zeros, but same shape as original dataset
dummy_array = np.zeros((test_predict.shape[0], 4))

# Fill the column corresponding to 'Close' with test_predict values.
dummy_array[:, 0] = test_predict[:, 0]
```

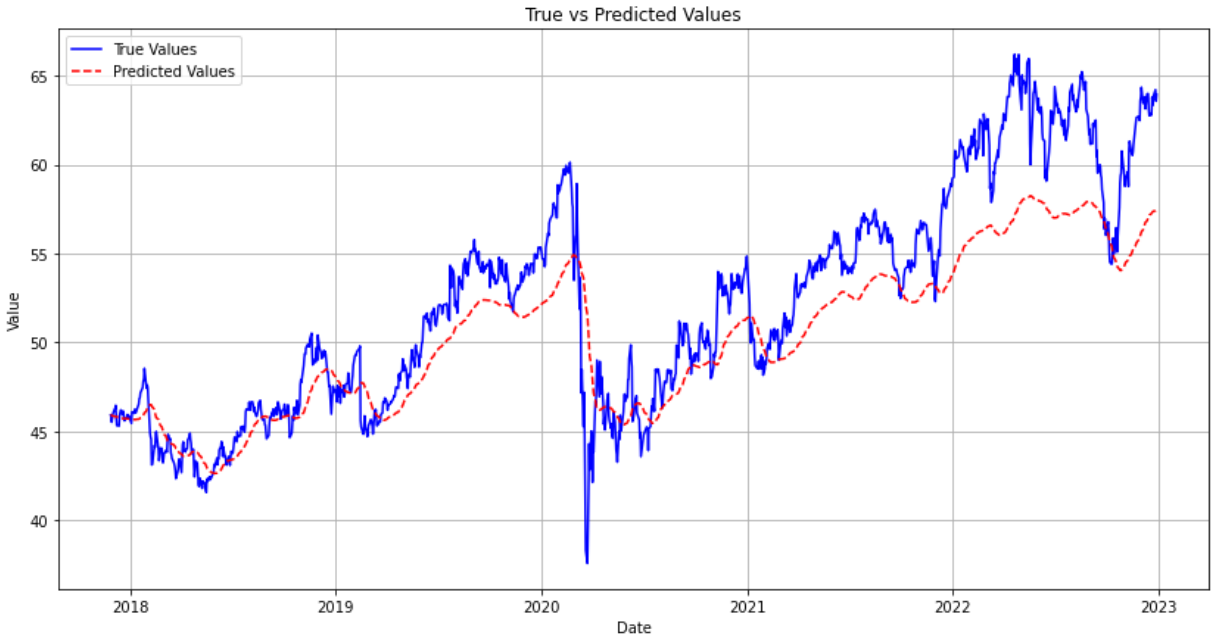
```
# Use inverse_transform
unscaled_array = scaler.inverse_transform(dummy_array)

# Extract unscaled predictions for 'Close'
test_predict_original = unscaled_array[:, 0]
```

We follow the same steps for unscaling the actual values of `Close` on the test set, taking into account that a reshape to a 2D array is needed:

In [87]:
```
y_test_reshaped = y_test.reshape(-1, 1)

# Create a dummy array filled with zeros of the right shape
dummy_array = np.zeros((y_test_reshaped.shape[0], 4))

# Fill the column corresponding to 'Close' with y_test_reshaped values
dummy_array[:, 0] = y_test_reshaped[:, 0]

# Use inverse_transform
unscaled_array = scaler.inverse_transform(dummy_array)

# Extract the unscaled values for 'Close'
y_test_unscaled = unscaled_array[:, 0]
```

We are now in busineess to plot our predictions versus actual values of `Close` on the test set.

In [88]:
```
# Extract the date-time index for the test set
test_dates = test.index

offset_test_dates = test_dates[seq_length:]
offset_test_dates = offset_test_dates[:-1]

# Plot the true values
plt.figure(figsize=(14, 7))
plt.plot(offset_test_dates, y_test_unscaled, label="True Values", color='blue')

# Plot the predicted values
plt.plot(offset_test_dates, test_predict_original, label="Predicted Values", color=

plt.title("True vs Predicted Values")
plt.xlabel("Date")
plt.ylabel("Value")
plt.legend()
plt.grid(True)
plt.show()
```



In [89]:
```
mse = mean_squared_error(y_test_unscaled, test_predict_original)
print(mse)
```

```
10.455389575860284
```

The introduction of new features seem to actually worsen the performance of our model. Some general reasons underlying this phenomenon are:

1. Added Noise: Some added features might introduce noise or collinearity which can confuse the model and degrade performance.
2. Need for a more complex model: A simple model that worked well for univariate data might not suffice to capture all the relationships between features and target for multivariate data.
3. Temporal Dynamics: In a multivariate setting we are also considering other features' past values aside from the target. If these extra features do not have a clear and consistent temporal relationship with the target, it can lead to errors.

An additional study to determine the actual causes of this worsening upon introduction of additional features is out of the scope of this work. We can nevertheless mention that the rather low volatility of our stock and the relatively good results obtained from the univariate case might already be pointing out that it might not be necessary to consider extra features.

2. Tesla stock

Univariate model

We shall repeat the previous analysis but focusing now on the TSLA stock. The purposeis again to compare how differently the LSTM model performs for a highly votalile stock, as

oppose to a stock with low volatility such as Coca-Cola.

In [7]:
```
df, train, test, train_scaled, test_scaled, scaler = scale_dataframe('TSLA.csv')
```

In [ ]:
```

```

In [8]:
```
# Create sequences from train and test data
X_train, y_train = create_sequences(train_scaled['Close'], seq_length)
X_test, y_test = create_sequences(test_scaled['Close'], seq_length)
```

In [9]:
```
# Reshape the inputs to be suitable for LSTM
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))
```

In [ ]:
```

```

In [10]:
```
lstm_model2 = LSTMModel((None, 1))
```

In [ ]:
```

```

In [11]:
```
lstm_model2.model.compile(optimizer='adam', loss='mean_squared_error')
```

In [ ]:
```

```

In [12]:
```
checkpointer2 = ModelCheckpoint(
    filepath = 'weights_best2.hdf5',
    verbose = 2,
    save_best_only = True
)
```

In [ ]:
```

```

In [13]:
```
lstm_model2.model.fit(
    X_train,
    y_train,
    epochs=25,
    batch_size = 32,
    callbacks = [checkpointer2]
)
```

```
Epoch 1/25
77/77 [==============================] - ETA: 0s - loss: 0.0030WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
77/77 [==============================] - 16s 89ms/step - loss: 0.0030
Epoch 2/25
77/77 [==============================] - ETA: 0s - loss: 0.0013WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 0.0013
Epoch 3/25
77/77 [==============================] - ETA: 0s - loss: 8.8886e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 87ms/step - loss: 8.8886e-04
Epoch 4/25
77/77 [==============================] - ETA: 0s - loss: 8.7696e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 8.7696e-04
Epoch 5/25
77/77 [==============================] - ETA: 0s - loss: 8.7476e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 85ms/step - loss: 8.7476e-04
Epoch 6/25
77/77 [==============================] - ETA: 0s - loss: 8.6827e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 8.6827e-04
Epoch 7/25
77/77 [==============================] - ETA: 0s - loss: 9.8427e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 9.8427e-04
Epoch 8/25
77/77 [==============================] - ETA: 0s - loss: 7.6437e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 7.6437e-04
Epoch 9/25
77/77 [==============================] - ETA: 0s - loss: 7.2496e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 7.2496e-04
Epoch 10/25
77/77 [==============================] - ETA: 0s - loss: 5.9099e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 87ms/step - loss: 5.9099e-04
Epoch 11/25
77/77 [==============================] - ETA: 0s - loss: 5.8935e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 88ms/step - loss: 5.8935e-04
Epoch 12/25
77/77 [==============================] - ETA: 0s - loss: 5.5801e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 5.5801e-04
Epoch 13/25
77/77 [==============================] - ETA: 0s - loss: 5.2589e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 87ms/step - loss: 5.2589e-04
Epoch 14/25
77/77 [==============================] - ETA: 0s - loss: 5.7145e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 85ms/step - loss: 5.7145e-04
```

```
Epoch 15/25
77/77 [==============================] - ETA: 0s - loss: 6.4421e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 6.4421e-04
Epoch 16/25
77/77 [==============================] - ETA: 0s - loss: 5.3308e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 85ms/step - loss: 5.3308e-04
Epoch 17/25
77/77 [==============================] - ETA: 0s - loss: 7.4622e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 7.4622e-04
Epoch 18/25
77/77 [==============================] - ETA: 0s - loss: 5.5462e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 88ms/step - loss: 5.5462e-04
Epoch 19/25
77/77 [==============================] - ETA: 0s - loss: 4.9312e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 4.9312e-04
Epoch 20/25
77/77 [==============================] - ETA: 0s - loss: 4.9949e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 4.9949e-04
Epoch 21/25
77/77 [==============================] - ETA: 0s - loss: 4.2326e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 4.2326e-04
Epoch 22/25
77/77 [==============================] - ETA: 0s - loss: 4.4901e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 86ms/step - loss: 4.4901e-04
Epoch 23/25
77/77 [==============================] - ETA: 0s - loss: 5.5908e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 89ms/step - loss: 5.5908e-04
Epoch 24/25
77/77 [==============================] - ETA: 0s - loss: 4.8420e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 87ms/step - loss: 4.8420e-04
Epoch 25/25
77/77 [==============================] - ETA: 0s - loss: 4.7496e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 7s 90ms/step - loss: 4.7496e-04
```

Out[13]: <keras.src.callbacks.History at 0x7f1da0c547c0>

In [14]:
```
# Compute the predicted values on the test set
test_predict = lstm_model2.model.predict(X_test)

# Undo the scaling transformation on the predicted values 'test_predict' and the ac
test_predict = scaler.inverse_transform(test_predict)
```

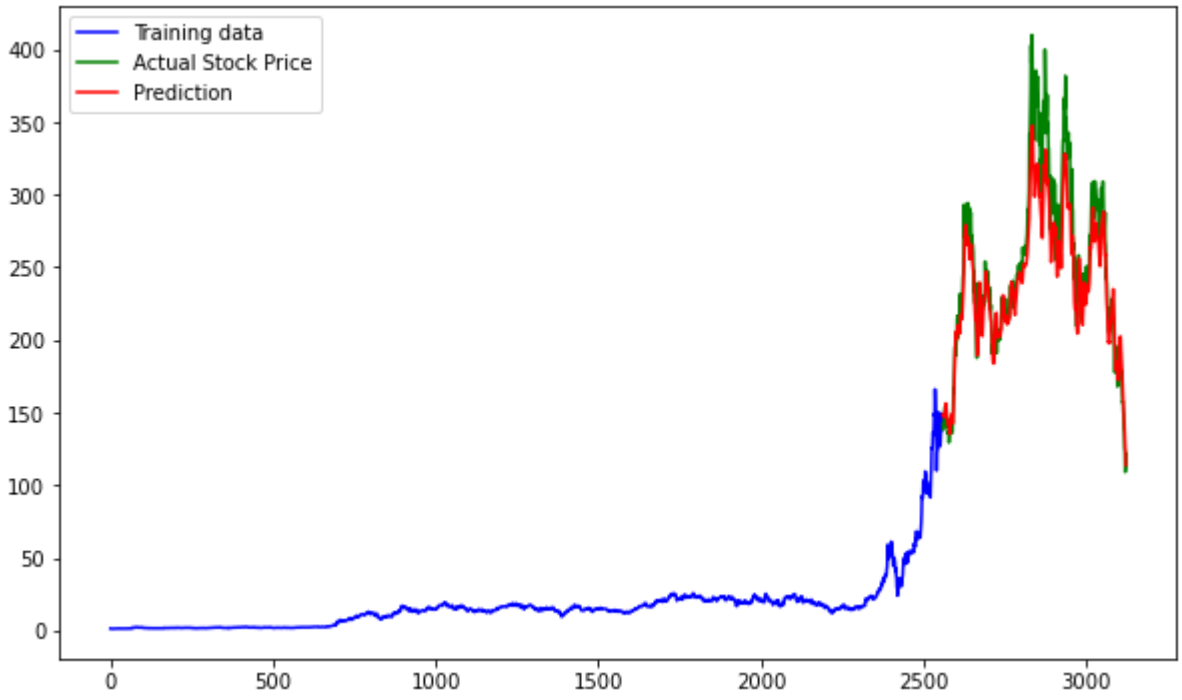
```
18/18 [==============================] - 2s 33ms/step
```

In [15]:
```
y_test_reshaped = y_test.reshape(-1, 1)
```

```
y_test = scaler.inverse_transform(y_test_reshaped)
```

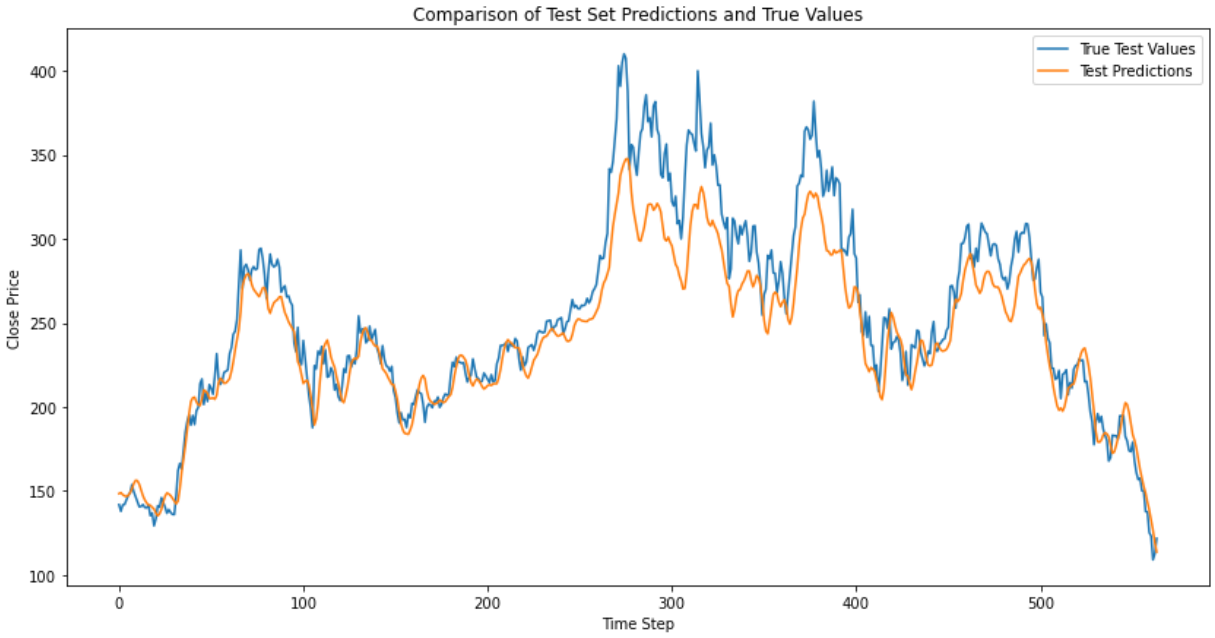
In [ ]:

In [16]:
```
plt.figure(figsize=(10,6))
plt.plot(df.index[:len(train)+seq_length], df['Close'][:len(train)+seq_length], col
plt.plot(df.index[len(train)+seq_length:-1], df['Close'][len(train)+seq_length:-1],
plt.plot(df.index[len(train)+seq_length:-1], test_predict, color='red', label='Pred
plt.legend()
plt.show()
```



In [ ]:

In [17]:
```
# Plot test predictions and actual values
plt.figure(figsize=(14,7))
plt.plot(y_test, label='True Test Values')
plt.plot(test_predict, label='Test Predictions')
plt.title('Comparison of Test Set Predictions and True Values')
plt.xlabel('Time Step')
plt.ylabel('Close Price')
plt.legend()
plt.show()
```



In [18]:
```
mse = mean_squared_error(y_test, test_predict)
print(mse)
```

```
533.4336816324611
```

The model seems to capture the overall behavior of the data but suffers from a high MSE: the RMSE is around 23 dollars, which is high even for the scale of prices we are dealing with here.

We are going to check now how increasing the number of units in our layers affects the MSE metric. We will increase from 50 to 100 the units of the first two layers.

In [21]:
```
class LSTMModel4:
    # Call the '__init__' method to perform initializations and set up attributes
    def __init__(self, input_shape=(None,1)):
        self.input_shape = input_shape

        # Call the '_build_model' method to construct the actual LSTM and store it
        self.model = self._build_model()

    # Define the architecture of the LSTM neural network
    def _build_model(self):
        model = Sequential()

        model.add(LSTM(units=100, return_sequences=True, input_shape=self.input_sha
        model.add(Dropout(0.2))

        model.add(LSTM(units=100, return_sequences=True))
        model.add(Dropout(0.2))

        model.add(LSTM(units=50))
        model.add(Dropout(0.2))

        model.add(Dense(units=1))
```

```
        return model
```

In [ ]:

In [22]:
```
lstm_model2 = LSTMModel4((None, 1))
```

In [23]:
```
lstm_model2.model.compile(optimizer='adam', loss='mean_squared_error')
```

In [24]:
```
lstm_model2.model.fit(
    X_train,
    y_train,
    epochs=50,
    batch_size = 32,
    callbacks = [checkpointer2]
)
```

```
Epoch 1/50
77/77 [==============================] - ETA: 0s - loss: 0.0025WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
77/77 [==============================] - 20s 130ms/step - loss: 0.0025
Epoch 2/50
77/77 [==============================] - ETA: 0s - loss: 0.0011WARNING:tensorflow:Ca
n save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 130ms/step - loss: 0.0011
Epoch 3/50
77/77 [==============================] - ETA: 0s - loss: 7.3091e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 130ms/step - loss: 7.3091e-04
Epoch 4/50
77/77 [==============================] - ETA: 0s - loss: 9.0448e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 130ms/step - loss: 9.0448e-04
Epoch 5/50
77/77 [==============================] - ETA: 0s - loss: 8.8358e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 130ms/step - loss: 8.8358e-04
Epoch 6/50
77/77 [==============================] - ETA: 0s - loss: 7.3916e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 131ms/step - loss: 7.3916e-04
Epoch 7/50
77/77 [==============================] - ETA: 0s - loss: 7.6411e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 130ms/step - loss: 7.6411e-04
Epoch 8/50
77/77 [==============================] - ETA: 0s - loss: 6.7496e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 130ms/step - loss: 6.7496e-04
Epoch 9/50
77/77 [==============================] - ETA: 0s - loss: 7.2491e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 134ms/step - loss: 7.2491e-04
Epoch 10/50
77/77 [==============================] - ETA: 0s - loss: 6.6411e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 130ms/step - loss: 6.6411e-04
Epoch 11/50
77/77 [==============================] - ETA: 0s - loss: 6.1566e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 126ms/step - loss: 6.1566e-04
Epoch 12/50
77/77 [==============================] - ETA: 0s - loss: 5.3870e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 129ms/step - loss: 5.3870e-04
Epoch 13/50
77/77 [==============================] - ETA: 0s - loss: 5.3412e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 128ms/step - loss: 5.3412e-04
Epoch 14/50
77/77 [==============================] - ETA: 0s - loss: 5.0913e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 130ms/step - loss: 5.0913e-04
```

```
Epoch 15/50
77/77 [==============================] - ETA: 0s - loss: 5.8659e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 126ms/step - loss: 5.8659e-04
Epoch 16/50
77/77 [==============================] - ETA: 0s - loss: 4.8934e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 134ms/step - loss: 4.8934e-04
Epoch 17/50
77/77 [==============================] - ETA: 0s - loss: 7.1509e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 148ms/step - loss: 7.1509e-04
Epoch 18/50
77/77 [==============================] - ETA: 0s - loss: 4.5803e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 134ms/step - loss: 4.5803e-04
Epoch 19/50
77/77 [==============================] - ETA: 0s - loss: 4.0422e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 141ms/step - loss: 4.0422e-04
Epoch 20/50
77/77 [==============================] - ETA: 0s - loss: 4.8127e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 137ms/step - loss: 4.8127e-04
Epoch 21/50
77/77 [==============================] - ETA: 0s - loss: 3.4809e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 142ms/step - loss: 3.4809e-04
Epoch 22/50
77/77 [==============================] - ETA: 0s - loss: 3.8508e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 137ms/step - loss: 3.8508e-04
Epoch 23/50
77/77 [==============================] - ETA: 0s - loss: 3.8269e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 141ms/step - loss: 3.8269e-04
Epoch 24/50
77/77 [==============================] - ETA: 0s - loss: 4.0433e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 144ms/step - loss: 4.0433e-04
Epoch 25/50
77/77 [==============================] - ETA: 0s - loss: 4.0434e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 137ms/step - loss: 4.0434e-04
Epoch 26/50
77/77 [==============================] - ETA: 0s - loss: 3.8438e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 143ms/step - loss: 3.8438e-04
Epoch 27/50
77/77 [==============================] - ETA: 0s - loss: 3.3938e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 145ms/step - loss: 3.3938e-04
Epoch 28/50
77/77 [==============================] - ETA: 0s - loss: 3.2426e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 142ms/step - loss: 3.2426e-04
```

```
Epoch 29/50
77/77 [==============================] - ETA: 0s - loss: 3.8651e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 143ms/step - loss: 3.8651e-04
Epoch 30/50
77/77 [==============================] - ETA: 0s - loss: 4.2349e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 147ms/step - loss: 4.2349e-04
Epoch 31/50
77/77 [==============================] - ETA: 0s - loss: 3.6016e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 146ms/step - loss: 3.6016e-04
Epoch 32/50
77/77 [==============================] - ETA: 0s - loss: 4.1406e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 12s 151ms/step - loss: 4.1406e-04
Epoch 33/50
77/77 [==============================] - ETA: 0s - loss: 3.6635e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 144ms/step - loss: 3.6635e-04
Epoch 34/50
77/77 [==============================] - ETA: 0s - loss: 3.2006e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 143ms/step - loss: 3.2006e-04
Epoch 35/50
77/77 [==============================] - ETA: 0s - loss: 3.1119e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 136ms/step - loss: 3.1119e-04
Epoch 36/50
77/77 [==============================] - ETA: 0s - loss: 3.4473e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 132ms/step - loss: 3.4473e-04
Epoch 37/50
77/77 [==============================] - ETA: 0s - loss: 4.0923e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 141ms/step - loss: 4.0923e-04
Epoch 38/50
77/77 [==============================] - ETA: 0s - loss: 3.4304e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 11s 137ms/step - loss: 3.4304e-04
Epoch 39/50
77/77 [==============================] - ETA: 0s - loss: 3.0391e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 129ms/step - loss: 3.0391e-04
Epoch 40/50
77/77 [==============================] - ETA: 0s - loss: 4.3926e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 129ms/step - loss: 4.3926e-04
Epoch 41/50
77/77 [==============================] - ETA: 0s - loss: 2.5096e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 129ms/step - loss: 2.5096e-04
Epoch 42/50
77/77 [==============================] - ETA: 0s - loss: 2.6892e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 128ms/step - loss: 2.6892e-04
```

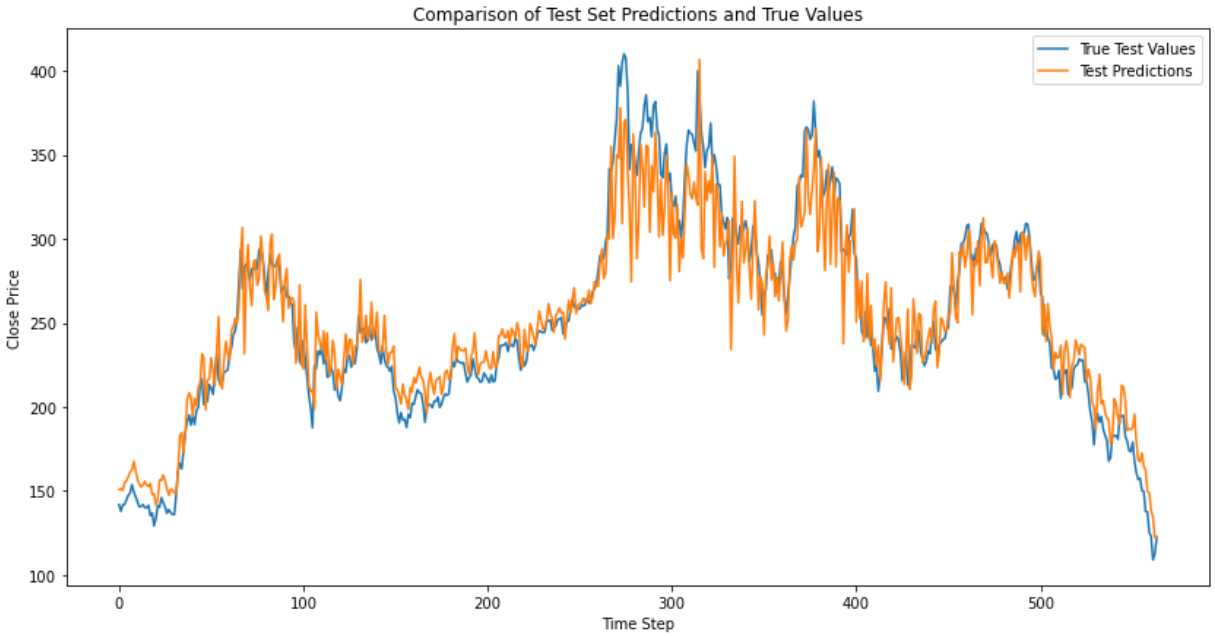
```
Epoch 43/50
77/77 [==============================] - ETA: 0s - loss: 3.2835e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 129ms/step - loss: 3.2835e-04
Epoch 44/50
77/77 [==============================] - ETA: 0s - loss: 2.8626e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 136ms/step - loss: 2.8626e-04
Epoch 45/50
77/77 [==============================] - ETA: 0s - loss: 3.0700e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 129ms/step - loss: 3.0700e-04
Epoch 46/50
77/77 [==============================] - ETA: 0s - loss: 2.7534e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 131ms/step - loss: 2.7534e-04
Epoch 47/50
77/77 [==============================] - ETA: 0s - loss: 2.7680e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 132ms/step - loss: 2.7680e-04
Epoch 48/50
77/77 [==============================] - ETA: 0s - loss: 3.3493e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 129ms/step - loss: 3.3493e-04
Epoch 49/50
77/77 [==============================] - ETA: 0s - loss: 3.6740e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 135ms/step - loss: 3.6740e-04
Epoch 50/50
77/77 [==============================] - ETA: 0s - loss: 2.9313e-04WARNING:tensorflo
w:Can save best model only with val_loss available, skipping.
77/77 [==============================] - 10s 129ms/step - loss: 2.9313e-04
```

Out[24]: <keras.src.callbacks.History at 0x7fec2ae5cc10>

In [25]:
```
# Compute the predicted values on the test set
test_predict = lstm_model2.model.predict(X_test)

# Undo the scaling transformation on the predicted values 'test_predict' and the ac
test_predict = scaler.inverse_transform(test_predict)
```

```
18/18 [==============================] - 3s 50ms/step
```

In [26]:
```
# Plot test predictions and actual values
plt.figure(figsize=(14,7))
plt.plot(y_test, label='True Test Values')
plt.plot(test_predict, label='Test Predictions')
plt.title('Comparison of Test Set Predictions and True Values')
plt.xlabel('Time Step')
plt.ylabel('Close Price')
plt.legend()
plt.show()
```



In [27]:
```
mse = mean_squared_error(y_test, test_predict)
print(mse)
```

```
396.5515904628778
```

We conclude that increasing the number of units in the first two layers significantly reduces the MSE but the model seems to exceedingly learn from noise and overfits. One could address this issue by introducing regularization, as we did for the Coca-Cola dataset, tuning other hyperparameters and aspects of the model architecture, and also investigating the effect of introducing additional features.

Conclusions

Our analysis of the stock price forecasting capabilities of Random Forests abd LSTM netowrks for Coca-Cola (KO) and Tesla (TSLA) has resulted in several nuanced insights, achievements, and areas for potential improvement:

Insights from Stock Analysis:

1. Coca-Cola (KO): The analysis of KO stock revealed a pattern of steady growth and lower volatility, characteristic of a mature company with a stable market presence. This predictability in the stock's behavior was reflected in the models' ability to forecast with a higher degree of accuracy.
2. Tesla (TSLA): In contrast, TSLA's stock exhibited higher volatility and unpredictability, a trait often seen in growth-oriented technology stocks. This presented a more challenging scenario for accurate forecasting, as the stock was more susceptible to market dynamics and external influences.

Milestones Reached:

1. Comprehensive data collection and processing: the first step was the collection and preprocessing of a large dataset of stock prices and earnings for two stocks with very different behavior over time: KO and TSLA.

2. Development of predictive models: building and fine-tuning two advanced machine learning models (Random Forests and LSTM neural networks) to forecast stock prices.

3. Comparative Analysis: we performed a thorough comparative analysis of the two models for the KO and TSLA stocks, providing valuable insights into their relative strengths and weaknesses in the context of stock price forecasting.

Model Performance and Suitability:

1. Random Forests: These models, despite their non-linear capabilities, were somewhat limited in handling the temporal dependencies crucial in stock price data. Best suited for stable markets with less temporal dependency. They may struggle during periods of high market volatility or when sudden, unforeseen events impact the market.

2. LSTM Neural Networks: The strength of LSTMs lay in their ability to model sequential and time-dependent data. Ideal for markets with strong temporal dependencies and longer-term trends. However, they may produce high errors in markets lacking discernible patterns or during extreme, unpredictable market events.

Conditions Leading to Failure or High Error:

Both models might struggle during periods of high market volatility or when unexpected events (like geopolitical events or global economic crises) significantly impact the market. These scenarios can introduce patterns and volatilities not present in the training data.

LSTMs might produce intolerable errors if the stock price movement becomes highly erratic and lacks any discernible trend or pattern, as their predictions are heavily based on past sequences.

Random Forests may fail in scenarios where the market's movement is predominantly influenced by factors not captured in the model's features.

Limitations and Future Enhancements:

Data scope: a fundamental limitation of our approach is the reliance on historical prices and company earnings. Incorporating a broader range of data, such as market sentiment or macroeconomic factors, could potentially enhance model accuracy. This could be particularly relevant for the TSLA stock: being a newer stock, there is a lesser amount of price data available, and the high volatility and different phases of behavior of the stock price make the relative scarcity of data even more problematic for learning purposes.

Model complexity and overfitting: Both models, especially the LSTM, faced challenges with overfitting. Future work could explore more sophisticated regularization techniques or model architectures to address this.

Real-time Forecasting Capability: The current models are trained on historical data. Adapting these models for real-time forecasting in the ever-changing stock market environment remains a challenge.

More complex deep learning architectures: Advanced neural networks like Convolutional Neural Networks (CNNs) for pattern recognition in time series, or more intricate Recurrent Neural Network (RNN) variants like GRU (Gated Recurrent Units) can be explored. These models might capture complex patterns in stock price data more effectively.

In [ ]: