IFTE0004_18004520-This

March 31, 2023

[1]:
```python
# import needed modules
!pip install yfinance
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import norm
import numpy as np
import pandas as pd
import pandas_datareader.data as pdr
import yfinance as yf
import datetime as dt
import random as rn
from sklearn.metrics import mean_squared_error
import warnings
warnings.filterwarnings("ignore")
from pandas_datareader import data as pdr
import yfinance as yf
yf.pdr_override()
import seaborn as sns
%matplotlib inline

import keras
import tensorflow as tf
from tensorflow.keras.models import Sequential
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.models import Model
from statsmodels.tsa.arima.model import ARIMA
from math import exp, sqrt
from math import sqrt
import glob
import os
import statsmodels.api as sm
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller as adf
from sklearn.metrics import mean_absolute_percentage_error
from sklearn.preprocessing import StandardScaler
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: yfinance in /usr/local/lib/python3.9/dist-
packages (0.2.14)
Requirement already satisfied: pytz>=2022.5 in /usr/local/lib/python3.9/dist-
packages (from yfinance) (2022.7.1)
Requirement already satisfied: html5lib>=1.1 in /usr/local/lib/python3.9/dist-
packages (from yfinance) (1.1)
Requirement already satisfied: beautifulsoup4>=4.11.1 in
/usr/local/lib/python3.9/dist-packages (from yfinance) (4.11.2)
Requirement already satisfied: multitasking>=0.0.7 in
/usr/local/lib/python3.9/dist-packages (from yfinance) (0.0.11)
Requirement already satisfied: requests>=2.26 in /usr/local/lib/python3.9/dist-
packages (from yfinance) (2.27.1)
Requirement already satisfied: numpy>=1.16.5 in /usr/local/lib/python3.9/dist-
packages (from yfinance) (1.22.4)
Requirement already satisfied: cryptography>=3.3.2 in
/usr/local/lib/python3.9/dist-packages (from yfinance) (40.0.1)
Requirement already satisfied: pandas>=1.3.0 in /usr/local/lib/python3.9/dist-
packages (from yfinance) (1.4.4)
Requirement already satisfied: lxml>=4.9.1 in /usr/local/lib/python3.9/dist-
packages (from yfinance) (4.9.2)
Requirement already satisfied: appdirs>=1.4.4 in /usr/local/lib/python3.9/dist-
packages (from yfinance) (1.4.4)
Requirement already satisfied: frozendict>=2.3.4 in
/usr/local/lib/python3.9/dist-packages (from yfinance) (2.3.6)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.9/dist-
packages (from beautifulsoup4>=4.11.1->yfinance) (2.4)
Requirement already satisfied: cffi>=1.12 in /usr/local/lib/python3.9/dist-
packages (from cryptography>=3.3.2->yfinance) (1.15.1)
Requirement already satisfied: webencodings in /usr/local/lib/python3.9/dist-
packages (from html5lib>=1.1->yfinance) (0.5.1)
Requirement already satisfied: six>=1.9 in /usr/local/lib/python3.9/dist-
packages (from html5lib>=1.1->yfinance) (1.16.0)
Requirement already satisfied: python-dateutil>=2.8.1 in
/usr/local/lib/python3.9/dist-packages (from pandas>=1.3.0->yfinance) (2.8.2)
Requirement already satisfied: charset-normalizer~=2.0.0 in
/usr/local/lib/python3.9/dist-packages (from requests>=2.26->yfinance) (2.0.12)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.9/dist-packages (from requests>=2.26->yfinance)
(2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/usr/local/lib/python3.9/dist-packages (from requests>=2.26->yfinance) (1.26.15)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-
packages (from requests>=2.26->yfinance) (3.4)
Requirement already satisfied: pycparser in /usr/local/lib/python3.9/dist-
packages (from cffi>=1.12->cryptography>=3.3.2->yfinance) (2.21)
```

[1]:

1 1.1. Short dissertation

This essay examines the empirical facts characterizing stock returns, focusing on distributional properties, dynamics, and risk features. We discuss how these properties change as a function of the time scale , ranging from high-frequency time scales (e.g., tick-by-tick, 1 min, 5 min) to longer ones (e.g., daily, weekly, or monthly returns). Furthermore, we explore the role of Geometric Brownian Motion (GBM) in capturing these empirical facts, highlighting its strengths and limitations. Through analysing studies, we aim to provide a comprehensive understanding of stock characteristics and the applicability of GBM in representing these empirical features.

Dynamics

In highly liquid markets, autocorrelation in stock returns is typically insignificant, for periods longer than 15 minutes, it can be assumed to be zero (Cont et al., 1997). This results from statistical arbitrage, as trading opportunities from autocorrelations are quickly exploited away. At higher frequencies e.g below 5 minutes, negative autocorrelation occurs due to the bid-ask bounce phenomenon. This is caused by stock prices oscillating between bid and ask prices, as people transact closer to the bid or ask the subsequent transactions revert towards the mean as this provides the next best transaction. Market makers contribute to this mean reversion at the tick level (Cont, 2000). Cont (2000) supported this negative autocorrelation through finding a negative autocorrelation in KLM shares at a tick-by-tick level.

However, at longer durations, autocorrelation is present. Lo and MacKinlay (1999) found positive autocorrelation in weekly returns, while Guo (2019) found varying autocorrelation in monthly returns. Guo's study revealed positive autocorrelation in the first half of firm's reporting cycles to negative in the next, attributed to investors' behaviour and their overreaction to inconsistent earnings news.

Risk Features

Anderson et al. (2000) discovered strong autocorrelation in return volatility, which represents the risk features of returns. Cont (2000) stated that this positive autocorrelation decays slowly over days or even weeks. Dracogna et al (1993) supported this using absolute and squared returns as proxies for volatility, finding significant positive autocorrelations. The slow decay of these autocorrelation is evidenced by Ding et al (1993) who found that the autocorrelation of the first lag for absolute returns was 0.318 decreasing to only 0.162 by 100 lags. This leads to volatility clustering – large volatility followed by large volatility - and non-stationarity of returns. Additionally, volatility is asymmetric, with negative returns correlating with greater volatility than equal positive returns, suggesting losses may be more severe and harder to predict (Black, 1976). This phenomenon can manifest as the leverage effect which suggests as returns decrease, volatility increases.

Furthermore, Anderson et al. (2000) observed that as volatility rises, correlations between equity returns also increase, reducing the protection of diversified portfolios during extreme events. Ang & Bekaert (2002) showed that during high volatility periods, correlations between stocks and market factors, such as macroeconomic variables, also increase. This "conditional correlation" effect, driven by common responses to market stress, further diminishes portfolio diversification benefits.

Distribution

Stock return distributions deviate from normality, exhibiting leptokurtic features, which implies higher peaks and fatter tail and thus more frequent extreme events than predicted by normal distributions. This is supported by Taylor (2005) who showed in daily returns data that the sample kurtosis was always at least 10, although ths may decrease as time increase which will be discussed later. Furthermore, it's found return distribution tails are more aligned with power-law or Pareto-like tails. This is supported through the extreme value theorem which states if there exists a non-degenerate limit distribution H for the normalized maximum returns of an IID log return sequence, then the limit distribution H belongs to one of three classes: Gumbel, Weibull, or Frechet. The class of the limit distribution is determined by the sign of the shape parameter , which is estimated to be between 0.2 and 0.4 Cont (2000) which would indicate heavy tails belonging to the Frechet domain of attraction with tail index between 2 and 5, hence confirming the improbability of a normal distribution.

Viswanathan et al. (2003), states fat tails are caused by a combination of several factors, including the aforementioned leverage effects, asymmetric volatility, and long memory each of which leads to a higher frequency of large negative returns. This contributes to the asymmetry of the stock return, with steeper drawdowns than recoveries. Gabaix et al. (2003) analyzed stock returns from 1926-2000, discovering a highly asymmetric distribution with a longer left tail. Ignoring this asymmetry could lead investors to underestimate downside risks and overestimate upside potential.

Although the true return distribution remains unidentified, the above features suggest a parametric model needs a location parameter, a scale (volatility) parameter, a tail decay parameter, and an asymmetry parameter, allowing left and right tails to behave differently.

Distribution Over Time

The distributional properties of returns depend on data frequency. At higher frequencies (<20 minutes), the distribution is more peaked with fatter tails, aligning with previously discussed characteristics (Cont 2000). As the time scale increases, the distribution gradually converges to a standard normal, a phenomenon known as aggregational Gaussianity. As log returns are additive one might expect this convergence to occur at time scales larger than just the highest frequency, e.g <5 minutes, due to the central limit theorem. However, Amaral et al. (2000) discovered that the distribution remained strongly leptokurtic from a data frequency of 5 minutes to 16 days before slowly converging to Gaussian, becoming approximately normal at 1 year. Additionally, in the book Financial Econometrics Unit 1 by the University of London log returns of the S&P 500 were analysed, daily, weekly, and monthly, finding negatively skewed distributions in all cases, with kurtosis declining from 11.64 to 10.27 to 4.58, respectively, supporting the notion of declining kurtosis as the time scale increases and hence increasing Gaussanity.

Empirical facts captured by GBM

Geometric Brownian Motion (GBM) assumes daily returns are Independent and Identically Distributed (IID), therefore capturing the stylised fact of no autocorrelation between returns. However, while GBM captures no autocorrelation in stock returns, Cont (2000) argues that IID returns would also show no autocorrelation in nonlinear combinations, which is not observed empirically for absolute and squared returns.

Additionally, GBM assumes continuous compounded returns are normally distributed. Therefore GBM would be able to achieve the stylised fact of aggregational Gaussianity at large time scales, previously stated to be 1 year.

Nonetheless, although GBM captures certain stylized facts, such as no autocorrelation in returns and Gaussianity at some time scales, it fails to account for other empirical features like volatility clustering, fat-tailed distributions, and asymmetry. As a result, alternative models have been developed to better represent these characteristics, such as GARCH models and stochastic volatility models.

References:

Cont, R. "Empirical Properties of Asset Returns: Stylized Facts and Statistical Issues." Quantitative Finance, vol. 1, no. 2, Feb. 2001, pp. 223–236, https://doi.org/10.1080/713665670.

Cont, Rama, et al. Scaling in Stock Market Data: Stable Laws and Beyond. 1997, rama.cont.perso.math.cnrs.fr/pdf/TemperedStable1997.pdf. Accessed 23 Mar. 2023.

Lo, Andrew, and Craig MacKinlay. A Non-Random Walk down Wall St. 1999, assets.press.princeton.edu/chapters/s6558.pdf. Accessed 20 Mar. 2023.

Guo, Hongye. Underreaction, Overreaction, and Dynamic Autocorrelation of Stock Returns. 2019, jacobslevycenter.wharton.upenn.edu/wp-content/uploads/2020/02/2019.12.16-Underreaction-Overreaction-and-Dynamic-Autocorrelation-of-Stock-Returns.pdf. Accessed 21 Mar. 2023.

Andersen, T. "The Distribution of Realized Stock Return Volatility." Journal of Financial Economics, vol. 61, no. 1, July 2001, pp. 43–76, https://doi.org/10.1016/s0304-405x(01)00055-1. Accessed 25 Mar. 2023.

Black, Fischer. "Studies of Stock Price Volatility Changes." Scirp.org, 1976. Accessed 27 Mar. 2023.

Ang, Andrew, and Geert Bekaert. "International Asset Allocation with Regime Shifts." Review of Financial Studies, vol. 15, no. 4, July 2002, pp. 1137–1187, https://doi.org/10.1093/rfs/15.4.1137.

Astakhov, Sergey A., et al. "Chaos-Assisted Capture of Irregular Moons." Nature, vol. 423, no. 6937, May 2003, pp. 264–267, https://doi.org/10.1038/nature01622. Accessed 27 Mar. 2023.

Amaral, Nilson Figueiredo, et al. The Distribution of Returns of Stock Prices. July 2000, www.researchgate.net/publication/228931664_The_distribution_of_returns_of_stock_prices. Accessed 28 Mar. 2023.

Viswanathan, G.M., et al. "The Origin of Fat-Tailed Distributions in Financial Time Series." Physica A: Statistical Mechanics and Its Applications, vol. 329, no. 1-2, Nov. 2003, pp. 273–280, https://doi.org/10.1016/s0378-4371(03)00608-3. Accessed 21 Mar. 2023.

Taylor, S. J. (2005). Asset Price Dynamics, Volatility, and Prediction. Princeton, NJ: Princeton University Press

Ding, Z, Granger, C., and Engle, R. F. (1993). "A long memory property of stock market returns and a new model". Journal of Empirical Finance

Dacorogna, Michael M., et al. "A Geographical Model for the Daily and Weekly Seasonal Volatility in the Foreign Exchange Market." Journal of International Money and Finance, vol. 12, no. 4, 1 Aug. 1993, pp. 413–438, www.sciencedirect.com/science/article/pii/026156069390004U, https://doi.org/10.1016/0261-5606(93)90004-U. Accessed 21 Mar. 2023.

2 1.2. Python coding: forecasting stock prices.

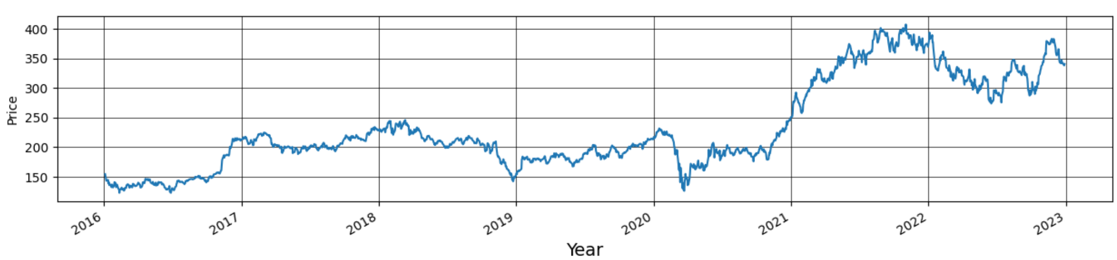
GBM model

[2]:
```
# Importing and setting up the pandas-datareader package to override any␣
↪potential future API changes.
yf.pdr_override()
# These two lines define the start and end dates for the data retrieval.
start = dt.datetime(2016,1,4)
end = dt.datetime(2022,12,30)
# This line uses the pandas-datareader package to retrieve daily stock data for␣
↪the stock symbol 'GS' (Goldman Sachs)
# between the start and end dates provided. The data is returned as a pandas␣
↪DataFrame and assigned to the variable 'GS'.
GS = pdr.get_data_yahoo('GS', start, end, interval='1d')
```

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

[3]:
```
# This line plots the adjusted closing price of the Goldman Sachs stock, which␣
↪is accessed from the DataFrame
#'GS' using the 'Adj Close' column.
GS['Adj Close'].plot(figsize=(15, 3))
# This line adds an x-axis label to the plot with the text 'Year'
plt.xlabel('Year', fontsize=14)
# This line adds a y-axis label to the plot with the text 'Price'.
plt.ylabel('Price')
# This line turns on the grid for the plot.
plt.grid()
# This line sets the grid line style to solid, with a width of 0.5 points, and␣
↪a color of black.
plt.grid(which="major", color='k', linestyle='-', linewidth=0.5)

plt.show
```

[3]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



[4]:
```python
# This line extracts the 'Adj Close' column from the existing 'GS' DataFrame␣
↪and assigns it back to the 'GS' variable.
GS = GS['Adj Close']
# This line creates a new pandas DataFrame from the 'GS' variable, effectively␣
↪converting it from a pandas Series to a DataFrame.
GS = pd.DataFrame(GS)
GS
```

[4]:
```
             Adj Close
Date
2016-01-04  154.552505
2016-01-05  151.891373
2016-01-06  148.183319
2016-01-07  143.628906
2016-01-08  143.035614
…                    …
2022-12-22  343.123138
2022-12-23  343.053650
2022-12-27  339.538818
2022-12-28  338.446625
2022-12-29  340.988434

[1761 rows x 1 columns]
```

[5]:
```python
# This line creates a new column 'Year' in the 'GS' DataFrame and assigns it␣
↪the year value from the index of the DataFrame, as a string.
GS['Year'] = GS.index.year.astype(str)
# This line creates a new column 'Month' in the 'GS' DataFrame and assigns it␣
↪the month value from the index of the DataFrame, as a string.
GS['Month'] = GS.index.month.astype(str)
# This line creates a new column 'Day' in the 'GS' DataFrame and assigns it the␣
↪day value from the index of the DataFrame, as a string.
GS['Day'] = GS.index.day.astype(str)
# This line concatenates the 'Year', 'Month', and 'Day' columns of the 'GS'␣
↪DataFrame into a single string and assigns it as the new index of the␣
↪DataFrame.
GS['Day'] = GS.index.day.astype(str)
GS.index = GS['Year'].str.cat(GS[['Month','Day']], sep='-')
```

[6]:
```python
# This drops the unwanted columns
GS.drop(['Year', 'Month', 'Day'], axis=1, inplace=True)
```

[7]:
```python
# calculate the logarithm of the real prices
GS['log_price'] = np.log(GS['Adj Close'])
# find the log return
GS['log_ret'] = GS['log_price'].diff()
```

[8]:
```python
# Define date ranges
train_start = '2016-1-4'
train_end = '2021-12-31'
test_start = '2022-1-3'

train = GS[:train_end] # all the data points up to and including 2021-12-31
test = GS[test_start:] # all the data points after 2022-1-3
```

[9]:
```python
train
```

[9]:
```
             Adj Close  log_price   log_ret
Year
2016-1-4    154.552505   5.040534       NaN
2016-1-5    151.891373   5.023166 -0.017368
2016-1-6    148.183319   4.998450 -0.024715
2016-1-7    143.628906   4.967233 -0.031217
2016-1-8    143.035614   4.963094 -0.004139
…                  …          …         …
2021-12-27  375.377716   5.927933  0.007761
2021-12-28  374.961731   5.926824 -0.001109
2021-12-29  373.597778   5.923180 -0.003644
2021-12-30  372.939941   5.921417 -0.001762
2021-12-31  370.066895   5.913684 -0.007734

[1511 rows x 3 columns]
```

[10]:
```python
train_log_returns = train['log_ret']
```

[11]:
```python
# This line calculates the mean, standard deviation, and variance of the␣
↪logarithmic returns.
mean = np.mean(train_log_returns)
mean = np.mean(train_log_returns)
stdev = np.std(train_log_returns)
var = np.var(train_log_returns)

# This line sets the time step ('dt') to 1 (in units of days) and calculates␣
↪the drift of the logarithmic returns using the mean and variance.
dt = 1
drift = (mean-(var/2))*dt
```

[12]:
```python
# We take the known last real stock price in the train period as the starting␣
↪point.
sim_prices = [0]*(len(test)+1) # we add 1 because in the following line we make␣
↪the first value the last value of the training set so we still need to␣
↪predict 250 more prices
sim_prices[0] = GS.loc[train_end, 'Adj Close']
```

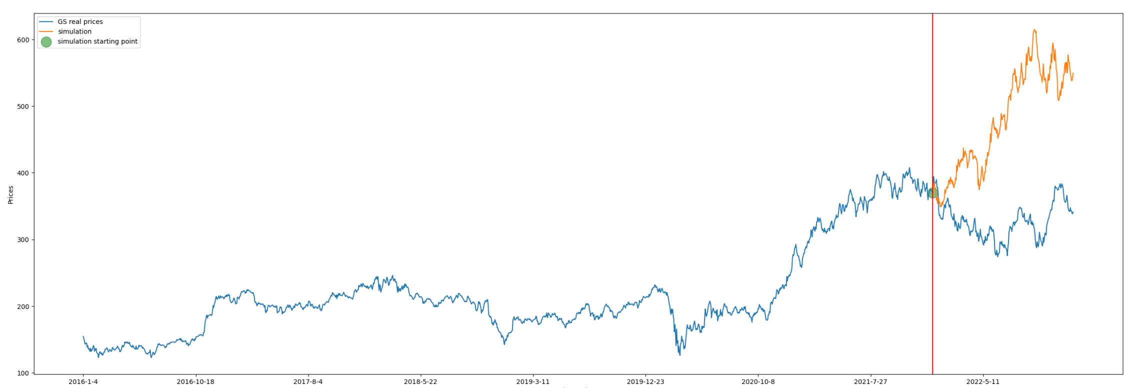
[13]:
```python
# create the simulated predicted prices

# Simulate stock prices using a geometric Brownian motion model
for i in range(1, len(sim_prices)):
    sim_prices[i] = sim_prices[i-1]*np.exp(drift + stdev*sqrt(dt)*np.random.
 ↪normal(0,1))
```

[14]:
```python
# obtaining the simulation range for the graph

# This line gets the index location of the 'train_end' date in the index of the␣
 ↪'GS' DataFrame, and assigns it to the 'test_start_x_axis' variable.
test_start_x_axis = GS.index.get_loc(train_end)

# This line creates a list called 'sim_range' that contains the indices of the␣
 ↪'GS' DataFrame from 'test_start_x_axis' onwards
sim_range = list(range(test_start_x_axis, test_start_x_axis + len(sim_prices)))
```

[14]:
```python

```

[15]:
```python
# plot the simulated prices against the real prices for GS

plt.figure(figsize=(30,10))
plt.plot(GS['Adj Close'], label='GS real prices')
plt.plot(sim_range, sim_prices, label='simulation')
plt.axvline(x=test_start_x_axis, color='r')
plt.scatter(x=test_start_x_axis, y=GS.iloc[test_start_x_axis]['Adj Close'],␣
 ↪c='g', s=250, alpha=0.5, label='simulation starting point')
plt.xticks(range(0,len(GS),200))
plt.xlabel('Time points')
plt.ylabel('Prices')
plt.legend()
```

[15]: <matplotlib.legend.Legend at 0x7f0c9f1c0970>



[15]:

```python

```

[15]:

```python

```

[15]:

```python

```

[15]:

```python

```

[16]:

```python
# obtain the log returns for the actual data for the test set

actual_log_return_test_data = test['log_ret'].reset_index()
actual_log_return_test_data = actual_log_return_test_data['log_ret'][1:] # we␣
↪do one onwards as the first predicted value is lost due to differencing
```

[17]:

```python
# calculate the logarithm of the simulated price
log_sim_prices = np.log(sim_prices[1:]) # we do it from 1 onwards because 0 was␣
↪the last value of the training set not a simulation

# calculate the differences between the logarithms of consecutive prices
sim_prices_log_returns = np.diff(log_sim_prices)
```

[18]:

```python
# calculate the mean squared error between the simulated log returns and the␣
↪actual log returns for the test data
mse_log_returns = mean_squared_error(actual_log_return_test_data,␣
↪sim_prices_log_returns)
print("Mean Squared Error of Log Returns:", mse_log_returns)
```

```
Mean Squared Error of Log Returns: 0.0006485676169971517
```

[19]:

```python
actual_prices_test_data = test['Adj Close'].reset_index()
actual_prices_test_data = actual_prices_test_data['Adj Close']
```

[20]:

```python
mse_prices = mean_squared_error(actual_prices_test_data, sim_prices[1:],)␣
↪# again here we have done sim prices from 1 onwards as we are not counting␣
↪the end of training set value
print("Mean Squared Error of Prices:", mse_prices)
```

```
Mean Squared Error of Prices: 29515.49417913668
```

[21]:

```python
# this is finding the sample variance of the time series
sample_var = np.var(test, ddof=1)
print("Sample variance of test set:", sample_var)
```

```
Sample variance of test set: Adj Close    746.893007
log_price      0.006868
log_ret        0.000349
dtype: float64
```

[22]:
```
# We can improve our GBM simulations by taking many GBM simulations and taking␣
↪the mean at each time point to obtain one mean path for the GBM this will␣
↪reduce the volatility present in any
# one predicted path
```

[23]:
```
# Create a 2-dimensional numpy array called 'sim_prices_multi' with shape␣
↪(len(test),100) filled with zeros
# This array will be used to store the simulated prices for multiple paths
sim_prices_multi = np.zeros(shape=(len(test)+1,100)) # again we are adding one␣
↪because we are making the first value the value at the end of the train set

# Set the initial price for each path in 'sim_prices_multi' to be the closing␣
↪price of the stock on the last day of the training period
# The iloc method is used to select a specific row and column in the 'GS'␣
↪DataFrame
sim_prices_multi[0,:] = GS.iloc[GS.index.get_loc(train_end),0]
```

[24]:
```
for i in range (0,100):
    for j in range (1,len(sim_prices_multi)):
        sim_prices_multi[j,i]=sim_prices_multi[j-1,i]*np.exp(drift +␣
↪stdev*sqrt(dt)*np.random.normal(0,1))
```

[25]:
```
sim_prices_multi=pd.DataFrame(sim_prices_multi)
sim_prices_multi
```

[25]:
```
              0           1           2           3           4           5  \
0    370.066895  370.066895  370.066895  370.066895  370.066895  370.066895
1    370.007646  364.759091  370.597632  373.524557  365.669882  368.519305
2    378.268757  376.098933  373.905348  374.377321  379.867174  364.013597
3    373.700111  377.981661  370.613442  381.501524  379.346144  364.996652
4    367.283770  373.644373  366.256880  380.358160  380.116149  374.215993
..          …           …           …           …           …           …
246  532.134717  343.177944  314.210462  443.853731  404.549757  319.704958
247  522.134522  354.121433  319.667800  444.948913  398.959226  322.340834
248  536.091128  351.013210  317.474319  437.669327  406.460162  321.191831
249  531.520931  344.466082  320.211977  441.833776  383.344762  323.165264
250  519.615112  343.650352  327.248194  446.649989  389.264538  310.948853

              6           7           8           9  …          90  \
0    370.066895  370.066895  370.066895  370.066895  …  370.066895
1    374.549760  368.447867  376.076042  364.653452  …  363.737828
2    370.812501  365.747029  380.878910  370.070911  …  348.338561
3    355.112999  366.584492  383.966148  361.488696  …  346.494607
4    356.610645  370.984618  390.354317  362.773936  …  347.559398
..          …           …           …           …  …           …
246  363.134671  482.883792  316.565131  365.561194  …  176.791804
247  359.605074  484.845523  316.664679  372.158942  …  177.429070
```

```
248  351.473985  487.397197  304.693413  379.123657  …  178.745374
249  349.645972  482.179200  305.865205  384.026345  …  175.620387
250  348.494531  483.557836  314.176059  385.900371  …  174.077157

             91          92          93          94          95          96  \
0    370.066895  370.066895  370.066895  370.066895  370.066895  370.066895
1    383.851093  363.402845  370.384090  384.489788  372.645060  370.823134
2    386.972565  383.647376  369.991978  385.909491  373.047322  367.408325
3    388.325083  386.766787  362.896820  392.799128  367.683633  358.648254
4    394.878646  403.167158  374.392162  391.940022  374.814061  367.689598
..          …           …           …           …           …           …
246  350.507398  620.247076  770.092733  393.035982  542.345571  364.958680
247  352.931256  628.653002  791.646522  388.896818  516.180462  367.202285
248  354.910354  622.191219  777.118914  398.186538  532.201571  374.107957
249  353.454020  624.315796  788.221298  396.431722  544.177907  374.715957
250  344.558903  617.548032  749.862318  395.966279  534.088178  391.212630

             97          98          99
0    370.066895  370.066895  370.066895
1    367.845632  362.553737  361.683056
2    380.053377  364.626899  360.334536
3    364.941303  363.343289  361.262756
4    374.213505  367.626809  366.745061
..          …           …           …
246  593.839545  345.091319  387.542755
247  592.488106  352.381357  386.767437
248  611.672830  346.615927  396.126295
249  619.054074  352.572595  398.139886
250  626.510054  347.864695  404.770408

[251 rows x 100 columns]
```

[26]:
```
# Calculate the mean of the simulated prices for each day across all paths in the 'sim_prices_multi'
multi_simulated_prices_mean=sim_prices_multi.mean(axis=1) # axis=0 means "column", axis=1 means "row"
multi_simulated_prices_mean
```

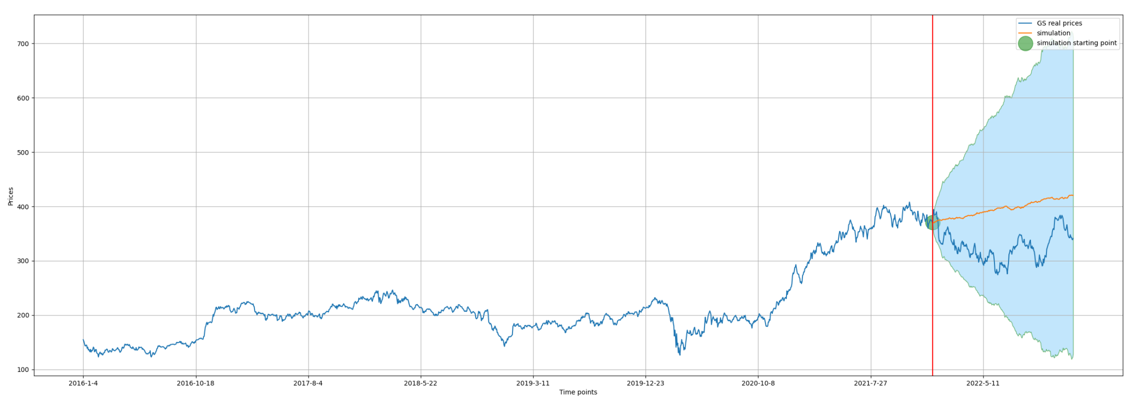
[26]:
```
0      370.066895
1      369.646253
2      370.071481
3      369.423552
4      371.682578
          …
246    420.243271
247    420.717329
248    420.255135
```

```
249    420.486162
250    419.984591
Length: 251, dtype: float64
```

[27]:
```
# Calculate the lower bounds of the 95% confidence intervals for the simulated prices for each day across all paths in the 'sim_prices_multi'
lower=multi_simulated_prices_mean - 2*sim_prices_multi.std(axis=1)
upper=multi_simulated_prices_mean + 2*sim_prices_multi.std(axis=1)
```

[28]:
```
# create graph of the average GBM path

plt.figure(figsize=(30,10))
plt.plot(GS['Adj Close'],label='GS real prices')
plt.plot(sim_range,multi_simulated_prices_mean,label='simulation')
plt.axvline(x=test_start_x_axis, color='r')
plt.scatter(x=test_start_x_axis, y=GS.iloc[test_start_x_axis]['Adj Close'], c='g', s=500, alpha=0.5, label='simulation starting point')
plt.xticks(range(0,len(GS),200))
plt.xlabel('Time points')
plt.ylabel('Prices')
plt.grid(True)
# We need to add one row code to fill the band with some color
plt.fill_between(sim_range, lower, upper, color='LightSkyBlue',edgecolor='g',alpha=0.5)
plt.legend()

# You can see, now our simulation is much more stable as we take the average of multiple simulations
```

[28]:
```
<matplotlib.legend.Legend at 0x7f0c9d63bfd0>
```



[29]:
```
# calculate the logarithm of the mean simulated price
multi_simulated_log_prices = np.log(multi_simulated_prices_mean)
```

```
# calculate the differences between the logarithms of consecutive prices
multi_simulated_log_returns = np.diff(multi_simulated_log_prices, axis=0)
```

[29]:

```

```

[29]:

```

```

[29]:

```

```

[29]:

```

```

[30]:

```
# Calculate MSE for GBM multiple simulation log returns
mse_log_returns = mean_squared_error(actual_log_return_test_data,␣
↪multi_simulated_log_returns[1:])
print("Mean Squared Error of multiple simulation Log Returns:", mse_log_returns)
```

```
Mean Squared Error of multiple simulation Log Returns: 0.0003593816806948411
```

[31]:

```
predicted_prices_GBM = multi_simulated_prices_mean.to_numpy()
```

[32]:

```
# Calculate MSE for GBM multiple simulation prices
mse_multi_prices = mean_squared_error(actual_prices_test_data,␣
↪predicted_prices_GBM[1:])
print("Mean Squared Error of multiple simulation prices:", mse_multi_prices)
```

```
Mean Squared Error of multiple simulation prices: 5631.392633690118
```

[33]:

```
# The following code runs the exact same code as above to simulate the MSE a␣
↪number of times in order to obtain the average mse from a both the single␣
↪and multi GBM simulations

# initialize empty data frames to store MSE values for each simulation
mse_sim_prices_list = []
mse_sim_log_returns_list = []
mse_multi_sim_prices_list = []
mse_multi_sim_log_returns_list = []
# set the number of simulations to run
num_simulations = 100
for i in range(num_simulations):
  import datetime as dt
  yf.pdr_override()
  start = dt.datetime(2016,1,4)
  end = dt.datetime(2022,12,30)
  GS = pdr.get_data_yahoo('GS', start, end, interval='1d')
  GS.index = GS.index.tz_localize(None)
  # convert time-zone-aware date-time values to time-zone-naive date-time values
```

```
GS.index = GS.index.tz_localize(None)
GS = GS['Adj Close']
GS = pd.DataFrame(GS)
GS['Year'] = GS.index.year.astype(str)
GS['Month'] = GS.index.month.astype(str)
GS['Day'] = GS.index.day.astype(str)
GS.index = GS['Year'].str.cat(GS[['Month','Day']], sep='-')

GS.drop(['Year', 'Month', 'Day'], axis=1, inplace=True)

# Define date ranges
train_start = '2016-1-4'
train_end = '2021-12-31'
test_start = '2022-1-3'
train = GS[:train_end] # all the data points up to and including 2021-12-31
test = GS[test_start:] # all the data points after 2022-1-3
logret = [0]*len(GS[train_start:train_end])
for i in range(len(train)):
    logret[i] = np.log(GS.iloc[i+1]/GS.iloc[i])
mean = np.mean(logret)
stdev = np.std(logret)
var = np.var(logret)
dt = 1
drift = (mean-(var/2))*dt
# We take the known last real stock price in the 9-month period as the␣
↪starting point.
sim_prices = [0]*(len(test))
sim_prices[0] = GS.loc[train_end, 'Adj Close']
# create the simulated predicted prices
# Simulate stock prices using a geometric Brownian motion model
for i in range(1, len(sim_prices)):
    sim_prices[i] = sim_prices[i-1]*np.exp(drift + stdev*sqrt(dt)*np.random.
↪normal(0,1))
# calculate the logarithm of the real prices
GS['log_price'] = np.log(GS['Adj Close'])
# find the log return
GS['log_ret'] = GS['log_price'].diff()
# Now we re-define train and test to include the log prices and log returns
train = GS[:train_end] # all the data points up to and including 2021-12-31
test = GS[test_start:] # all the data points after 2022-1-3
actual_log_return_test_data = test['log_ret'].reset_index()
actual_log_return_test_data = actual_log_return_test_data['log_ret'][1:]
# calculate the logarithm of the mean simulated price
log_sim_prices = np.log(sim_prices)
# calculate the differences between the logarithms of consecutive prices
sim_prices_log_returns = np.diff(log_sim_prices, axis=0)
```

```
  mse_single_sim_log_returns = np.mean((sim_prices_log_returns -␣
 ↪actual_log_return_test_data)**2)
  actual_prices_test_data = test['Adj Close'].reset_index()
  actual_prices_test_data = actual_prices_test_data['Adj Close']
  mse_single_sim_prices = np.mean((sim_prices - actual_prices_test_data)**2)
  # generates a simulation of stock prices for a financial asset over predicted␣
 ↪time periods for 100 simulations.
  sim_prices_multi = np.zeros(shape=(len(test),100))
  sim_prices_multi[0,:] = GS.iloc[GS.index.get_loc(train_end),0]
  for i in range (0,100):
      for j in range (1,len(sim_prices_multi)):
          sim_prices_multi[j,i]=sim_prices_multi[j-1,i]*np.exp(drift +␣
 ↪stdev*sqrt(dt)*np.random.normal(0,1))
  sim_prices_multi=pd.DataFrame(sim_prices_multi)
  multi_simulated_prices_mean=sim_prices_multi.mean(axis=1) # axis=0 means␣
 ↪"column", axis=1 means "row"
  # You can see, now our simulation is much more stable as we take the average␣
 ↪of multiple simulations
  # calculate the logarithm of the mean simulated price
  multi_simulated_log_prices = np.log(multi_simulated_prices_mean)
  # calculate the differences between the logarithms of consecutive prices
  multi_simulated_log_returns = np.diff(multi_simulated_log_prices, axis=0)
  # Calculate MSE
  mse_multi_sim_log_returns = np.mean((actual_log_return_test_data -␣
 ↪multi_simulated_log_returns)**2)
  predicted_prices_GBM = multi_simulated_prices_mean.to_numpy()
  mse__multi_sim_prices = np.mean((predicted_prices_GBM -␣
 ↪actual_prices_test_data)**2)
  mse_sim_prices_list.append(mse_single_sim_prices)
  mse_sim_log_returns_list.append(mse_single_sim_log_returns)
  mse_multi_sim_prices_list.append(mse__multi_sim_prices)
  mse_multi_sim_log_returns_list.append(mse_multi_sim_log_returns)

mse_sim_log_returns_list
mean_mse_sim_log_returns_list = np.mean(mse_sim_log_returns_list)
print('Mean mse of single simulation log returns',␣
 ↪mean_mse_sim_log_returns_list)

mse_sim_prices_list
mean_mse_sim_prices_list = np.mean(mse_sim_prices_list)
print('Mean mse of single simulation prices', mean_mse_sim_prices_list)

mse_multi_sim_log_returns_list
mean_mse_multi_sim_log_returns_list = np.mean(mse_multi_sim_log_returns_list)
print('Mean mse of multiple simulation log returns',␣
 ↪mean_mse_multi_sim_log_returns_list)
```

```python
mse_multi_sim_prices_list
mean_mse_multi_sim_prices_list = np.mean(mse_multi_sim_prices_list)
print('Mean mse of multiple simulation prices', mean_mse_multi_sim_prices_list)
```

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

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

```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Mean mse of single simulation log returns 0.0007233461345460477
Mean mse of single simulation prices 14918.20782783372
Mean mse of multiple simulation log returns 0.00035026070432525336
Mean mse of multiple simulation prices 6375.732074444002
```

2.0.1 ARIMA model

[33]:

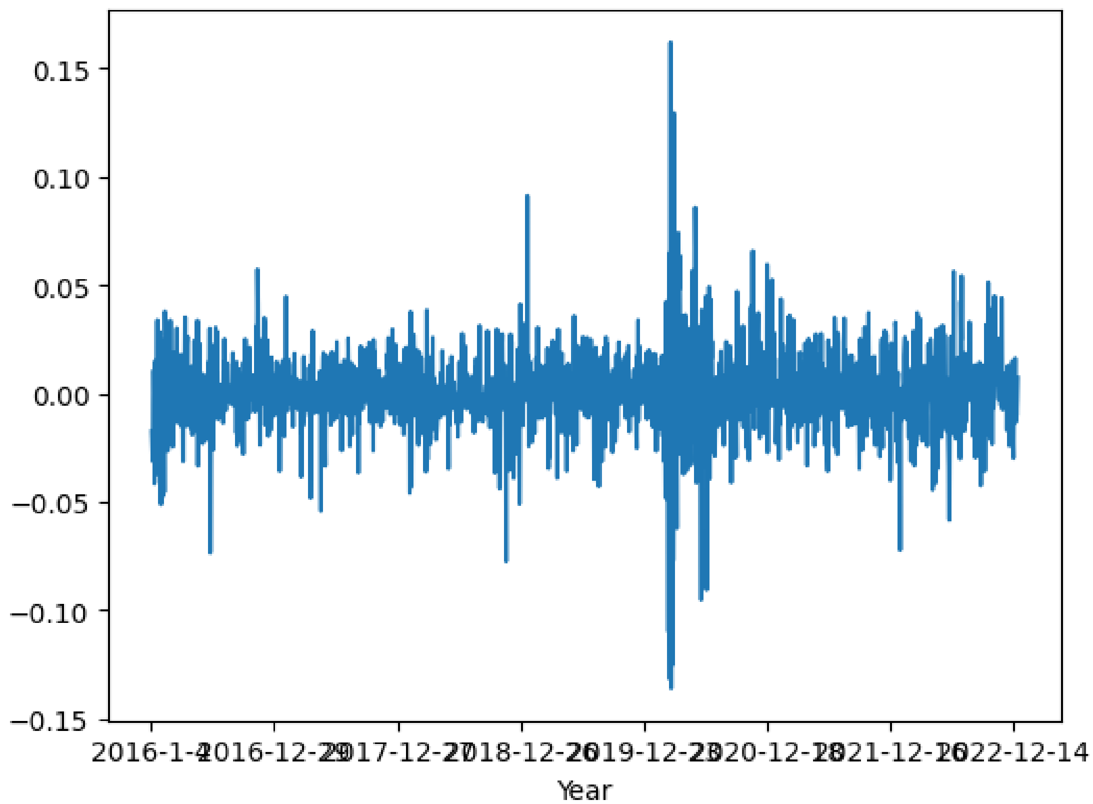
```
```

[34]:

```
GS['log_ret'].plot()
```

[34]:

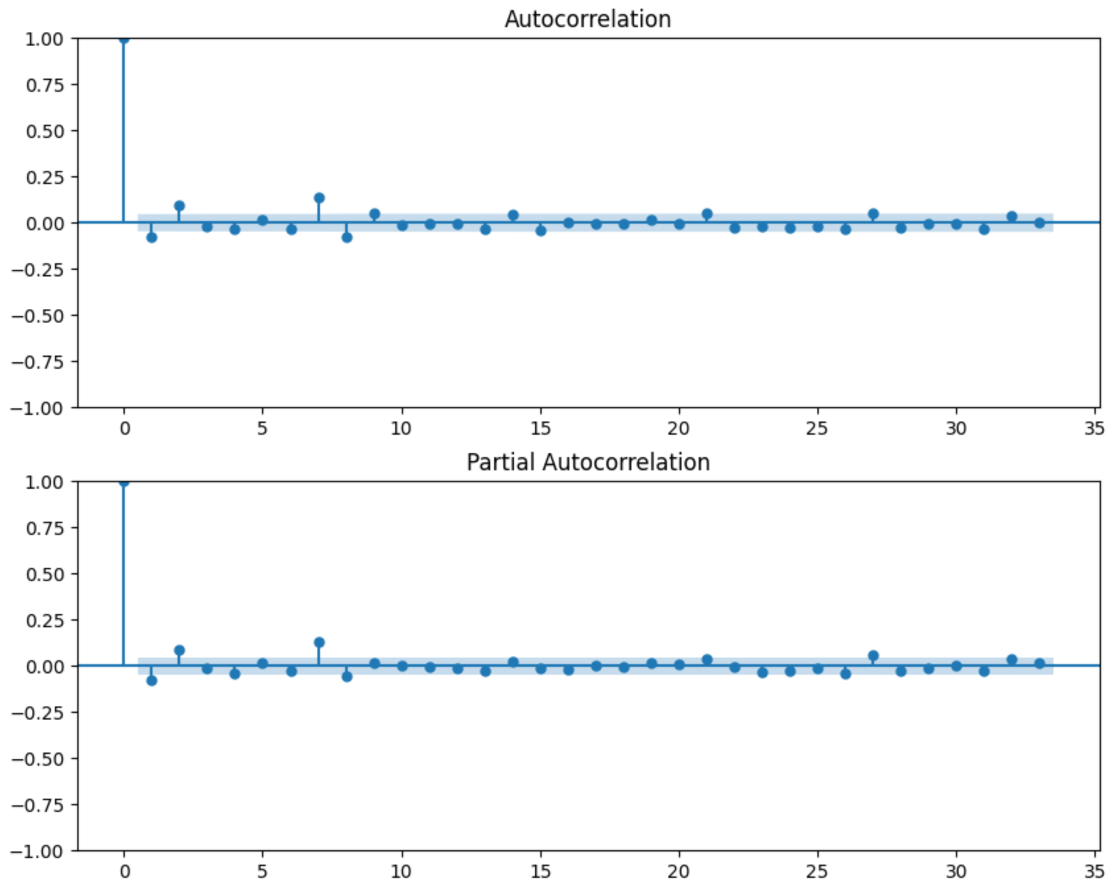
```
<Axes: xlabel='Year'>
```



[35]:
```
# We want to check for auto correlation in our model:
# testing for stationarity

adf(GS['log_ret'].dropna())
```

[35]:
```
(-14.252089082761698,
 1.4751057077575547e-26,
 7,
 1752,
 {'1%': -3.4340879605755426,
  '5%': -2.8631911014332876,
  '10%': -2.567648997323346},
 -8837.191922488015)
```

[36]:
```
# these results show that returns are stationary. therefore d=1 for the ARIMA

# Now we want to find what p and q are for the ARIMA which can be found through␣
↪pacf and acf respectively

fig, axes = plt.subplots(2,1,figsize = (10,8))
plot_acf(GS['log_ret'].dropna(), ax = axes[0])
plot_pacf(GS['log_ret'].dropna(), ax = axes[1])
plt.show()
```



[37]:
```
# These graphs show that lags up to 2 are outside the confidence interval and
↪so can be used in the ARIMA model however for the purpose of this question
↪we are going to use an ARIMA of order (111)
```

[38]:
```
# We use log prices because it helps reduce trends and assist in stationarity
↪when we difference the data in the ARIMA model which is a property required
↪for such a model

arima = ARIMA(train['log_price'], order=(1,1,1))
arima_result = arima.fit()
```

```
/usr/local/lib/python3.9/dist-packages/statsmodels/tsa/base/tsa_model.py:471:
ValueWarning: A date index has been provided, but it has no associated frequency
information and so will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
/usr/local/lib/python3.9/dist-packages/statsmodels/tsa/base/tsa_model.py:471:
ValueWarning: A date index has been provided, but it has no associated frequency
information and so will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
```

```
/usr/local/lib/python3.9/dist-packages/statsmodels/tsa/base/tsa_model.py:471:
ValueWarning: A date index has been provided, but it has no associated frequency
information and so will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
```

[39]:
```
# Obtain the predicted values for the training set from the ARIMA model
# The 'fittedvalues' attribute is called on the 'arima_result' object to obtain␣
 ↪the predicted values for the training set. Although train_pred holds all the␣
 ↪fitted values we shall restrict it later to the required range (e.g when␣
 ↪plotting)
training_pred = arima_result.fittedvalues
```

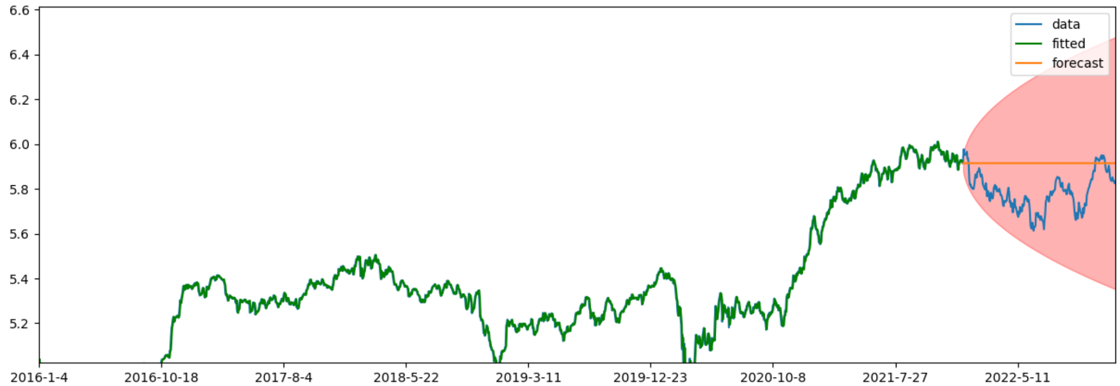
[40]:
```
prediction_result = arima_result.get_forecast(len(test))
```

```
/usr/local/lib/python3.9/dist-packages/statsmodels/tsa/base/tsa_model.py:834:
ValueWarning: No supported index is available. Prediction results will be given
with an integer index beginning at `start`.
  return get_prediction_index(
```

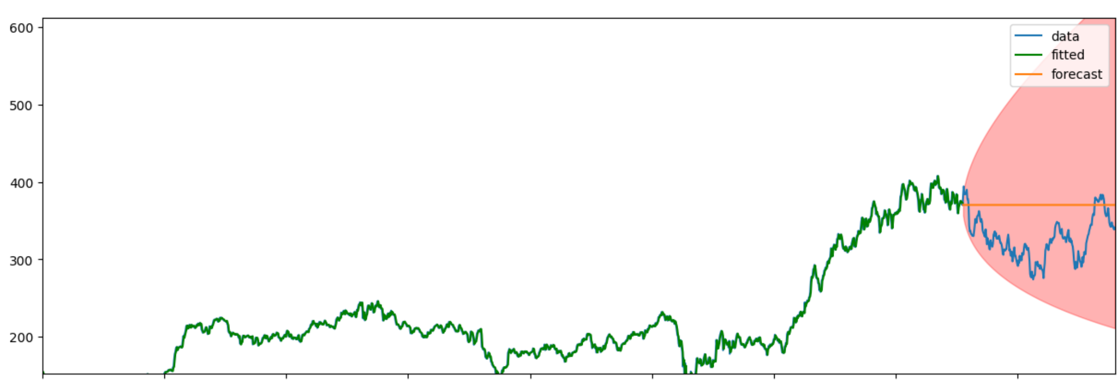
[41]:
```
# Confidence interval
conf_inte = prediction_result.conf_int()
```

[41]:

[42]:
```
# Plot the graph of the log prices

fig, ax = plt.subplots(figsize=(15, 5))
lower = conf_inte['lower log_price']
upper = conf_inte['upper log_price']
forecast = prediction_result.predicted_mean
ax.plot(GS['log_price'], label='data')
ax.plot(train.index, training_pred, color='green', label='fitted')
ax.plot(test.index, forecast, label='forecast')
ax.fill_between(test.index, lower, upper, color='red', alpha=0.3)
ax.set_xlim(test.index[0], test.index[-1])
max_log_price = GS['log_price'].max()
ax.set_ylim(GS['log_price'][1], 1.1*max_log_price)
plt.xticks(range(0,len(GS),200))
ax.legend()

plt.show()
```



[43]:
```
fig, ax = plt.subplots(figsize=(15, 5))

# Transform log prices to regular prices
train_price = np.exp(train['log_price'])
fitted_price = np.exp(training_pred)
forecast_price = np.exp(forecast)
lower_price = np.exp(lower)
upper_price = np.exp(upper)

# Plot regular prices
ax.plot(GS['Adj Close'], label='data')
ax.plot(train_price.index, fitted_price, color='green', label='fitted')
ax.plot(test.index, forecast_price, label='forecast')
ax.fill_between(test.index, lower_price, upper_price, color='red', alpha=0.3)
ax.set_xlim(test.index[0], test.index[-1])
max_price = GS['Adj Close'].max()
ax.set_ylim(GS['Adj Close'][1], 1.5*max_price)
plt.xticks(range(0,len(GS),200))
ax.legend()

plt.show()
```



[44]:
```python
# Calculate the log returns for the test set
predicted_log_returns = np.diff(forecast)
```

[44]:
```python

```

[44]:
```python

```

[45]:
```python
# calculate the mean squared error of log returns for ARIMA
mse_log_returns = mean_squared_error(actual_log_return_test_data,␣
 ↪predicted_log_returns)
print("Mean Squared Error of ARIMA log returns:", mse_log_returns)
```

```
Mean Squared Error of ARIMA log returns: 0.0003445580036908478
```

[45]:
```python

```

[45]:
```python

```

[46]:
```python
# Turn the price forecasts into a numpy array to make mse calculation esier
Arima_predicted_prices = forecast_price.to_numpy()
```

[47]:
```python
# calculate the mean squared error of prices for ARIMA
mse_prices = mean_squared_error(actual_prices_test_data, Arima_predicted_prices)
print("Mean Squared Error of ARIMA prices:", mse_prices)
```

```
Mean Squared Error of ARIMA prices: 2613.4157809427156
```

[48]:
```python
# We dont need to run the ARIMA multiple times to find the average MSE becasue
# the ARIMA model is a deterministic model, meaning that it produces the same␣
 ↪results for the same input data and parameters every time.
# Therefore, if you fit the ARIMA model once to a given dataset, the same model␣
 ↪will produce the same forecast every time it is used to predict future␣
 ↪values.
# This means that you do not need to run the ARIMA model multiple times to find␣
 ↪an average forecast, as the same forecast will be produced each time.
```

2.0.2 ANN model FFNN

Feed Foward Neural Network

[49]:
```python
# Turn GS index into a datetime to make splitting the data easier in the␣
 ↪following code
GS.index = pd.to_datetime(GS.index)
```

[50]:
```
# initialise the StandardScaler object, which will be used to normalize the␣
↪data.
scaler = StandardScaler()

# normalize the training and testing data using the StandardScaler object.
# In this FFNN we use log returns as this provides stationarity which will␣
↪improve the training of the FFNN as it is not designed specifically for time␣
↪series
# Therefore, we try and improve upon neural network predictions by in the next␣
↪section using LSTM as LSTM is a neural network specifically designed for␣
↪time series data and hence we directly predict prices in line with the␣
↪previous models
train_scaled = scaler.fit_transform(train[['log_ret']])
test_scaled = scaler.transform(test[['log_ret']])
```

[51]:
```
# create boolean series that will be used to index the training and testing␣
↪data in the dataframe.
train_idx = GS.index <= train.index[-1]
test_idx = GS.index > train.index[-1]
```

[52]:
```
# assign the normalized training and testing data to a new column called␣
↪'Log_return_scaled' in the dataframe.

GS.loc[train_idx, 'Log_return_scaled'] = train_scaled.flatten()

GS.loc[test_idx, 'Log_return_scaled'] = test_scaled.flatten()
```

[53]:
```
# supervised dataset creation
# convert the 'Log_return_scaled' column of the dataframe to a numpy array,␣
↪with NaN values removed.
series = GS['Log_return_scaled'].dropna().to_numpy() # contains the data␣
↪required later to make predictions



T = 10 # windows size
X = [] # initialize empty lists X and Y for the input and output data in the␣
↪prediction.
Y = []
# This loop extracts a window of size T from the series array, and appends the␣
↪input/output pairs to the X and Y lists.
for t in range(len(series) - T):
  x = series[t:t+T] #This line extracts a window of T time steps from the␣
↪series array, starting at time step t and ending at time step t+T-1. This␣
↪window T will be used as input to the prediction model.
  X.append(x) # This line appends the x array to the X list, which will store␣
↪all of the input data.
```

```
  y = series[t+T] # This line extracts the value of the time series at time␣
↪step t+T, which will be used as the output value for the prediction model.
  Y.append(y) # This line appends the output value y to the Y list, which will␣
↪store all of the output data.

X = np.array(X).reshape(-1, T)# The reshape() method is used to ensure that␣
↪each input/output pair has T time steps.
Y = np.array(Y)
N = len(X) # set N to be the number of input/output pairs in the data.
print("X.shape", X.shape, "Y.shape", Y.shape) # prints the shapes of the input␣
↪and output data.
```

```
X.shape (1750, 10) Y.shape (1750,)
```

[54]:
```
# split the data into training and testing sets.
Xtrain, Ytrain = X[:-len(test)], Y[:-len(test)]
Xtest, Ytest = X[-len(test):], Y[-len(test):]
```

[55]:
```
# create and compile the FFNN model using the Keras API, and train it on the␣
↪training data.
i = Input(shape=(T,))
Layer = Dense(32, activation='relu')(i)
Layer = Dense(1)(Layer)

FFNN = Model(i, Layer)
```

[56]:
```
# Compile the FFNN model by specifying the optimizer and the loss function
FFNN.compile(loss='mse',optimizer='adam',)
```

[57]:
```
# Fit the FFNN model to the training data, specifying the input and output␣
↪data, number of epochs, and validation data
r = FFNN.fit(
  Xtrain,
  Ytrain,
  epochs=200,
  validation_data=(Xtest, Ytest)
)
```

```
Epoch 1/200
47/47 [==============================] - 1s 6ms/step - loss: 1.4657 - val_loss:
1.2923
Epoch 2/200
47/47 [==============================] - 0s 3ms/step - loss: 1.1249 - val_loss:
1.1574
Epoch 3/200
47/47 [==============================] - 0s 3ms/step - loss: 1.0212 - val_loss:
1.1165
```

```
Epoch 4/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9752 - val_loss:
1.0982
Epoch 5/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9454 - val_loss:
1.0966
Epoch 6/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9259 - val_loss:
1.0903
Epoch 7/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9124 - val_loss:
1.0976
Epoch 8/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8991 - val_loss:
1.1005
Epoch 9/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8901 - val_loss:
1.1051
Epoch 10/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8842 - val_loss:
1.1069
Epoch 11/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8759 - val_loss:
1.1051
Epoch 12/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8688 - val_loss:
1.1095
Epoch 13/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8628 - val_loss:
1.1070
Epoch 14/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8572 - val_loss:
1.1113
Epoch 15/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8520 - val_loss:
1.1108
Epoch 16/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8478 - val_loss:
1.1130
Epoch 17/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8448 - val_loss:
1.1156
Epoch 18/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8400 - val_loss:
1.1145
Epoch 19/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8355 - val_loss:
1.1150
```

```
Epoch 20/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8317 - val_loss:
1.1126
Epoch 21/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8305 - val_loss:
1.1193
Epoch 22/200
47/47 [==============================] - 0s 5ms/step - loss: 0.8273 - val_loss:
1.1183
Epoch 23/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8233 - val_loss:
1.1153
Epoch 24/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8206 - val_loss:
1.1135
Epoch 25/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8179 - val_loss:
1.1143
Epoch 26/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8152 - val_loss:
1.1160
Epoch 27/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8122 - val_loss:
1.1172
Epoch 28/200
47/47 [==============================] - 0s 4ms/step - loss: 0.8101 - val_loss:
1.1130
Epoch 29/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8070 - val_loss:
1.1125
Epoch 30/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8066 - val_loss:
1.1089
Epoch 31/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8005 - val_loss:
1.1157
Epoch 32/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8020 - val_loss:
1.1171
Epoch 33/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7985 - val_loss:
1.1143
Epoch 34/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7958 - val_loss:
1.1080
Epoch 35/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7944 - val_loss:
1.1185
```

```
Epoch 36/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7917 - val_loss:
1.1127
Epoch 37/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7882 - val_loss:
1.1154
Epoch 38/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7866 - val_loss:
1.1116
Epoch 39/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7846 - val_loss:
1.1164
Epoch 40/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7848 - val_loss:
1.1151
Epoch 41/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7829 - val_loss:
1.1114
Epoch 42/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7788 - val_loss:
1.1174
Epoch 43/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7760 - val_loss:
1.1185
Epoch 44/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7757 - val_loss:
1.1164
Epoch 45/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7733 - val_loss:
1.1120
Epoch 46/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7720 - val_loss:
1.1145
Epoch 47/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7684 - val_loss:
1.1143
Epoch 48/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7668 - val_loss:
1.1107
Epoch 49/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7649 - val_loss:
1.1097
Epoch 50/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7625 - val_loss:
1.1185
Epoch 51/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7607 - val_loss:
1.1143
```

```
Epoch 52/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7592 - val_loss:
1.1141
Epoch 53/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7578 - val_loss:
1.1200
Epoch 54/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7572 - val_loss:
1.1147
Epoch 55/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7540 - val_loss:
1.1173
Epoch 56/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7529 - val_loss:
1.1174
Epoch 57/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7508 - val_loss:
1.1187
Epoch 58/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7514 - val_loss:
1.1137
Epoch 59/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7491 - val_loss:
1.1259
Epoch 60/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7483 - val_loss:
1.1257
Epoch 61/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7442 - val_loss:
1.1148
Epoch 62/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7425 - val_loss:
1.1226
Epoch 63/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7415 - val_loss:
1.1245
Epoch 64/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7463 - val_loss:
1.1212
Epoch 65/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7405 - val_loss:
1.1277
Epoch 66/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7363 - val_loss:
1.1269
Epoch 67/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7384 - val_loss:
1.1343
```

```
Epoch 68/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7332 - val_loss:
1.1278
Epoch 69/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7343 - val_loss:
1.1321
Epoch 70/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7302 - val_loss:
1.1263
Epoch 71/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7278 - val_loss:
1.1295
Epoch 72/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7271 - val_loss:
1.1324
Epoch 73/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7279 - val_loss:
1.1349
Epoch 74/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7250 - val_loss:
1.1320
Epoch 75/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7232 - val_loss:
1.1319
Epoch 76/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7225 - val_loss:
1.1309
Epoch 77/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7192 - val_loss:
1.1360
Epoch 78/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7219 - val_loss:
1.1367
Epoch 79/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7167 - val_loss:
1.1371
Epoch 80/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7158 - val_loss:
1.1370
Epoch 81/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7171 - val_loss:
1.1378
Epoch 82/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7166 - val_loss:
1.1373
Epoch 83/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7117 - val_loss:
1.1412
```

```
Epoch 84/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7135 - val_loss:
1.1456
Epoch 85/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7112 - val_loss:
1.1396
Epoch 86/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7104 - val_loss:
1.1412
Epoch 87/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7097 - val_loss:
1.1437
Epoch 88/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7083 - val_loss:
1.1483
Epoch 89/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7068 - val_loss:
1.1492
Epoch 90/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7058 - val_loss:
1.1424
Epoch 91/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7042 - val_loss:
1.1462
Epoch 92/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7047 - val_loss:
1.1547
Epoch 93/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7031 - val_loss:
1.1489
Epoch 94/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7017 - val_loss:
1.1475
Epoch 95/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7019 - val_loss:
1.1458
Epoch 96/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7021 - val_loss:
1.1502
Epoch 97/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6984 - val_loss:
1.1504
Epoch 98/200
47/47 [==============================] - 0s 5ms/step - loss: 0.6995 - val_loss:
1.1436
Epoch 99/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6986 - val_loss:
1.1455
```

```
Epoch 100/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6957 - val_loss:
1.1469
Epoch 101/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6941 - val_loss:
1.1516
Epoch 102/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6974 - val_loss:
1.1474
Epoch 103/200
47/47 [==============================] - 1s 13ms/step - loss: 0.6923 - val_loss:
1.1527
Epoch 104/200
47/47 [==============================] - 1s 23ms/step - loss: 0.6921 - val_loss:
1.1496
Epoch 105/200
47/47 [==============================] - 0s 8ms/step - loss: 0.6900 - val_loss:
1.1460
Epoch 106/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6880 - val_loss:
1.1475
Epoch 107/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6876 - val_loss:
1.1538
Epoch 108/200
47/47 [==============================] - 0s 9ms/step - loss: 0.6874 - val_loss:
1.1433
Epoch 109/200
47/47 [==============================] - 0s 10ms/step - loss: 0.6878 - val_loss:
1.1441
Epoch 110/200
47/47 [==============================] - 0s 6ms/step - loss: 0.6847 - val_loss:
1.1442
Epoch 111/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6835 - val_loss:
1.1434
Epoch 112/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6826 - val_loss:
1.1500
Epoch 113/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6837 - val_loss:
1.1474
Epoch 114/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6824 - val_loss:
1.1550
Epoch 115/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6809 - val_loss:
1.1435
```

```
Epoch 116/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6817 - val_loss:
1.1496
Epoch 117/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6792 - val_loss:
1.1499
Epoch 118/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6793 - val_loss:
1.1489
Epoch 119/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6779 - val_loss:
1.1456
Epoch 120/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6796 - val_loss:
1.1519
Epoch 121/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6784 - val_loss:
1.1533
Epoch 122/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6748 - val_loss:
1.1520
Epoch 123/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6756 - val_loss:
1.1538
Epoch 124/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6736 - val_loss:
1.1458
Epoch 125/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6766 - val_loss:
1.1516
Epoch 126/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6744 - val_loss:
1.1452
Epoch 127/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6726 - val_loss:
1.1604
Epoch 128/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6703 - val_loss:
1.1567
Epoch 129/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6706 - val_loss:
1.1538
Epoch 130/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6691 - val_loss:
1.1476
Epoch 131/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6683 - val_loss:
1.1557
```

```
Epoch 132/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6668 - val_loss:
1.1532
Epoch 133/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6648 - val_loss:
1.1529
Epoch 134/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6645 - val_loss:
1.1530
Epoch 135/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6633 - val_loss:
1.1541
Epoch 136/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6637 - val_loss:
1.1597
Epoch 137/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6631 - val_loss:
1.1583
Epoch 138/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6620 - val_loss:
1.1582
Epoch 139/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6610 - val_loss:
1.1514
Epoch 140/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6615 - val_loss:
1.1660
Epoch 141/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6635 - val_loss:
1.1596
Epoch 142/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6584 - val_loss:
1.1604
Epoch 143/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6587 - val_loss:
1.1615
Epoch 144/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6575 - val_loss:
1.1572
Epoch 145/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6566 - val_loss:
1.1586
Epoch 146/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6554 - val_loss:
1.1557
Epoch 147/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6562 - val_loss:
1.1606
```

```
Epoch 148/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6562 - val_loss:
1.1628
Epoch 149/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6537 - val_loss:
1.1590
Epoch 150/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6556 - val_loss:
1.1669
Epoch 151/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6533 - val_loss:
1.1662
Epoch 152/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6566 - val_loss:
1.1722
Epoch 153/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6535 - val_loss:
1.1610
Epoch 154/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6495 - val_loss:
1.1602
Epoch 155/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6530 - val_loss:
1.1661
Epoch 156/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6490 - val_loss:
1.1730
Epoch 157/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6482 - val_loss:
1.1537
Epoch 158/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6512 - val_loss:
1.1705
Epoch 159/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6468 - val_loss:
1.1635
Epoch 160/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6478 - val_loss:
1.1684
Epoch 161/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6484 - val_loss:
1.1674
Epoch 162/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6467 - val_loss:
1.1640
Epoch 163/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6443 - val_loss:
1.1702
```

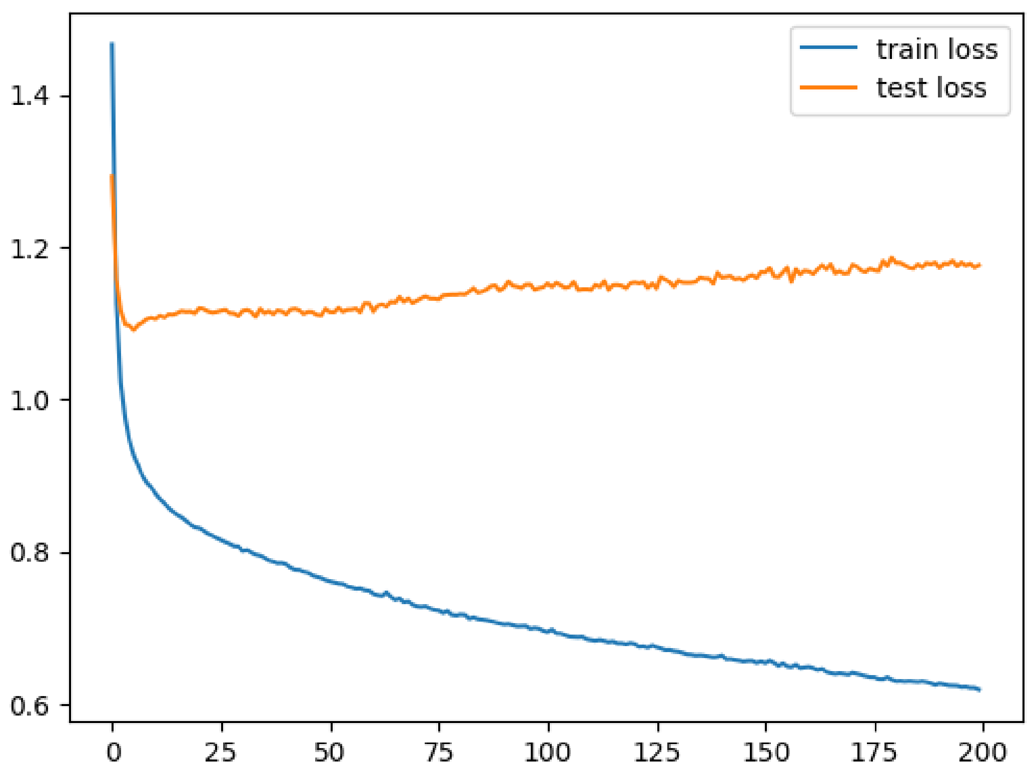
```
Epoch 164/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6458 - val_loss:
1.1758
Epoch 165/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6420 - val_loss:
1.1702
Epoch 166/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6402 - val_loss:
1.1769
Epoch 167/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6391 - val_loss:
1.1650
Epoch 168/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6400 - val_loss:
1.1681
Epoch 169/200
47/47 [==============================] - 0s 5ms/step - loss: 0.6392 - val_loss:
1.1647
Epoch 170/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6381 - val_loss:
1.1653
Epoch 171/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6408 - val_loss:
1.1765
Epoch 172/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6393 - val_loss:
1.1739
Epoch 173/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6378 - val_loss:
1.1686
Epoch 174/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6361 - val_loss:
1.1668
Epoch 175/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6349 - val_loss:
1.1713
Epoch 176/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6347 - val_loss:
1.1691
Epoch 177/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6324 - val_loss:
1.1681
Epoch 178/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6321 - val_loss:
1.1813
Epoch 179/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6349 - val_loss:
1.1745
```

```
Epoch 180/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6315 - val_loss:
1.1856
Epoch 181/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6294 - val_loss:
1.1787
Epoch 182/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6298 - val_loss:
1.1788
Epoch 183/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6291 - val_loss:
1.1755
Epoch 184/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6296 - val_loss:
1.1724
Epoch 185/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6290 - val_loss:
1.1713
Epoch 186/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6286 - val_loss:
1.1768
Epoch 187/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6294 - val_loss:
1.1728
Epoch 188/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6285 - val_loss:
1.1784
Epoch 189/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6268 - val_loss:
1.1770
Epoch 190/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6246 - val_loss:
1.1793
Epoch 191/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6266 - val_loss:
1.1721
Epoch 192/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6253 - val_loss:
1.1779
Epoch 193/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6240 - val_loss:
1.1773
Epoch 194/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6237 - val_loss:
1.1815
Epoch 195/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6235 - val_loss:
1.1742
```

```
Epoch 196/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6219 - val_loss:
1.1794
Epoch 197/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6223 - val_loss:
1.1752
Epoch 198/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6209 - val_loss:
1.1776
Epoch 199/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6210 - val_loss:
1.1726
Epoch 200/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6187 - val_loss:
1.1757
```

[58]:

```
# plot the training loss
plt.plot(r.history['loss'], label='train loss')
# plot the test loss
plt.plot(r.history['val_loss'], label='test loss')
plt.legend();
```



[59]:
```
train_idx[:T+1] = False # first T+1 values are not predictable
```

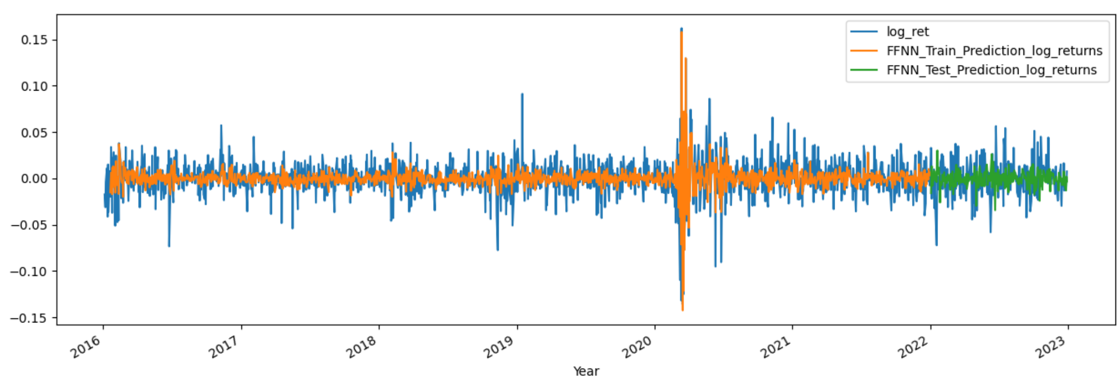
[60]:
```
# Use the FFNN model to predict the train and test data
predicted_train_data = FFNN.predict(Xtrain)
predicted_test_data = FFNN.predict(Xtest)

# Inverse transform the predicted data to get the actual values and flatten the␣
↪array
# Inverse transform is needed because the model was trained on scaled data
predicted_train_data = scaler.inverse_transform(predicted_train_data).flatten()
predicted_test_data = scaler.inverse_transform(predicted_test_data).flatten()
```

```
47/47 [==============================] - 0s 2ms/step
8/8 [==============================] - 0s 2ms/step
```

[61]:
```
# Store the predicted log returns in the FFNN_Train_Prediction_log_returns and␣
↪FFNN_Test_Prediction_log_returns columns of the dataframe.
GS.loc[train_idx, 'FFNN_Train_Prediction_log_returns'] = predicted_train_data

GS.loc[test_idx, 'FFNN_Test_Prediction_log_returns'] = predicted_test_data #␣
↪same way for test data
```

[62]:
```
# Plot the actual log returns, and the predicted log returns for the training␣
↪and testing data.
cols = ['log_ret',
        'FFNN_Train_Prediction_log_returns',
        'FFNN_Test_Prediction_log_returns']
GS[cols].plot(figsize=(15, 5));r
```

[62]:
```
<keras.callbacks.History at 0x7f0c9da0e610>
```

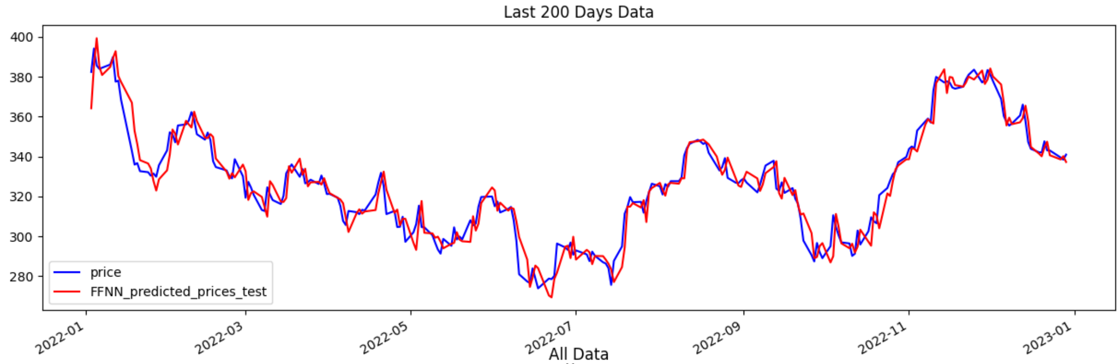


[63]:
```
# We shift the 'log_price' column by one to get the previous day's closing␣
↪price.
```

```
# Then we assign the shifted values to 'prev'.
# 'last_train' is the last training set value of 'log_price'.
# We add the predicted training data to 'prev' for the training set and assign␣
↪it to the '1step_train' column.
# Similarly, we add the predicted test data to 'prev' for the test set and␣
↪assign it to the '1step_test' column.

GS['Previous_log_price'] = GS['log_price'].shift(1)
previous_log_price = GS['Previous_log_price']
last_train = train.iloc[-1]['log_price']
GS.loc[train_idx, 'FFNN_predicted_log_prices_train'] =␣
↪previous_log_price[train_idx] + predicted_train_data
GS.loc[test_idx, 'FFNN_predicted_log_prices_test'] =␣
↪previous_log_price[test_idx] + predicted_test_data
```

[64]:

```
# convert log prices to normal prices
GS['price'] = np.exp(GS['log_price'])
GS['FFNN_predicted_prices_train'] = np.
↪exp(GS['FFNN_predicted_log_prices_train'])
GS['FFNN_predicted_prices_test'] = np.exp(GS['FFNN_predicted_log_prices_test'])


fig, axs = plt.subplots(2, 1, figsize=(15, 10), sharex=False)

# Plot the first graph which shows the last 250 days of the price data and the␣
↪1-step test predictions
GS.iloc[-250:][['price', 'FFNN_predicted_prices_test']].plot(ax=axs[0],␣
↪color=['blue', 'red'])
axs[0].set_title('Last 200 Days Data')

# Plot the second graph which shows all the price data, train and test␣
↪predictions
axs[1].plot(GS['price'], color = 'blue', label='price')
axs[1].plot(GS['FFNN_predicted_prices_train'],␣
↪label='FFNN_predicted_prices_train')
axs[1].plot(GS['FFNN_predicted_prices_test'], color = 'red',␣
↪label='FFNN_predicted_prices_test')
axs[1].legend()
axs[1].set_title('All Data')

plt.show()
```




```
[65]: FFNN_predicted_log_returns = GS['FFNN_Test_Prediction_log_returns'][test_idx].
      ↪to_numpy()
```

```
[66]: FNN_actual_log_return_test_data = test['log_ret'].reset_index() # As we have␣
      ↪directly predicted log returns we done lose a value to differencing so we␣
      ↪dont have to skip the first value in the actual log returns
      FNN_actual_log_return_test_data = FNN_actual_log_return_test_data['log_ret']
```

```
[67]: FFNN_mse_log_returns = mean_squared_error(FNN_actual_log_return_test_data,␣
      ↪FFNN_predicted_log_returns)
      print("Mean Squared Error of FFNN log returns:", FFNN_mse_log_returns)
```

```
Mean Squared Error of FFNN log returns: 0.0004428078815934635
```

```
[68]: FFNN_predicted_price = GS['FFNN_predicted_prices_test'][test_idx].to_numpy()
```

```
[69]: FFNN_mse_prices = mean_squared_error(actual_prices_test_data,␣
      ↪FFNN_predicted_price)
      print("Mean Squared Error of FFNN prices:", FFNN_mse_prices)
```

```
Mean Squared Error of FFNN prices: 45.67385671116432
```

```
[70]: # We now run the FFNN many times to get many MSE simulation so we can take the␣
      ↪average, again we do this be looping the above code
```

```python
mse_FFANN_sim_prices_list = []
mse_FFANN_sim_log_returns_list = []


# set the number of simulations to run
num_simulations = 20

for i in range(num_simulations):



  GS.index = pd.to_datetime(GS.index)
  scaler = StandardScaler()

  train_scaled = scaler.fit_transform(train[['log_ret']])
  test_scaled = scaler.transform(test[['log_ret']])

  # boolean series to index df rows
  train_idx = GS.index <= train.index[-1]
  test_idx = GS.index > train.index[-1]

  GS.loc[train_idx, 'Log_return_scaled'] = train_scaled.flatten()
  GS.loc[test_idx, 'Log_return_scaled'] = test_scaled.flatten()

  # Make supervised dataset
  series = GS['Log_return_scaled'].dropna().to_numpy()
  T = 10
  X = []
  Y = []
  for t in range(len(series) - T):
    x = series[t:t+T]
    X.append(x)
    y = series[t+T]
    Y.append(y)

  X = np.array(X).reshape(-1, T)
  Y = np.array(Y)
  N = len(X)
  Xtrain, Ytrain = X[:-len(test)], Y[:-len(test)]
  Xtest, Ytest = X[-len(test):], Y[-len(test):]
  i = Input(shape=(T,))
  Layer = Dense(32, activation='relu')(i)
  Layer = Dense(1)(Layer)
  FFNN = Model(i, Layer)
```

```
FFNN.compile(loss='mse',optimizer='adam',)

r = FFNN.fit(
  Xtrain,
  Ytrain,
  epochs=200,
  validation_data=(Xtest, Ytest)
)


train_idx[:T+1] = False

predicted_train_data = FFNN.predict(Xtrain)
predicted_test_data = FFNN.predict(Xtest)

predicted_train_data = scaler.inverse_transform(predicted_train_data).
↪flatten()
predicted_test_data = scaler.inverse_transform(predicted_test_data).flatten()
GS.loc[train_idx, 'FFNN_Train_Prediction_log_returns'] = predicted_train_data
GS.loc[test_idx, 'FFNN_Test_Prediction_log_returns'] = predicted_test_data

cols = ['log_ret',
        'FFNN_Train_Prediction_log_returns',
        'FFNN_Test_Prediction_log_returns']
# GS[cols].plot(figsize=(15, 5));r

GS['Previous_log_price'] = GS['log_price'].shift(1)
previous_log_price = GS['Previous_log_price']
last_train = train.iloc[-1]['log_price']
GS.loc[train_idx, 'FFNN_predicted_log_prices_train'] =␣
↪previous_log_price[train_idx] + predicted_train_data
GS.loc[test_idx, 'FFNN_predicted_log_prices_test'] =␣
↪previous_log_price[test_idx] + predicted_test_data

GS['price'] = np.exp(GS['log_price'])
GS['FFNN_predicted_prices_train'] = np.
↪exp(GS['FFNN_predicted_log_prices_train'])
GS['FFNN_predicted_prices_test'] = np.
↪exp(GS['FFNN_predicted_log_prices_test'])


FFNN_predicted_log_returns = GS['FFNN_Test_Prediction_log_returns'][test_idx].
↪to_numpy()
FNN_actual_log_return_test_data = test['log_ret'].reset_index()
FNN_actual_log_return_test_data = FNN_actual_log_return_test_data['log_ret']
```

```
    FFNN_mse_log_returns = mean_squared_error(FNN_actual_log_return_test_data,␣
 ↪FFNN_predicted_log_returns)


    FFNN_predicted_price = GS['FFNN_predicted_prices_test'][test_idx].to_numpy()
    FFNN_mse_prices = mean_squared_error(actual_prices_test_data,␣
 ↪FFNN_predicted_price)



    mse_FFANN_sim_log_returns_list.append(FFNN_mse_log_returns)
    mse_FFANN_sim_prices_list.append(FFNN_mse_prices)


print("MSE for FFNN log returns", np.mean(mse_FFANN_sim_log_returns_list))

print("MSE for FFNN prices", np.mean(mse_FFANN_sim_prices_list))
```

```
Epoch 1/200
47/47 [==============================] - 1s 6ms/step - loss: 1.1348 - val_loss:
1.0171
Epoch 2/200
47/47 [==============================] - 0s 3ms/step - loss: 1.0208 - val_loss:
1.0007
Epoch 3/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9860 - val_loss:
0.9998
Epoch 4/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9595 - val_loss:
1.0030
Epoch 5/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9448 - val_loss:
1.0015
Epoch 6/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9270 - val_loss:
1.0068
Epoch 7/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9175 - val_loss:
1.0087
Epoch 8/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9073 - val_loss:
1.0126
Epoch 9/200
```

```
47/47 [==============================] - 0s 3ms/step - loss: 0.8952 - val_loss:
1.0100
Epoch 10/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8885 - val_loss:
1.0091
Epoch 11/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8791 - val_loss:
1.0121
Epoch 12/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8694 - val_loss:
1.0164
Epoch 13/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8625 - val_loss:
1.0177
Epoch 14/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8563 - val_loss:
1.0204
Epoch 15/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8483 - val_loss:
1.0218
Epoch 16/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8436 - val_loss:
1.0256
Epoch 17/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8368 - val_loss:
1.0296
Epoch 18/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8331 - val_loss:
1.0372
Epoch 19/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8298 - val_loss:
1.0321
Epoch 20/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8227 - val_loss:
1.0373
Epoch 21/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8199 - val_loss:
1.0400
Epoch 22/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8171 - val_loss:
1.0339
Epoch 23/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8115 - val_loss:
1.0431
Epoch 24/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8093 - val_loss:
1.0452
Epoch 25/200
```

```
47/47 [==============================] - 0s 3ms/step - loss: 0.8021 - val_loss:
1.0450
Epoch 26/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8014 - val_loss:
1.0444
Epoch 27/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7985 - val_loss:
1.0499
Epoch 28/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7948 - val_loss:
1.0542
Epoch 29/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7933 - val_loss:
1.0682
Epoch 30/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7898 - val_loss:
1.0518
Epoch 31/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7889 - val_loss:
1.0662
Epoch 32/200
47/47 [==============================] - 0s 5ms/step - loss: 0.7833 - val_loss:
1.0631
Epoch 33/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7810 - val_loss:
1.0614
Epoch 34/200
47/47 [==============================] - 0s 5ms/step - loss: 0.7813 - val_loss:
1.0641
Epoch 35/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7772 - val_loss:
1.0715
Epoch 36/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7745 - val_loss:
1.0741
Epoch 37/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7735 - val_loss:
1.0725
Epoch 38/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7689 - val_loss:
1.0710
Epoch 39/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7671 - val_loss:
1.0701
Epoch 40/200
47/47 [==============================] - 1s 18ms/step - loss: 0.7652 - val_loss:
1.0800
Epoch 41/200
```

```
47/47 [==============================] - 1s 15ms/step - loss: 0.7639 - val_loss:
1.0827
Epoch 42/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7611 - val_loss:
1.0787
Epoch 43/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7603 - val_loss:
1.0765
Epoch 44/200
47/47 [==============================] - 1s 13ms/step - loss: 0.7565 - val_loss:
1.0799
Epoch 45/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7579 - val_loss:
1.0839
Epoch 46/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7546 - val_loss:
1.0789
Epoch 47/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7522 - val_loss:
1.0831
Epoch 48/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7495 - val_loss:
1.0804
Epoch 49/200
47/47 [==============================] - 0s 8ms/step - loss: 0.7480 - val_loss:
1.0841
Epoch 50/200
47/47 [==============================] - 0s 6ms/step - loss: 0.7459 - val_loss:
1.0900
Epoch 51/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7442 - val_loss:
1.0826
Epoch 52/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7430 - val_loss:
1.0868
Epoch 53/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7431 - val_loss:
1.0880
Epoch 54/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7397 - val_loss:
1.0870
Epoch 55/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7404 - val_loss:
1.0931
Epoch 56/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7380 - val_loss:
1.0898
Epoch 57/200
```

```
47/47 [==============================] - 0s 3ms/step - loss: 0.7364 - val_loss:
1.0936
Epoch 58/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7348 - val_loss:
1.0882
Epoch 59/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7358 - val_loss:
1.0963
Epoch 60/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7304 - val_loss:
1.0909
Epoch 61/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7292 - val_loss:
1.0975
Epoch 62/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7297 - val_loss:
1.0903
Epoch 63/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7279 - val_loss:
1.0946
Epoch 64/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7265 - val_loss:
1.0977
Epoch 65/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7243 - val_loss:
1.0870
Epoch 66/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7237 - val_loss:
1.0880
Epoch 67/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7230 - val_loss:
1.0878
Epoch 68/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7229 - val_loss:
1.0912
Epoch 69/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7214 - val_loss:
1.0929
Epoch 70/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7205 - val_loss:
1.0981
Epoch 71/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7202 - val_loss:
1.0951
Epoch 72/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7196 - val_loss:
1.0981
Epoch 73/200
```

```
47/47 [==============================] - 0s 3ms/step - loss: 0.7172 - val_loss:
1.0849
Epoch 74/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7155 - val_loss:
1.1083
Epoch 75/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7140 - val_loss:
1.0958
Epoch 76/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7115 - val_loss:
1.0896
Epoch 77/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7131 - val_loss:
1.1047
Epoch 78/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7092 - val_loss:
1.0921
Epoch 79/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7103 - val_loss:
1.0944
Epoch 80/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7117 - val_loss:
1.1012
Epoch 81/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7073 - val_loss:
1.0976
Epoch 82/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7090 - val_loss:
1.1045
Epoch 83/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7057 - val_loss:
1.0992
Epoch 84/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7058 - val_loss:
1.0991
Epoch 85/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7056 - val_loss:
1.0943
Epoch 86/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7031 - val_loss:
1.1060
Epoch 87/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7014 - val_loss:
1.0946
Epoch 88/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7014 - val_loss:
1.1113
Epoch 89/200
```

```
47/47 [==============================] - 0s 3ms/step - loss: 0.7008 - val_loss:
1.1134
Epoch 90/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7017 - val_loss:
1.0994
Epoch 91/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6991 - val_loss:
1.1129
Epoch 92/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7001 - val_loss:
1.1084
Epoch 93/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7011 - val_loss:
1.1092
Epoch 94/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7016 - val_loss:
1.0969
Epoch 95/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6968 - val_loss:
1.1067
Epoch 96/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6957 - val_loss:
1.1042
Epoch 97/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6939 - val_loss:
1.1107
Epoch 98/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6947 - val_loss:
1.1207
Epoch 99/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6911 - val_loss:
1.1109
Epoch 100/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6912 - val_loss:
1.1084
Epoch 101/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6897 - val_loss:
1.1056
Epoch 102/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6896 - val_loss:
1.1066
Epoch 103/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6884 - val_loss:
1.1126
Epoch 104/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6860 - val_loss:
1.1019
Epoch 105/200
```

```
47/47 [==============================] - 0s 3ms/step - loss: 0.6881 - val_loss:
1.1065
Epoch 106/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6864 - val_loss:
1.1027
Epoch 107/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6852 - val_loss:
1.1115
Epoch 108/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6815 - val_loss:
1.1029
Epoch 109/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6811 - val_loss:
1.1031
Epoch 110/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6827 - val_loss:
1.1062
Epoch 111/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6801 - val_loss:
1.1036
Epoch 112/200
47/47 [==============================] - 0s 5ms/step - loss: 0.6800 - val_loss:
1.1120
Epoch 113/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6796 - val_loss:
1.1081
Epoch 114/200
47/47 [==============================] - 0s 5ms/step - loss: 0.6771 - val_loss:
1.1029
Epoch 115/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6766 - val_loss:
1.0970
Epoch 116/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6770 - val_loss:
1.1055
Epoch 117/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6769 - val_loss:
1.1105
Epoch 118/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6763 - val_loss:
1.1012
Epoch 119/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6731 - val_loss:
1.1014
Epoch 120/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6737 - val_loss:
1.1074
Epoch 121/200
```

```
47/47 [==============================] - 0s 4ms/step - loss: 0.6738 - val_loss:
1.1123
Epoch 122/200
47/47 [==============================] - 0s 5ms/step - loss: 0.6718 - val_loss:
1.1017
Epoch 123/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6704 - val_loss:
1.1016
Epoch 124/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6708 - val_loss:
1.1182
Epoch 125/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6710 - val_loss:
1.1176
Epoch 126/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6693 - val_loss:
1.1149
Epoch 127/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6685 - val_loss:
1.1067
Epoch 128/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6670 - val_loss:
1.1149
Epoch 129/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6688 - val_loss:
1.1062
Epoch 130/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6652 - val_loss:
1.1068
Epoch 131/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6682 - val_loss:
1.1168
Epoch 132/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6656 - val_loss:
1.1175
Epoch 133/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6662 - val_loss:
1.1164
Epoch 134/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6638 - val_loss:
1.1154
Epoch 135/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6636 - val_loss:
1.1305
Epoch 136/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6629 - val_loss:
1.1093
Epoch 137/200
```

```
47/47 [==============================] - 0s 3ms/step - loss: 0.6631 - val_loss:
1.1310
Epoch 138/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6640 - val_loss:
1.1186
Epoch 139/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6608 - val_loss:
1.1132
Epoch 140/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6608 - val_loss:
1.1175
Epoch 141/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6598 - val_loss:
1.1137
Epoch 142/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6601 - val_loss:
1.1041
Epoch 143/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6600 - val_loss:
1.1133
Epoch 144/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6588 - val_loss:
1.1190
Epoch 145/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6565 - val_loss:
1.1305
Epoch 146/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6574 - val_loss:
1.1200
Epoch 147/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6574 - val_loss:
1.1106
Epoch 148/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6576 - val_loss:
1.1229
Epoch 149/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6555 - val_loss:
1.1069
Epoch 150/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6546 - val_loss:
1.1190
Epoch 151/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6553 - val_loss:
1.1208
Epoch 152/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6535 - val_loss:
1.1309
Epoch 153/200
```

```
47/47 [==============================] - 0s 3ms/step - loss: 0.6552 - val_loss:
1.1216
Epoch 154/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6519 - val_loss:
1.1166
Epoch 155/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6490 - val_loss:
1.1132
Epoch 156/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6501 - val_loss:
1.1262
Epoch 157/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6516 - val_loss:
1.1249
Epoch 158/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6523 - val_loss:
1.1167
Epoch 159/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6502 - val_loss:
1.1397
Epoch 160/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6498 - val_loss:
1.1247
Epoch 161/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6480 - val_loss:
1.1196
Epoch 162/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6453 - val_loss:
1.1305
Epoch 163/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6461 - val_loss:
1.1256
Epoch 164/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6476 - val_loss:
1.1185
Epoch 165/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6489 - val_loss:
1.1203
Epoch 166/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6464 - val_loss:
1.1285
Epoch 167/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6469 - val_loss:
1.1157
Epoch 168/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6444 - val_loss:
1.1213
Epoch 169/200
```

```
47/47 [==============================] - 0s 3ms/step - loss: 0.6430 - val_loss:
1.1383
Epoch 170/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6460 - val_loss:
1.1288
Epoch 171/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6432 - val_loss:
1.1220
Epoch 172/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6409 - val_loss:
1.1364
Epoch 173/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6432 - val_loss:
1.1183
Epoch 174/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6422 - val_loss:
1.1376
Epoch 175/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6406 - val_loss:
1.1279
Epoch 176/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6391 - val_loss:
1.1284
Epoch 177/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6402 - val_loss:
1.1279
Epoch 178/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6395 - val_loss:
1.1267
Epoch 179/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6423 - val_loss:
1.1372
Epoch 180/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6411 - val_loss:
1.1336
Epoch 181/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6375 - val_loss:
1.1165
Epoch 182/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6382 - val_loss:
1.1254
Epoch 183/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6386 - val_loss:
1.1215
Epoch 184/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6350 - val_loss:
1.1336
Epoch 185/200
```

```
47/47 [==============================] - 0s 3ms/step - loss: 0.6361 - val_loss:
1.1265
Epoch 186/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6351 - val_loss:
1.1286
Epoch 187/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6375 - val_loss:
1.1390
Epoch 188/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6336 - val_loss:
1.1338
Epoch 189/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6348 - val_loss:
1.1347
Epoch 190/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6318 - val_loss:
1.1228
Epoch 191/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6318 - val_loss:
1.1338
Epoch 192/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6302 - val_loss:
1.1276
Epoch 193/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6319 - val_loss:
1.1375
Epoch 194/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6320 - val_loss:
1.1248
Epoch 195/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6303 - val_loss:
1.1399
Epoch 196/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6306 - val_loss:
1.1403
Epoch 197/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6299 - val_loss:
1.1317
Epoch 198/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6291 - val_loss:
1.1345
Epoch 199/200
47/47 [==============================] - 0s 5ms/step - loss: 0.6269 - val_loss:
1.1316
Epoch 200/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6299 - val_loss:
1.1367
47/47 [==============================] - 0s 2ms/step
```

```
8/8 [==============================] - 0s 3ms/step
Mean Squared Error of FFNN log returns: 0.00042811614116084693
Mean Squared Error of FFNN prices: 44.35149858958689
Epoch 1/200
47/47 [==============================] - 1s 5ms/step - loss: 1.3942 - val_loss:
1.1968
Epoch 2/200
47/47 [==============================] - 0s 3ms/step - loss: 1.1883 - val_loss:
1.1113
Epoch 3/200
47/47 [==============================] - 0s 3ms/step - loss: 1.1072 - val_loss:
1.0781
Epoch 4/200
47/47 [==============================] - 0s 3ms/step - loss: 1.0577 - val_loss:
1.0609
Epoch 5/200
47/47 [==============================] - 0s 3ms/step - loss: 1.0202 - val_loss:
1.0518
Epoch 6/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9899 - val_loss:
1.0504
Epoch 7/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9718 - val_loss:
1.0453
Epoch 8/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9557 - val_loss:
1.0421
Epoch 9/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9395 - val_loss:
1.0396
Epoch 10/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9301 - val_loss:
1.0453
Epoch 11/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9124 - val_loss:
1.0386
Epoch 12/200
47/47 [==============================] - 0s 3ms/step - loss: 0.9009 - val_loss:
1.0432
Epoch 13/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8917 - val_loss:
1.0417
Epoch 14/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8825 - val_loss:
1.0439
Epoch 15/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8768 - val_loss:
1.0464
```

```
Epoch 16/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8708 - val_loss:
1.0443
Epoch 17/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8645 - val_loss:
1.0458
Epoch 18/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8626 - val_loss:
1.0508
Epoch 19/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8510 - val_loss:
1.0511
Epoch 20/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8443 - val_loss:
1.0574
Epoch 21/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8397 - val_loss:
1.0551
Epoch 22/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8347 - val_loss:
1.0526
Epoch 23/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8308 - val_loss:
1.0520
Epoch 24/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8275 - val_loss:
1.0620
Epoch 25/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8252 - val_loss:
1.0553
Epoch 26/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8198 - val_loss:
1.0586
Epoch 27/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8156 - val_loss:
1.0617
Epoch 28/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8117 - val_loss:
1.0559
Epoch 29/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8100 - val_loss:
1.0590
Epoch 30/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8072 - val_loss:
1.0686
Epoch 31/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8047 - val_loss:
1.0673
```

```
Epoch 32/200
47/47 [==============================] - 0s 3ms/step - loss: 0.8032 - val_loss:
1.0580
Epoch 33/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7993 - val_loss:
1.0664
Epoch 34/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7969 - val_loss:
1.0678
Epoch 35/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7953 - val_loss:
1.0667
Epoch 36/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7941 - val_loss:
1.0677
Epoch 37/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7912 - val_loss:
1.0685
Epoch 38/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7872 - val_loss:
1.0652
Epoch 39/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7879 - val_loss:
1.0760
Epoch 40/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7849 - val_loss:
1.0710
Epoch 41/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7818 - val_loss:
1.0744
Epoch 42/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7786 - val_loss:
1.0728
Epoch 43/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7777 - val_loss:
1.0750
Epoch 44/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7758 - val_loss:
1.0791
Epoch 45/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7743 - val_loss:
1.0735
Epoch 46/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7730 - val_loss:
1.0785
Epoch 47/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7719 - val_loss:
1.0768
```

```
Epoch 48/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7683 - val_loss:
1.0751
Epoch 49/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7708 - val_loss:
1.0800
Epoch 50/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7659 - val_loss:
1.0766
Epoch 51/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7656 - val_loss:
1.0757
Epoch 52/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7617 - val_loss:
1.0783
Epoch 53/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7609 - val_loss:
1.0793
Epoch 54/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7592 - val_loss:
1.0798
Epoch 55/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7557 - val_loss:
1.0805
Epoch 56/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7578 - val_loss:
1.0835
Epoch 57/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7541 - val_loss:
1.0833
Epoch 58/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7517 - val_loss:
1.0728
Epoch 59/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7524 - val_loss:
1.0798
Epoch 60/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7510 - val_loss:
1.0783
Epoch 61/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7478 - val_loss:
1.0811
Epoch 62/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7474 - val_loss:
1.0788
Epoch 63/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7450 - val_loss:
1.0871
```

```
Epoch 64/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7447 - val_loss:
1.0831
Epoch 65/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7429 - val_loss:
1.0836
Epoch 66/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7417 - val_loss:
1.0739
Epoch 67/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7418 - val_loss:
1.0912
Epoch 68/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7383 - val_loss:
1.0834
Epoch 69/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7394 - val_loss:
1.0927
Epoch 70/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7404 - val_loss:
1.0809
Epoch 71/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7358 - val_loss:
1.0890
Epoch 72/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7358 - val_loss:
1.0859
Epoch 73/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7349 - val_loss:
1.0901
Epoch 74/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7332 - val_loss:
1.0854
Epoch 75/200
47/47 [==============================] - 0s 5ms/step - loss: 0.7332 - val_loss:
1.0934
Epoch 76/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7316 - val_loss:
1.0887
Epoch 77/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7308 - val_loss:
1.0892
Epoch 78/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7310 - val_loss:
1.0843
Epoch 79/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7293 - val_loss:
1.0886
```

```
Epoch 80/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7273 - val_loss:
1.0945
Epoch 81/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7254 - val_loss:
1.0966
Epoch 82/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7265 - val_loss:
1.0866
Epoch 83/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7230 - val_loss:
1.0859
Epoch 84/200
47/47 [==============================] - 0s 4ms/step - loss: 0.7210 - val_loss:
1.0932
Epoch 85/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7210 - val_loss:
1.0918
Epoch 86/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7191 - val_loss:
1.0927
Epoch 87/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7230 - val_loss:
1.1018
Epoch 88/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7192 - val_loss:
1.0929
Epoch 89/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7163 - val_loss:
1.0928
Epoch 90/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7130 - val_loss:
1.0934
Epoch 91/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7163 - val_loss:
1.0908
Epoch 92/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7138 - val_loss:
1.0955
Epoch 93/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7139 - val_loss:
1.0900
Epoch 94/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7116 - val_loss:
1.0959
Epoch 95/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7097 - val_loss:
1.0918
```

```
Epoch 96/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7099 - val_loss:
1.0904
Epoch 97/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7093 - val_loss:
1.0988
Epoch 98/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7089 - val_loss:
1.0959
Epoch 99/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7086 - val_loss:
1.0922
Epoch 100/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7048 - val_loss:
1.0980
Epoch 101/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7032 - val_loss:
1.0983
Epoch 102/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7050 - val_loss:
1.0917
Epoch 103/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7048 - val_loss:
1.1018
Epoch 104/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7018 - val_loss:
1.0967
Epoch 105/200
47/47 [==============================] - 0s 3ms/step - loss: 0.7005 - val_loss:
1.0980
Epoch 106/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6982 - val_loss:
1.0982
Epoch 107/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6982 - val_loss:
1.0968
Epoch 108/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6984 - val_loss:
1.1009
Epoch 109/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6974 - val_loss:
1.0944
Epoch 110/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6957 - val_loss:
1.1011
Epoch 111/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6933 - val_loss:
1.0993
```

```
Epoch 112/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6947 - val_loss:
1.1016
Epoch 113/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6934 - val_loss:
1.1074
Epoch 114/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6916 - val_loss:
1.1063
Epoch 115/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6915 - val_loss:
1.1128
Epoch 116/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6889 - val_loss:
1.1108
Epoch 117/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6890 - val_loss:
1.1028
Epoch 118/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6860 - val_loss:
1.1180
Epoch 119/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6884 - val_loss:
1.1097
Epoch 120/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6858 - val_loss:
1.1137
Epoch 121/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6846 - val_loss:
1.1102
Epoch 122/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6839 - val_loss:
1.1076
Epoch 123/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6825 - val_loss:
1.1131
Epoch 124/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6808 - val_loss:
1.1127
Epoch 125/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6817 - val_loss:
1.1210
Epoch 126/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6801 - val_loss:
1.1153
Epoch 127/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6792 - val_loss:
1.1155
```

```
Epoch 128/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6788 - val_loss:
1.1108
Epoch 129/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6763 - val_loss:
1.1217
Epoch 130/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6750 - val_loss:
1.1198
Epoch 131/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6745 - val_loss:
1.1191
Epoch 132/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6747 - val_loss:
1.1130
Epoch 133/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6754 - val_loss:
1.1219
Epoch 134/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6713 - val_loss:
1.1277
Epoch 135/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6730 - val_loss:
1.1283
Epoch 136/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6694 - val_loss:
1.1200
Epoch 137/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6698 - val_loss:
1.1242
Epoch 138/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6682 - val_loss:
1.1151
Epoch 139/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6724 - val_loss:
1.1328
Epoch 140/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6674 - val_loss:
1.1239
Epoch 141/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6662 - val_loss:
1.1335
Epoch 142/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6663 - val_loss:
1.1254
Epoch 143/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6643 - val_loss:
1.1374
```

```
Epoch 144/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6642 - val_loss:
1.1227
Epoch 145/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6643 - val_loss:
1.1231
Epoch 146/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6625 - val_loss:
1.1324
Epoch 147/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6601 - val_loss:
1.1321
Epoch 148/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6589 - val_loss:
1.1275
Epoch 149/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6587 - val_loss:
1.1324
Epoch 150/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6626 - val_loss:
1.1292
Epoch 151/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6591 - val_loss:
1.1343
Epoch 152/200
47/47 [==============================] - 0s 5ms/step - loss: 0.6571 - val_loss:
1.1453
Epoch 153/200
47/47 [==============================] - 0s 5ms/step - loss: 0.6560 - val_loss:
1.1403
Epoch 154/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6563 - val_loss:
1.1361
Epoch 155/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6536 - val_loss:
1.1374
Epoch 156/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6529 - val_loss:
1.1349
Epoch 157/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6518 - val_loss:
1.1593
Epoch 158/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6540 - val_loss:
1.1429
Epoch 159/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6510 - val_loss:
1.1440
```

```
Epoch 160/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6515 - val_loss:
1.1358
Epoch 161/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6492 - val_loss:
1.1513
Epoch 162/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6461 - val_loss:
1.1462
Epoch 163/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6456 - val_loss:
1.1518
Epoch 164/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6461 - val_loss:
1.1392
Epoch 165/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6464 - val_loss:
1.1490
Epoch 166/200
47/47 [==============================] - 0s 5ms/step - loss: 0.6434 - val_loss:
1.1491
Epoch 167/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6449 - val_loss:
1.1503
Epoch 168/200
47/47 [==============================] - 0s 5ms/step - loss: 0.6452 - val_loss:
1.1482
Epoch 169/200
47/47 [==============================] - 0s 6ms/step - loss: 0.6439 - val_loss:
1.1600
Epoch 170/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6442 - val_loss:
1.1417
Epoch 171/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6406 - val_loss:
1.1508
Epoch 172/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6409 - val_loss:
1.1485
Epoch 173/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6443 - val_loss:
1.1402
Epoch 174/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6395 - val_loss:
1.1544
Epoch 175/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6382 - val_loss:
1.1445
```

```
Epoch 176/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6369 - val_loss:
1.1511
Epoch 177/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6350 - val_loss:
1.1454
Epoch 178/200
47/47 [==============================] - 0s 4ms/step - loss: 0.6395 - val_loss:
1.1666
Epoch 179/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6387 - val_loss:
1.1491
Epoch 180/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6331 - val_loss:
1.1513
Epoch 181/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6315 - val_loss:
1.1575
Epoch 182/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6348 - val_loss:
1.1512
Epoch 183/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6362 - val_loss:
1.1553
Epoch 184/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6316 - val_loss:
1.1545
Epoch 185/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6296 - val_loss:
1.1577
Epoch 186/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6273 - val_loss:
1.1484
Epoch 187/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6273 - val_loss:
1.1524
Epoch 188/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6292 - val_loss:
1.1599
Epoch 189/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6267 - val_loss:
1.1465
Epoch 190/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6267 - val_loss:
1.1522
Epoch 191/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6285 - val_loss:
1.1548
```

```
Epoch 192/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6241 - val_loss:
1.1520
Epoch 193/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6247 - val_loss:
1.1492
Epoch 194/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6279 - val_loss:
1.1388
Epoch 195/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6254 - val_loss:
1.1544
Epoch 196/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6236 - val_loss:
1.1568
Epoch 197/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6203 - val_loss:
1.1743
Epoch 198/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6198 - val_loss:
1.1477
Epoch 199/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6198 - val_loss:
1.1756
Epoch 200/200
47/47 [==============================] - 0s 3ms/step - loss: 0.6217 - val_loss:
1.1670
47/47 [==============================] - 0s 2ms/step
8/8 [==============================] - 0s 3ms/step
Mean Squared Error of FFNN log returns: 0.00043953447652737485
Mean Squared Error of FFNN prices: 45.581017731390766
MSE for FFNN log returns 0.00043382530884411086
MSE for FFNN prices 44.96625816048883
```

[70]:

```
# Ignore the first two MSE for FFNN as these are just the results from the
↪final loop. the ones of interest are the final two
```

[70]:

2.0.3 LSTM

[70]:

[70]:

[70]:

[70]:

```

```

[71]:

```
# streamline the data to make it easier to use

data_LSTM = GS.filter(['Adj Close'])
train = train.filter(['Adj Close'])
train = train.filter(['Adj Close'])
```

[72]:

```
data_unscaled = data_LSTM.values
```

[73]:

```
# Get the number of rows to train the model on
train_data_length = len(train)

# Reshape the array to have one column
data_reshaped = data_unscaled.reshape(-1, 1)

# Scale the data
mmscaler = MinMaxScaler(feature_range=(0, 1))
np_data = mmscaler.fit_transform(data_reshaped)
```

[73]:

```

```

[74]:

```
# The window variable sets the number of time steps used to make a single␣
↪prediction.
window = 50

# The index_Close variable is used to locate the column index of the 'Adj␣
↪Close' feature in the data. This index will be used to extract the true␣
↪values for the prediction.
index_Close = data_LSTM.columns.get_loc("Adj Close")
print(index_Close)
# The train_data_len variable is used to determine how much data to use for␣
↪training the model.
train_data_len = len(train)

# The train_data and test_data variables are used to split the data into␣
↪training and testing sets.
train_data = np_data[0:train_data_len, :]
test_data = np_data[train_data_len - window:, :]

# This function takes in a window size and a DataFrame of training data and␣
↪partitions the data into input/output pairs.
# It creates a list of input sequences with length equal to the window size and␣
↪corresponding output values for each sequence.
# The function then converts the lists to numpy arrays and returns them.
def partition_dataset(window, train_df):
```

```
    x, y = [], []
    data_len = train_df.shape[0]
    for i in range(window, data_len):
        x.append(train_df[i-window:i,:])
        y.append(train_df[i, index_Close])

    # Convert the x and y to numpy arrays
    x = np.array(x)
    y = np.array(y)
    return x, y

# The x_train and y_train arrays contain the training data, while the x_test␣
↪and y_test arrays contain the testing data
x_train, y_train = partition_dataset(window, train_data)
x_test, y_test = partition_dataset(window, test_data)


print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)

# This part of the code verifies that the predicted output and the input data␣
↪correspond correctly.
# It checks if the final closing price of the second input sample is the same␣
↪as the initial predicted output value.
print(x_test[1][window-1][index_Close])
print(y_test[0])
```

```
0
(1461, 50, 1) (1461,)
(250, 50, 1) (250,)
0.9107838133610049
0.9107838133610049
```

[75]:
```
# Configure

# Create a Sequential model
LSTM_modl = Sequential()

# The number of neurons is set to the size of the window
neurons = window

# This model has 'window' number of neurons and the input shape is defined as␣
↪having 'window' timestamps.
# Add a LSTM layer with return sequences set to True, and the input shape set␣
↪to (training sequence length, 1)
LSTM_modl.add(LSTM(neurons, return_sequences=True, input_shape=(x_train.
↪shape[1], 1)))
```

```python
LSTM_modl.add(LSTM(neurons, return_sequences=False))
LSTM_modl.add(Dense(25, activation='relu'))
LSTM_modl.add(Dense(1))

# Compile
LSTM_modl.compile(optimizer='adam', loss='mean_squared_error')
```

[76]:
```python
# Training the model
LSTM_modl.fit(x_train, y_train, batch_size=16, epochs=200)
```

```
Epoch 1/150
92/92 [==============================] - 11s 69ms/step - loss: 0.0149
Epoch 2/150
92/92 [==============================] - 5s 56ms/step - loss: 0.0014
Epoch 3/150
92/92 [==============================] - 6s 67ms/step - loss: 0.0012
Epoch 4/150
92/92 [==============================] - 5s 54ms/step - loss: 0.0010
Epoch 5/150
92/92 [==============================] - 6s 60ms/step - loss: 9.2529e-04
Epoch 6/150
92/92 [==============================] - 6s 63ms/step - loss: 9.1167e-04
Epoch 7/150
92/92 [==============================] - 5s 55ms/step - loss: 0.0011
Epoch 8/150
92/92 [==============================] - 6s 68ms/step - loss: 9.1727e-04
Epoch 9/150
92/92 [==============================] - 5s 57ms/step - loss: 7.1406e-04
Epoch 10/150
92/92 [==============================] - 6s 67ms/step - loss: 6.8549e-04
Epoch 11/150
92/92 [==============================] - 5s 56ms/step - loss: 6.5130e-04
Epoch 12/150
92/92 [==============================] - 5s 58ms/step - loss: 7.1215e-04
Epoch 13/150
92/92 [==============================] - 6s 66ms/step - loss: 5.8833e-04
Epoch 14/150
92/92 [==============================] - 5s 57ms/step - loss: 6.3485e-04
Epoch 15/150
92/92 [==============================] - 6s 67ms/step - loss: 5.2784e-04
Epoch 16/150
92/92 [==============================] - 5s 56ms/step - loss: 5.0798e-04
Epoch 17/150
92/92 [==============================] - 6s 66ms/step - loss: 4.8348e-04
Epoch 18/150
92/92 [==============================] - 5s 55ms/step - loss: 4.6698e-04
Epoch 19/150
```

```
92/92 [==============================] - 5s 57ms/step - loss: 4.6357e-04
Epoch 20/150
92/92 [==============================] - 6s 63ms/step - loss: 4.9169e-04
Epoch 21/150
92/92 [==============================] - 5s 57ms/step - loss: 4.9502e-04
Epoch 22/150
92/92 [==============================] - 6s 67ms/step - loss: 4.1920e-04
Epoch 23/150
92/92 [==============================] - 5s 55ms/step - loss: 4.2657e-04
Epoch 24/150
92/92 [==============================] - 6s 64ms/step - loss: 4.4508e-04
Epoch 25/150
92/92 [==============================] - 5s 58ms/step - loss: 4.6083e-04
Epoch 26/150
92/92 [==============================] - 5s 56ms/step - loss: 4.9100e-04
Epoch 27/150
92/92 [==============================] - 6s 68ms/step - loss: 3.8181e-04
Epoch 28/150
92/92 [==============================] - 5s 57ms/step - loss: 3.3999e-04
Epoch 29/150
92/92 [==============================] - 6s 69ms/step - loss: 3.3197e-04
Epoch 30/150
92/92 [==============================] - 5s 57ms/step - loss: 3.2849e-04
Epoch 31/150
92/92 [==============================] - 6s 67ms/step - loss: 3.2541e-04
Epoch 32/150
92/92 [==============================] - 5s 57ms/step - loss: 3.3733e-04
Epoch 33/150
92/92 [==============================] - 5s 55ms/step - loss: 3.0748e-04
Epoch 34/150
92/92 [==============================] - 6s 68ms/step - loss: 3.6880e-04
Epoch 35/150
92/92 [==============================] - 5s 57ms/step - loss: 3.2678e-04
Epoch 36/150
92/92 [==============================] - 6s 67ms/step - loss: 2.8632e-04
Epoch 37/150
92/92 [==============================] - 5s 56ms/step - loss: 3.3125e-04
Epoch 38/150
92/92 [==============================] - 6s 67ms/step - loss: 2.8393e-04
Epoch 39/150
92/92 [==============================] - 5s 58ms/step - loss: 2.5162e-04
Epoch 40/150
92/92 [==============================] - 5s 56ms/step - loss: 2.5160e-04
Epoch 41/150
92/92 [==============================] - 6s 69ms/step - loss: 2.6123e-04
Epoch 42/150
92/92 [==============================] - 5s 56ms/step - loss: 2.3169e-04
Epoch 43/150
```

```
92/92 [==============================] - 6s 68ms/step - loss: 2.7231e-04
Epoch 44/150
92/92 [==============================] - 5s 57ms/step - loss: 2.2511e-04
Epoch 45/150
92/92 [==============================] - 6s 65ms/step - loss: 2.8823e-04
Epoch 46/150
92/92 [==============================] - 5s 58ms/step - loss: 2.6126e-04
Epoch 47/150
92/92 [==============================] - 5s 57ms/step - loss: 2.3885e-04
Epoch 48/150
92/92 [==============================] - 6s 67ms/step - loss: 2.3176e-04
Epoch 49/150
92/92 [==============================] - 5s 54ms/step - loss: 2.1953e-04
Epoch 50/150
92/92 [==============================] - 6s 68ms/step - loss: 2.3703e-04
Epoch 51/150
92/92 [==============================] - 5s 56ms/step - loss: 2.5605e-04
Epoch 52/150
92/92 [==============================] - 6s 64ms/step - loss: 2.8949e-04
Epoch 53/150
92/92 [==============================] - 5s 58ms/step - loss: 2.6962e-04
Epoch 54/150
92/92 [==============================] - 5s 54ms/step - loss: 2.5943e-04
Epoch 55/150
92/92 [==============================] - 6s 68ms/step - loss: 2.2789e-04
Epoch 56/150
92/92 [==============================] - 5s 56ms/step - loss: 2.3066e-04
Epoch 57/150
92/92 [==============================] - 6s 68ms/step - loss: 2.2216e-04
Epoch 58/150
92/92 [==============================] - 5s 56ms/step - loss: 2.5802e-04
Epoch 59/150
92/92 [==============================] - 5s 60ms/step - loss: 2.8640e-04
Epoch 60/150
92/92 [==============================] - 6s 62ms/step - loss: 2.4093e-04
Epoch 61/150
92/92 [==============================] - 5s 55ms/step - loss: 2.2420e-04
Epoch 62/150
92/92 [==============================] - 6s 69ms/step - loss: 2.2851e-04
Epoch 63/150
92/92 [==============================] - 5s 55ms/step - loss: 2.1768e-04
Epoch 64/150
92/92 [==============================] - 6s 66ms/step - loss: 2.2420e-04
Epoch 65/150
92/92 [==============================] - 5s 55ms/step - loss: 2.2228e-04
Epoch 66/150
92/92 [==============================] - 5s 56ms/step - loss: 2.4116e-04
Epoch 67/150
```

```
92/92 [==============================] - 6s 65ms/step - loss: 2.4281e-04
Epoch 68/150
92/92 [==============================] - 5s 55ms/step - loss: 2.4225e-04
Epoch 69/150
92/92 [==============================] - 6s 69ms/step - loss: 2.2064e-04
Epoch 70/150
92/92 [==============================] - 5s 55ms/step - loss: 2.1297e-04
Epoch 71/150
92/92 [==============================] - 6s 68ms/step - loss: 2.2032e-04
Epoch 72/150
92/92 [==============================] - 5s 56ms/step - loss: 2.3909e-04
Epoch 73/150
92/92 [==============================] - 5s 56ms/step - loss: 2.1520e-04
Epoch 74/150
92/92 [==============================] - 6s 65ms/step - loss: 2.2250e-04
Epoch 75/150
92/92 [==============================] - 5s 55ms/step - loss: 2.7802e-04
Epoch 76/150
92/92 [==============================] - 6s 67ms/step - loss: 2.4105e-04
Epoch 77/150
92/92 [==============================] - 5s 54ms/step - loss: 2.0760e-04
Epoch 78/150
92/92 [==============================] - 5s 59ms/step - loss: 3.2962e-04
Epoch 79/150
92/92 [==============================] - 6s 63ms/step - loss: 2.1312e-04
Epoch 80/150
92/92 [==============================] - 5s 57ms/step - loss: 2.1523e-04
Epoch 81/150
92/92 [==============================] - 6s 67ms/step - loss: 2.4542e-04
Epoch 82/150
92/92 [==============================] - 5s 55ms/step - loss: 2.3678e-04
Epoch 83/150
92/92 [==============================] - 6s 69ms/step - loss: 2.3534e-04
Epoch 84/150
92/92 [==============================] - 5s 56ms/step - loss: 2.0779e-04
Epoch 85/150
92/92 [==============================] - 5s 58ms/step - loss: 2.1643e-04
Epoch 86/150
92/92 [==============================] - 6s 63ms/step - loss: 2.1700e-04
Epoch 87/150
92/92 [==============================] - 5s 56ms/step - loss: 2.4595e-04
Epoch 88/150
92/92 [==============================] - 6s 68ms/step - loss: 2.0826e-04
Epoch 89/150
92/92 [==============================] - 5s 55ms/step - loss: 1.9967e-04
Epoch 90/150
92/92 [==============================] - 6s 67ms/step - loss: 2.5447e-04
Epoch 91/150
```

```
92/92 [==============================] - 5s 55ms/step - loss: 2.4502e-04
Epoch 92/150
92/92 [==============================] - 5s 58ms/step - loss: 2.1931e-04
Epoch 93/150
92/92 [==============================] - 6s 64ms/step - loss: 2.1937e-04
Epoch 94/150
92/92 [==============================] - 5s 56ms/step - loss: 2.1030e-04
Epoch 95/150
92/92 [==============================] - 6s 70ms/step - loss: 2.1800e-04
Epoch 96/150
92/92 [==============================] - 5s 56ms/step - loss: 2.1388e-04
Epoch 97/150
92/92 [==============================] - 6s 68ms/step - loss: 2.0222e-04
Epoch 98/150
92/92 [==============================] - 5s 56ms/step - loss: 2.0937e-04
Epoch 99/150
92/92 [==============================] - 5s 59ms/step - loss: 2.2945e-04
Epoch 100/150
92/92 [==============================] - 6s 64ms/step - loss: 2.1057e-04
Epoch 101/150
92/92 [==============================] - 5s 56ms/step - loss: 2.4927e-04
Epoch 102/150
92/92 [==============================] - 6s 68ms/step - loss: 2.3501e-04
Epoch 103/150
92/92 [==============================] - 5s 56ms/step - loss: 2.1896e-04
Epoch 104/150
92/92 [==============================] - 6s 67ms/step - loss: 2.2302e-04
Epoch 105/150
92/92 [==============================] - 5s 55ms/step - loss: 2.0999e-04
Epoch 106/150
92/92 [==============================] - 5s 58ms/step - loss: 2.1080e-04
Epoch 107/150
92/92 [==============================] - 6s 64ms/step - loss: 2.3553e-04
Epoch 108/150
92/92 [==============================] - 5s 57ms/step - loss: 2.1379e-04
Epoch 109/150
92/92 [==============================] - 6s 69ms/step - loss: 2.4580e-04
Epoch 110/150
92/92 [==============================] - 5s 55ms/step - loss: 2.0339e-04
Epoch 111/150
92/92 [==============================] - 6s 66ms/step - loss: 2.0959e-04
Epoch 112/150
92/92 [==============================] - 5s 56ms/step - loss: 2.1441e-04
Epoch 113/150
92/92 [==============================] - 5s 56ms/step - loss: 2.0560e-04
Epoch 114/150
92/92 [==============================] - 6s 66ms/step - loss: 2.0897e-04
Epoch 115/150
```

```
92/92 [==============================] - 5s 56ms/step - loss: 2.1775e-04
Epoch 116/150
92/92 [==============================] - 6s 67ms/step - loss: 2.3615e-04
Epoch 117/150
92/92 [==============================] - 5s 56ms/step - loss: 2.1295e-04
Epoch 118/150
92/92 [==============================] - 6s 67ms/step - loss: 2.1856e-04
Epoch 119/150
92/92 [==============================] - 5s 58ms/step - loss: 2.0737e-04
Epoch 120/150
92/92 [==============================] - 5s 55ms/step - loss: 2.2115e-04
Epoch 121/150
92/92 [==============================] - 6s 68ms/step - loss: 2.1229e-04
Epoch 122/150
92/92 [==============================] - 5s 56ms/step - loss: 2.2145e-04
Epoch 123/150
92/92 [==============================] - 6s 67ms/step - loss: 2.1625e-04
Epoch 124/150
92/92 [==============================] - 5s 59ms/step - loss: 2.2343e-04
Epoch 125/150
92/92 [==============================] - 6s 68ms/step - loss: 2.0796e-04
Epoch 126/150
92/92 [==============================] - 5s 58ms/step - loss: 2.5563e-04
Epoch 127/150
92/92 [==============================] - 5s 55ms/step - loss: 2.2779e-04
Epoch 128/150
92/92 [==============================] - 6s 67ms/step - loss: 1.9726e-04
Epoch 129/150
92/92 [==============================] - 5s 56ms/step - loss: 2.0375e-04
Epoch 130/150
92/92 [==============================] - 6s 68ms/step - loss: 2.0530e-04
Epoch 131/150
92/92 [==============================] - 5s 53ms/step - loss: 2.4869e-04
Epoch 132/150
92/92 [==============================] - 6s 61ms/step - loss: 2.1686e-04
Epoch 133/150
92/92 [==============================] - 6s 62ms/step - loss: 2.2407e-04
Epoch 134/150
92/92 [==============================] - 5s 57ms/step - loss: 2.0410e-04
Epoch 135/150
92/92 [==============================] - 6s 69ms/step - loss: 1.9918e-04
Epoch 136/150
92/92 [==============================] - 5s 54ms/step - loss: 2.1627e-04
Epoch 137/150
92/92 [==============================] - 6s 68ms/step - loss: 2.0190e-04
Epoch 138/150
92/92 [==============================] - 6s 61ms/step - loss: 2.2791e-04
Epoch 139/150
```

```
92/92 [==============================] - 6s 65ms/step - loss: 2.4990e-04
Epoch 140/150
92/92 [==============================] - 5s 59ms/step - loss: 2.0674e-04
Epoch 141/150
92/92 [==============================] - 5s 55ms/step - loss: 2.2341e-04
Epoch 142/150
92/92 [==============================] - 6s 66ms/step - loss: 1.9496e-04
Epoch 143/150
92/92 [==============================] - 5s 56ms/step - loss: 2.0489e-04
Epoch 144/150
92/92 [==============================] - 6s 68ms/step - loss: 2.0465e-04
Epoch 145/150
92/92 [==============================] - 5s 56ms/step - loss: 2.1712e-04
Epoch 146/150
92/92 [==============================] - 6s 61ms/step - loss: 2.1424e-04
Epoch 147/150
92/92 [==============================] - 6s 62ms/step - loss: 1.9352e-04
Epoch 148/150
92/92 [==============================] - 5s 57ms/step - loss: 2.0395e-04
Epoch 149/150
92/92 [==============================] - 6s 68ms/step - loss: 2.0432e-04
Epoch 150/150
92/92 [==============================] - 5s 58ms/step - loss: 1.9505e-04
```

[76]:
```
<keras.callbacks.History at 0x7f0c95ab8850>
```

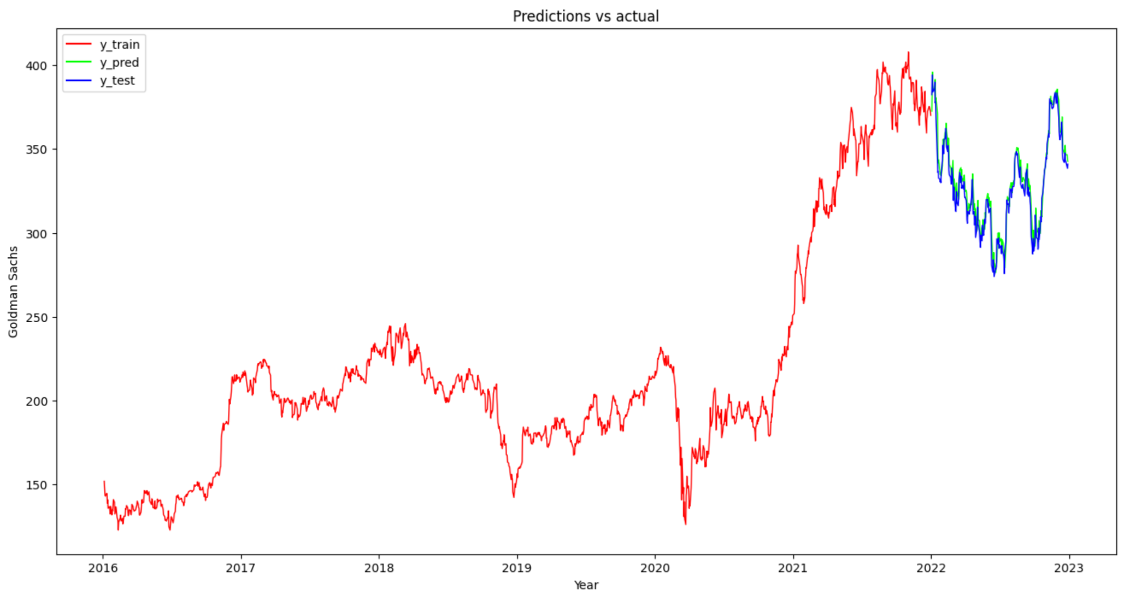
[77]:
```
# Obtain the predicted values
y_pred_scaled = LSTM_modl.predict(x_test)
y_pred = mmscaler.inverse_transform(y_pred_scaled)
y_test_unscaled = mmscaler.inverse_transform(y_test.reshape(-1, 1))
```

```
8/8 [==============================] - 1s 18ms/step
```

[77]:
```

```

[78]:
```
# The date from which on the date is displayed
start = "2016-01-04"

# Add the difference between the valid and predicted prices
actual_train = pd.DataFrame(data_LSTM[:train_data_length]).rename(columns={'Adj␣
↪Close': 'y_train'})
validation_period = pd.DataFrame(data_LSTM[train_data_length:train_data_length␣
↪+ len(y_pred)]).rename(columns={'Adj Close': 'y_test'})

validation_period.insert(1, "y_pred", y_pred, True)
validation_period.insert(1, "residuals", validation_period["y_pred"] -␣
↪validation_period["y_test"], True)
```

```
df_join = pd.concat([actual_train, validation_period])

# Zoom in to a closer timeframe
df_join_zoom = df_join[df_join.index > start]

# Create the lineplot
fig, ax1 = plt.subplots(figsize=(16, 8), sharex=True)
plt.title("Predictions vs actual")
sns.set_palette(["#FF0000", "#00FF00", "#0000FF"])
plt.ylabel("Goldman Sachs", fontsize=10)
sns.lineplot(data=df_join_zoom[['y_train', 'y_pred', 'y_test']], linewidth=1.0,␣
↪dashes=False, ax=ax1)

plt.legend()
plt.show()
```



[79]:

```
# Calculate the log returns of the predicted and test prices
log_returns_pred = np.log(y_pred[:-1]) - np.log(y_pred[1:])
# log_returns_test = np.log(y_test_unscaled[:-1]) - np.log(y_test_unscaled[1:])

# Calculate the Mean Squared Error
mse_log_returns = mean_squared_error(actual_log_return_test_data,␣
↪log_returns_pred)
print(f'Mean Squared Error (MSE) of log returns: {mse_log_returns:.6f}')
```

```
Mean Squared Error (MSE) of log returns: 0.000794
```

[80]:
```
# Calculate the Mean Squared Error
mse = mean_squared_error(actual_prices_test_data, y_pred)
print(f'Mean Squared Error of prices (MSE): {mse:.2f}')
```

```
Mean Squared Error of prices (MSE): 49.37
```

[84]:
```
# Now again if we want to find the average we can loop through this previous␣
↪code to get multiple MSE's and then take the average

mse_LSTM_sim_prices_list = []
mse_LSTM_sim_log_returns_list = []


# set the number of simulations to run
num_simulations = 10 # we have only chosen 5 simulations for the sake of time.␣
↪we could increase this to get a more precise average
for i in range(num_simulations):

  data_LSTM = GS.filter(['Adj Close'])
  train = train.filter(['Adj Close'])
  train = train.filter(['Adj Close'])

  data_unscaled = data_LSTM.values


  # Get the number of rows to train the model on
  train_data_length = len(train)

  # Reshape the array to have one column
  data_reshaped = data_unscaled.reshape(-1, 1)

  # Scale the data
  mmscaler = MinMaxScaler(feature_range=(0, 1))
  np_data = mmscaler.fit_transform(data_reshaped)

  # Set the sequence length - this is the timeframe used to make a single␣
↪prediction
  window = 50

  # The index_Close variable is used to locate the column index of the 'Adj␣
↪Close' feature in the data. This index will be used to extract the true␣
↪values for the prediction.
  index_Close = data_LSTM.columns.get_loc("Adj Close")
  print(index_Close)
  # The train_data_len variable is used to determine how much data to use for␣
↪training the model.
  train_data_len = len(train)
```

```
# The train_data and test_data variables are used to split the data into␣
↪training and testing sets.
train_data = np_data[0:train_data_len, :]
test_data = np_data[train_data_len - window:, :]

# The RNN needs data with the format of [samples, time steps, features]
# Here, we create N samples, sequence_length time steps per sample, and 6␣
↪features
def partition_dataset(window, train_df):
    x, y = [], []
    data_len = train_df.shape[0]
    for i in range(window, data_len):
        x.append(train_df[i-window:i,:])
        y.append(train_df[i, index_Close])

    # Convert the x and y to numpy arrays
    x = np.array(x)
    y = np.array(y)
    return x, y

# Generate training data and test data
x_train, y_train = partition_dataset(window, train_data)
x_test, y_test = partition_dataset(window, test_data)


# Configure the neural network model
def create_model():
  LSTM_modl = Sequential()

  neurons = window

  # Model with sequence_length Neurons
  # inputshape = sequence_length Timestamps
  LSTM_modl.add(LSTM(neurons, return_sequences=True, input_shape=(x_train.
↪shape[1], 1)))
  LSTM_modl.add(LSTM(neurons, return_sequences=False))
  LSTM_modl.add(Dense(25, activation='relu'))
  LSTM_modl.add(Dense(1))

  # Compile the model
  LSTM_modl.compile(optimizer='adam', loss='mean_squared_error')

  return LSTM_modl

LSTM_modl = create_model()
```

```python
LSTM_modl.compile(optimizer='adam', loss='mean_squared_error')


# Training the model
LSTM_modl.fit(x_train, y_train, batch_size=16, epochs=200)


# Get the predicted values

y_pred_scaled = LSTM_modl.predict(x_test)
y_pred = mmscaler.inverse_transform(y_pred_scaled)
y_test_unscaled = mmscaler.inverse_transform(y_test.reshape(-1, 1))


# The date from which on the date is displayed
start = "2016-01-04"

# Add the difference between the valid and predicted prices
actual_train = pd.DataFrame(data_LSTM[:train_data_length]).
↪rename(columns={'Adj Close': 'y_train'})
validation_period = pd.DataFrame(data_LSTM[train_data_length:
↪train_data_length + len(y_pred)]).rename(columns={'Adj Close': 'y_test'})

validation_period.insert(1, "y_pred", y_pred, True)
validation_period.insert(1, "residuals", validation_period["y_pred"] -␣
↪validation_period["y_test"], True)
df_join = pd.concat([actual_train, validation_period])

# Zoom in to a closer timeframe
df_join_zoom = df_join[df_join.index > start]


# Calculate the log returns of the predicted and test prices
log_returns_pred = np.log(y_pred[:-1]) - np.log(y_pred[1:])

# Calculate the Mean Squared Error
mse_log_returns = mean_squared_error(actual_log_return_test_data,␣
↪log_returns_pred)



# Calculate the Mean Squared Error
mse_prices = mean_squared_error(y_test_unscaled, y_pred)


mse_LSTM_sim_log_returns_list.append(mse_log_returns)
mse_LSTM_sim_prices_list.append(mse_prices)
```

```
mean_mse_log_returns = np.mean(mse_LSTM_sim_log_returns_list)
mean_mse_prices = np.mean(mse_LSTM_sim_prices_list)

print("mean LSTM MSE of log returns", mean_mse_log_returns)
print("mean LSTM MSE of prices", mean_mse_prices)
```

```
0
Epoch 1/150
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-84-20fae27bf25c> in <cell line: 9>()
     81
     82   # Training the model
---> 83   LSTM_modl.fit(x_train, y_train, batch_size=16, epochs=150)
     84
     85

/usr/local/lib/python3.9/dist-packages/keras/utils/traceback_utils.py in
 ↪error_handler(*args, **kwargs)
     63         filtered_tb = None
     64         try:
---> 65             return fn(*args, **kwargs)
     66         except Exception as e:
     67             filtered_tb = _process_traceback_frames(e.__traceback__)

/usr/local/lib/python3.9/dist-packages/keras/engine/training.py in fit(self, x,
 ↪y, batch_size, epochs, verbose, callbacks, validation_split, validation_data,
 ↪shuffle, class_weight, sample_weight, initial_epoch, steps_per_epoch,
 ↪validation_steps, validation_batch_size, validation_freq, max_queue_size,
 ↪workers, use_multiprocessing)
   1683                         ):
   1684                             callbacks.on_train_batch_begin(step)
-> 1685                             tmp_logs = self.train_function(iterator)
   1686                             if data_handler.should_sync:
   1687                                 context.async_wait()

/usr/local/lib/python3.9/dist-packages/tensorflow/python/util/traceback_utils.py
 ↪in error_handler(*args, **kwargs)
    148     filtered_tb = None
    149     try:
--> 150       return fn(*args, **kwargs)
    151     except Exception as e:
    152       filtered_tb = _process_traceback_frames(e.__traceback__)
```

```
/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
↪polymorphic_function/polymorphic_function.py in __call__(self, *args, **kwds)
    892
    893       with OptionalXlaContext(self._jit_compile):
--> 894         result = self._call(*args, **kwds)
    895
    896       new_tracing_count = self.experimental_get_tracing_count()

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
↪polymorphic_function/polymorphic_function.py in _call(self, *args, **kwds)
    940       # This is the first call of __call__, so we have to initialize.
    941       initializers = []
--> 942       self._initialize(args, kwds, add_initializers_to=initializers)
    943     finally:
    944       # At this point we know that the initialization is complete (or␣
↪less

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
↪polymorphic_function/polymorphic_function.py in _initialize(self, args, kwds,␣
↪add_initializers_to)
    761     self._graph_deleter = FunctionDeleter(self._lifted_initializer_grap )
    762     self._concrete_variable_creation_fn = (
--> 763         self._variable_creation_fn    # pylint: disable=protected-acces
    764         ._get_concrete_function_internal_garbage_collected(
    765             *args, **kwds))

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
↪polymorphic_function/tracing_compiler.py in␣
↪_get_concrete_function_internal_garbage_collected(self, *args, **kwargs)
    169     """Returns a concrete function which cleans up its graph function." "
    170     with self._lock:
--> 171       concrete_function, _ = self._maybe_define_concrete_function(args,␣
↪kwargs)
    172     return concrete_function
    173

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
↪polymorphic_function/tracing_compiler.py in␣
↪_maybe_define_concrete_function(self, args, kwargs)
    164       kwargs = {}
    165
--> 166     return self._maybe_define_function(args, kwargs)
    167
    168   def _get_concrete_function_internal_garbage_collected(self, *args,␣
↪**kwargs):

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
↪polymorphic_function/tracing_compiler.py in _maybe_define_function(self, args ␣
↪kwargs)
```

```
    394             kwargs = placeholder_bound_args.kwargs
    395 
--> 396             concrete_function = self._create_concrete_function(
    397                 args, kwargs, func_graph)
    398 

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/polymorphic_function/tracing_compiler.py in _create_concrete_function(self, args, kwargs, func_graph)
    298 
    299       concrete_function = monomorphic_function.ConcreteFunction(
--> 300         func_graph_module.func_graph_from_py_func(
    301             self._name,
    302             self._python_function,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/func_graph.py in func_graph_from_py_func(name, python_func, args, kwargs, signature, func_graph, autograph, autograph_options, add_control_dependencies, arg_names, op_return_value, collections, capture_by_value, create_placeholders, acd_record_initial_resource_uses)
   1212         _, original_func = tf_decorator.unwrap(python_func)
   1213 
-> 1214       func_outputs = python_func(*func_args, **func_kwargs)
   1215 
   1216       # invariant: `func_outputs` contains only Tensors, CompositeTensors,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/polymorphic_function/polymorphic_function.py in wrapped_fn(*args, **kwds)
    665         # the function a weak reference to itself to avoid a reference cycle.
    666         with OptionalXlaContext(compile_with_xla):
--> 667           out = weak_wrapped_fn().__wrapped__(*args, **kwds)
    668         return out
    669 

/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/func_graph.py in autograph_handler(*args, **kwargs)
   1187           # TODO(mdan): Push this block higher in tf.function's call stack.
   1188           try:
-> 1189             return autograph.converted_call(
   1190                 original_func,
   1191                 args,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/autograph/impl/api.py in converted_call(f, args, kwargs, caller_fn_scope, options)
```

```
    437     try:
    438       if kwargs is not None:
--> 439         result = converted_f(*effective_args, **kwargs)
    440       else:
    441         result = converted_f(*effective_args)

/usr/local/lib/python3.9/dist-packages/keras/engine/training.py in tf__train_function(iterator)
     13                 try:
     14                     do_return = True
---> 15                     retval_ = ag__.converted_call(ag__.ld(step_function), (ag__.ld(self), ag__.ld(iterator)), None, fscope)
     16                 except:
     17                     do_return = False

/usr/local/lib/python3.9/dist-packages/tensorflow/python/autograph/impl/api.py in converted_call(f, args, kwargs, caller_fn_scope, options)
    375
    376   if not options.user_requested and conversion.is_allowlisted(f):
--> 377     return _call_unconverted(f, args, kwargs, options)
    378
    379   # internal_convert_user_code is for example turned off when issuing a dynamic

/usr/local/lib/python3.9/dist-packages/tensorflow/python/autograph/impl/api.py in _call_unconverted(f, args, kwargs, options, update_cache)
    457   if kwargs is not None:
    458     return f(*args, **kwargs)
--> 459   return f(*args)
    460
    461

/usr/local/lib/python3.9/dist-packages/keras/engine/training.py in step_function(model, iterator)
   1266                 )
   1267             data = next(iterator)
-> 1268             outputs = model.distribute_strategy.run(run_step, args=(data,))
   1269             outputs = reduce_per_replica(
   1270                 outputs,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/distribute/distribute_lib.py in run(***failed resolving arguments***)
   1314       fn = autograph.tf_convert(
   1315           fn, autograph_ctx.control_status_ctx(), convert_by_default=False)
-> 1316       return self._extended.call_for_each_replica(fn, args=args, kwargs=kwargs)
```

```
   1317
   1318   def reduce(self, reduce_op, value, axis):

/usr/local/lib/python3.9/dist-packages/tensorflow/python/distribute/
 ↪distribute_lib.py in call_for_each_replica(self, fn, args, kwargs)
   2893       kwargs = {}
   2894     with self._container_strategy().scope():
-> 2895       return self._call_for_each_replica(fn, args, kwargs)
   2896
   2897   def _call_for_each_replica(self, fn, args, kwargs):

/usr/local/lib/python3.9/dist-packages/tensorflow/python/distribute/
 ↪distribute_lib.py in _call_for_each_replica(self, fn, args, kwargs)
   3694   def _call_for_each_replica(self, fn, args, kwargs):
   3695     with ReplicaContext(self._container_strategy(),␣
 ↪replica_id_in_sync_group=0):
-> 3696       return fn(*args, **kwargs)
   3697
   3698   def _reduce_to(self, reduce_op, value, destinations, options):

/usr/local/lib/python3.9/dist-packages/tensorflow/python/autograph/impl/api.py␣
 ↪in wrapper(*args, **kwargs)
    687       try:
    688         with conversion_ctx:
--> 689           return converted_call(f, args, kwargs, options=options)
    690       except Exception as e:  # pylint:disable=broad-except
    691         if hasattr(e, 'ag_error_metadata'):

/usr/local/lib/python3.9/dist-packages/tensorflow/python/autograph/impl/api.py␣
 ↪in converted_call(f, args, kwargs, caller_fn_scope, options)
    375
    376   if not options.user_requested and conversion.is_allowlisted(f):
--> 377     return _call_unconverted(f, args, kwargs, options)
    378
    379   # internal_convert_user_code is for example turned off when issuing a
 ↪dynamic

/usr/local/lib/python3.9/dist-packages/tensorflow/python/autograph/impl/api.py␣
 ↪in _call_unconverted(f, args, kwargs, options, update_cache)
    456
    457   if kwargs is not None:
--> 458     return f(*args, **kwargs)
    459   return f(*args)
    460

/usr/local/lib/python3.9/dist-packages/keras/engine/training.py in run_step(data)
   1247
   1248           def run_step(data):
```

```
-> 1249                 outputs = model.train_step(data)
   1250                 # Ensure counter is updated only if `train_step` succeeds.
   1251                 with tf.control_dependencies(_minimum_control_deps(outputs)):

/usr/local/lib/python3.9/dist-packages/keras/engine/training.py in train_step(self, data)
   1052         self._validate_target_and_loss(y, loss)
   1053         # Run backwards pass.
-> 1054         self.optimizer.minimize(loss, self.trainable_variables, tape=tape)
   1055         return self.compute_metrics(x, y, y_pred, sample_weight)
   1056 

/usr/local/lib/python3.9/dist-packages/keras/optimizers/optimizer.py in minimize(self, loss, var_list, tape)
    540           None
    541         """
--> 542         grads_and_vars = self.compute_gradients(loss, var_list, tape)
    543         self.apply_gradients(grads_and_vars)
    544 

/usr/local/lib/python3.9/dist-packages/keras/optimizers/optimizer.py in compute_gradients(self, loss, var_list, tape)
    273                 var_list = var_list()
    274 
--> 275         grads = tape.gradient(loss, var_list)
    276         return list(zip(grads, var_list))
    277 

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/backprop.py in gradient(self, target, sources, output_gradients, unconnected_gradients)
   1061                           for x in output_gradients]
   1062 
-> 1063     flat_grad = imperative_grad.imperative_grad(

   1064         self._tape,
   1065         flat_targets,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/imperative_grad.py in imperative_grad(tape, target, sources, output_gradients, sources_raw, unconnected_gradients)
     65         "Unknown value for unconnected_gradients: %r" % unconnected_gradients)
     66 
---> 67   return pywrap_tfe.TFE_Py_TapeGradient(

     68       tape._tape,  # pylint: disable=protected-access
```

```
     69         target,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
↪polymorphic_function/monomorphic_function.py in _backward_function(*args)
    632     def _backward_function(*args):
    633       call_op = outputs[0].op
--> 634       return self._rewrite_forward_and_call_backward(call_op, *args)
    635     return _backward_function, outputs
    636 

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
↪polymorphic_function/monomorphic_function.py in␣
↪_rewrite_forward_and_call_backward(self, op, *doutputs)
    548   def _rewrite_forward_and_call_backward(self, op, *doutputs):
    549     """Add outputs to the forward call and feed them to the grad␣
↪function."""
--> 550     forward_function, backwards_function = self.
↪forward_backward(len(doutputs))
    551     if not backwards_function.outputs:
    552       return backwards_function.structured_outputs

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
↪polymorphic_function/monomorphic_function.py in forward_backward(self,␣
↪num_doutputs)
    481     if forward_backward is not None:
    482       return forward_backward
--> 483     forward, backward = self._construct_forward_backward(num_doutputs)
    484     self._cached_function_pairs[num_doutputs] = (forward, backward)
    485     return forward, backward

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
↪polymorphic_function/monomorphic_function.py in␣
↪_construct_forward_backward(self, num_doutputs)
    524       backwards_graph = func_graph_module.FuncGraph(
    525           _backward_name(self._func_graph.name))
--> 526       func_graph_module.func_graph_from_py_func(

    527           name=backwards_graph.name,
    528           python_func=_backprop_function,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/func_graph.py␣
↪in func_graph_from_py_func(name, python_func, args, kwargs, signature,␣
↪func_graph, autograph, autograph_options, add_control_dependencies, arg_names␣
↪op_return_value, collections, capture_by_value, create_placeholders,␣
↪acd_record_initial_resource_uses)
   1212         _, original_func = tf_decorator.unwrap(python_func)
   1213 
-> 1214       func_outputs = python_func(*func_args, **func_kwargs)
   1215 
```

```
   1216       # invariant: `func_outputs` contains only Tensors,␣
 ↪CompositeTensors,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/eager/
 ↪polymorphic_function/monomorphic_function.py in _backprop_function(*grad_ys)
    515     def _backprop_function(*grad_ys):
    516       with ops.device(None):
--> 517         return gradients_util._GradientsHelper(  # pylint:␣
 ↪disable=protected-access
    518             trainable_outputs,
    519             self._func_graph.inputs,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/gradients_util.py␣
 ↪in _GradientsHelper(ys, xs, grad_ys, name, colocate_gradients_with_ops,␣
 ↪gate_gradients, aggregation_method, stop_gradients, unconnected_gradients,␣
 ↪src_graph)
    693                 # If grad_fn was found, do not use SymbolicGradient even␣
 ↪for
    694                 # functions.
--> 695                 in_grads = _MaybeCompile(grad_scope, op, func_call,
    696                                          lambda: grad_fn(op, *out_grads))
    697               else:

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/gradients_util.py␣
 ↪in _MaybeCompile(scope, op, func, grad_fn)
    327
    328   if not xla_compile:
--> 329     return grad_fn()  # Exit early
    330
    331   # If the gradients are supposed to be compiled separately, we give␣
 ↪them a

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/gradients_util.py␣
 ↪in <lambda>()
    694                 # functions.
    695                 in_grads = _MaybeCompile(grad_scope, op, func_call,
--> 696                                          lambda: grad_fn(op, *out_grads))
    697               else:
    698                 # For function call ops, we add a 'SymbolicGradient'

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/while_v2.py in␣
 ↪_WhileGrad(op, *grads)
    392       body_graph.outputs, body_graph.inputs, grads) if grad is not None])
    393
--> 394   body_grad_graph, args = _create_grad_func(
    395       ys, xs, non_none_grads, cond_graph, body_graph,
```

```
    396         util.unique_grad_fn_name(body_graph.name), op, maximum_iterations

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/while_v2.py in␣
 ↪_create_grad_func(ys, xs, grads, cond_graph, body_graph, name, while_op,␣
 ↪maximum_iterations, stateful_parallelism)
    694     # Note: The returned function does not have `args` in the list of
    695     # `external_captures`.
--> 696     grad_func_graph = func_graph_module.func_graph_from_py_func(

    697         name,
    698         lambda *args: _grad_fn(ys, xs, args, body_graph),

/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/func_graph.py␣
 ↪in func_graph_from_py_func(name, python_func, args, kwargs, signature,␣
 ↪func_graph, autograph, autograph_options, add_control_dependencies, arg_names,␣
 ↪op_return_value, collections, capture_by_value, create_placeholders,␣
 ↪acd_record_initial_resource_uses)
   1212           _, original_func = tf_decorator.unwrap(python_func)
   1213
-> 1214         func_outputs = python_func(*func_args, **func_kwargs)
   1215
   1216         # invariant: `func_outputs` contains only Tensors,␣
 ↪CompositeTensors,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/while_v2.py in␣
 ↪<lambda>(*args)
    696     grad_func_graph = func_graph_module.func_graph_from_py_func(
    697         name,
--> 698         lambda *args: _grad_fn(ys, xs, args, body_graph),
    699         args, {},
    700         func_graph=_WhileBodyGradFuncGraph(name, cond_graph, body_graph,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/while_v2.py in␣
 ↪_grad_fn(ys, xs, args, func_graph)
    752     # after the forward While op has been rewritten in␣
 ↪_resolve_grad_captures.
    753     # TODO(srbs): Mark GradientsHelper as public?
--> 754     grad_outs = gradients_util._GradientsHelper(

    755         ys, xs, grad_ys=grad_ys, src_graph=func_graph,
    756         unconnected_gradients="zero")

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/gradients_util.py␣
 ↪in _GradientsHelper(ys, xs, grad_ys, name, colocate_gradients_with_ops,␣
 ↪gate_gradients, aggregation_method, stop_gradients, unconnected_gradients,␣
 ↪src_graph)
    693               # If grad_fn was found, do not use SymbolicGradient even␣
 ↪for
    694               # functions.
```

```
--> 695                 in_grads = _MaybeCompile(grad_scope, op, func_call,

    696                                          lambda: grad_fn(op, *out_grads))
    697               else:

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/gradients_util.py␣
 ↪in _MaybeCompile(scope, op, func, grad_fn)
    327
    328   if not xla_compile:
--> 329     return grad_fn()  # Exit early
    330
    331   # If the gradients are supposed to be compiled separately, we give␣
 ↪them a

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/gradients_util.py␣
 ↪in <lambda>()
    694                 # functions.
    695                 in_grads = _MaybeCompile(grad_scope, op, func_call,
--> 696                                          lambda: grad_fn(op, *out_grads))

    697               else:
    698                 # For function call ops, we add a 'SymbolicGradient'

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/math_grad.py in␣
 ↪_MulGrad(op, grad)
   1367
   1368   (sx, rx, must_reduce_x), (sy, ry, must_reduce_y) = (
-> 1369       SmartBroadcastGradientArgs(x, y, grad))

   1370   x = math_ops.conj(x)
   1371   y = math_ops.conj(y)

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/math_grad.py in␣
 ↪SmartBroadcastGradientArgs(x, y, grad)
    103     sx = array_ops.shape_internal(x, optimize=False)
    104     sy = array_ops.shape_internal(y, optimize=False)
--> 105     rx, ry = gen_array_ops.broadcast_gradient_args(sx, sy)
    106     return (sx, rx, True), (sy, ry, True)
    107

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/gen_array_ops.py in␣
 ↪broadcast_gradient_args(s0, s1, name)
    770       pass  # Add nodes to the TensorFlow graph.
    771   # Add nodes to the TensorFlow graph.
--> 772   _, _, _op, _outputs = _op_def_library._apply_op_helper(

    773         "BroadcastGradientArgs", s0=s0, s1=s1, name=name)
    774   _result = _outputs[:]
```

```
/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/op_def_library.py in _apply_op_helper(op_type_name, name, **keywords)
    793       # Add Op to graph
    794       # pylint: disable=protected-access
--> 795       op = g._create_op_internal(op_type_name, inputs, dtypes=None,

    796                                  name=scope, input_types=input_types,
    797                                  attrs=attr_protos, op_def=op_def)

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/while_v2.py in _create_op_internal(self, op_type, inputs, dtypes, input_types, name, attrs, op_def, compute_device)
   1043           compute_device=compute_device)
   1044
-> 1045     return super(_WhileBodyGradFuncGraph, self)._create_op_internal(

   1046         op_type,
   1047         inputs,

/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/func_graph.py in _create_op_internal(self, op_type, inputs, dtypes, input_types, name, attrs, op_def, compute_device)
    703       if ctxt is not None and hasattr(ctxt, "AddValue"):
    704         inp = ctxt.AddValue(inp)
--> 705       inp = self.capture(inp)
    706       captured_inputs.append(inp)
    707     return super()._create_op_internal(  # pylint: disable=protected-access

/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/func_graph.py in capture(self, tensor, name, shape)
    770               f"it was defined in {tensor.graph}, which is out of scope")
    771         inner_graph = inner_graph.outer_graph
--> 772       return self._capture_helper(tensor, name)
    773     return tensor
    774

/usr/local/lib/python3.9/dist-packages/tensorflow/python/ops/while_v2.py in _capture_helper(self, tensor, name)
   1223       # Capture in the cond graph as well so the forward cond and body inputs
   1224       # match.
-> 1225       with self._forward_cond_graph.as_default():
   1226         self._forward_cond_graph.capture(tensor_list)
   1227

/usr/lib/python3.9/contextlib.py in __enter__(self)
    117         del self.args, self.kwds, self.func
```

```
    118         try:
--> 119             return next(self.gen)
    120         except StopIteration:
    121             raise RuntimeError("generator didn't yield") from None

/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/func_graph.p␣
↪in inner_cm()
    480     outer_cm = super().as_default()
    481
--> 482     @tf_contextlib.contextmanager
    483     def inner_cm():
    484       """Context manager for copying distribute.Strategy scope␣
↪information."""

KeyboardInterrupt:
```

[84]:
```
# The code above was interupted as it takes a long time to run throguh the␣
↪iterations so the following
# question uses the average found from the previous run
```

compare the average MSE associated with the one-step ahead forecast of each model with the sample variance of the test data:

Upon evaluating the given data, we can derive several insights by comparing the average Mean Squared Error (MSE) associated with the one-step ahead forecast of each model with the sample variance of the test data, which was 746.89 for the price and 0.000349 for the log returns. MSE's close to or lower than the sample variance imply greater accuracy.

First, let us examine the results for price predictions:

GBM single: The model exhibits the highest MSE at 14918.207, signifying that the single-path prediction using Geometric Brownian Motion (GBM) is the least accurate among the considered models. The GBM model's underlying assumptions, which stipulate that stock prices follow a random walk with drift, may not adequately capture the intricate dynamics and non-linear relationships present in stock prices.

GBM Multi: This model has an MSE of 6375.732, which, while relatively high, is lower than the GBM single model. The improvement can be attributed to the employment of multiple paths and averaging them to create a single mean path, thereby reducing the influence of individual path volatility and noise.

The ARIMA model, with an MSE of 2613.416, significantly outperforms the GBM models in predicting stock prices. This result indicates that the ARIMA model, with the specified order (1,1,1), exhibits a higher level of proficiency in forecasting prices compared to the GBM models. ARIMA models offer increased flexibility and are capable of capturing linear dependencies in the data, making them more appropriate for modeling stock prices that exhibit mean-reverting or trending behavior.

The improvement over the GBM models can be attributed to the ARIMA model's ability to consider autocorrelation and partial autocorrelation present in the stock prices, facilitated by the specified order (1,1,1). In contrast, a GBM model, which assumes that stock prices follow a random walk with drift, would be equivalent to an ARIMA model with the order (0,1,0) and would not account for these dependencies.

Upon examining the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots, it is observed that the second lags in both plots lie outside the confidence interval, indicating statistical significance. Consequently, it is plausible that the ARIMA model's performance could be further enhanced by adjusting the order to (2,1,2), thereby accounting for the presence of second lags in the underlying trends.

FFANN: The Feed-Forward Artificial Neural Network (FFANN) demonstrates a considerably lower MSE of 44.966, indicating enhanced accuracy in predicting prices compared to the ARIMA and GBM models. FFANN is capable of capturing complex relationships and nonlinear patterns within the data, which can contribute to superior predictions of stock prices. This is less than the sample variance hence is a better model to predict future patters. This goes for the following ANN.

LSTM: The Long Short-Term Memory (LSTM) model boasts the lowest MSE at 36.62, implying that it offers the most accurate price prediction among all considered models. LSTM models are engineered to capture long-term dependencies in time series data, which may be particularly relevant for predicting stock prices influenced by long-term trends.

These results are explicitly demonstrated through the price evolution graphs which demonstrate how the single GBM model is by far the worst and the ANN models are much more accurate with their predictions.

Next, let us analyze the results for log returns as the question asks us to "compare the performances in predicting the one-step ahead daily stock log-returns rt = log(St/St−1)":

GBM single: With an MSE of 0.000723, this model ranks second-highest among the models, indicating that the single-path GBM model may not be well-suited to capture the short-term fluctuations and volatility present in log returns.

GBM Multi: The model's MSE is 0.000350, reflecting an improvement over the single-path GBM, similar to the price predictions. The utilization of multiple paths and averaging them mitigates the impact of individual path volatility and noise, rendering the model more effective at predicting log returns than the single-path GBM.

ARIMA: The ARIMA model's MSE is 0.000344, marginally better than the GBM Multi model. As previously mentioned, ARIMA models are more appropriate for modeling stock returns that exhibit mean-reverting or trending behavior, which might explain their superior performance in predicting log returns compared to GBM models.

FFANN: This model has an MSE of 0.000433, which is worse than the ARIMA and GBM Multi models but better than the GBM single model. This suggests that the FFANN model may not be as effective at capturing the short-term fluctuations and volatility present in log returns, despite its ability to model complex relationships and nonlinear patterns.

The LSTM model exhibits an MSE of 0.000644 for log returns, which is inferior to the ARIMA, GBM Multi, and FFANN models but surpasses the GBM single model. The FANN models' superiority in predicting the MSE for log returns could be due to the FFANN model being specifically trained to predict log returns, whereas the other models were primarily designed to predict prices. Consequently, the log returns for these models were derived through a transformation of the predicted prices. The FFANN model's specialized training for log returns prediction may contribute to its enhanced accuracy in this domain.

In the above it is hard to argue that any model is particularly better than the sample variance to predict log prices, although the FFNN and GBM multi seem more accurate than the other models.

3 1.3. Python coding: stock portfolios.

[85]:
```
import datetime as dt
# Define the tickers and the training and testing periods
selected = ['AMZN', 'GOOGL', 'JPM', 'PG', 'MA', 'ABT', 'TSLA', 'HD', 'NFLX',␣
↪'MRK']
start = dt.datetime(2020,1,2)
end = dt.datetime(2021,12,30)

# Download the data from Yahoo Finance
yf.pdr_override()
data = pdr.get_data_yahoo(selected, start, end, interval='1d')
portfolio = pd.DataFrame(data['Adj Close'])
```

```
[*********************100%***********************]  10 of 10 completed
```

[86]:
```
# Set the start and end dates for the training and test sets
start_train = pd.to_datetime('2020-1-2')
end_train = pd.to_datetime('2020-12-31')
start_test = pd.to_datetime('2021-1-4')
end_test = pd.to_datetime('2021-12-30')
```

[87]:
```
train_data = portfolio.loc[start_train:end_train]
test_data = portfolio.loc[start_test:end_test]
```

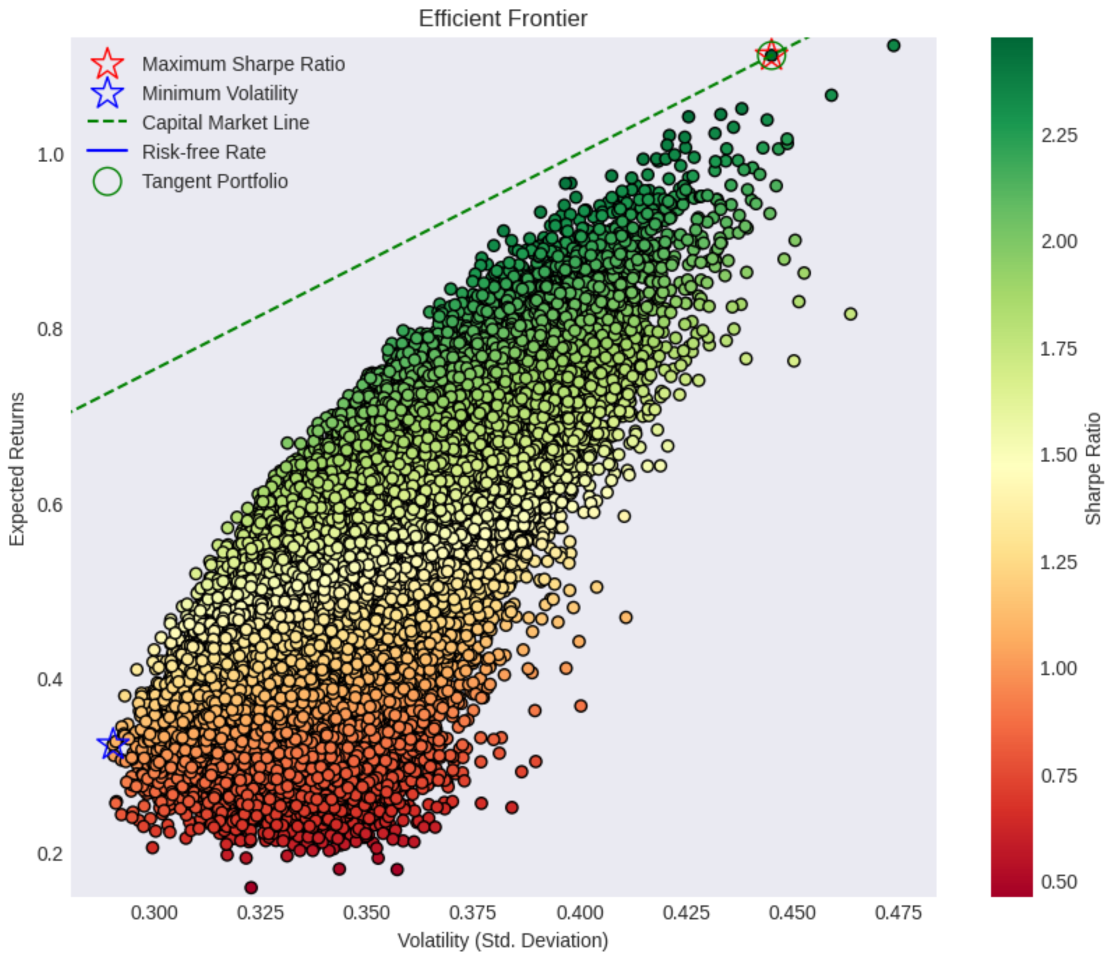
[88]:
```
# calculate daily and annual returns of the stocks
returns_daily = train_data.pct_change() # compute the percentage changes
returns_annual = returns_daily.mean() * 250 # this is the number of trading␣
↪days in the year

# get daily and covariance of returns of the stock
cov_daily = returns_daily.cov()
cov_annual = cov_daily * 250
```

[89]:
```
# calculate Sharpe ratio for each portfolio
port_returns = []
port_volatility = []
sharpe_ratio = []
stock_weights = []

num_assets = len(selected)
```

```
# Set the number of assets and the number of portfolios for simulation
num_portfolios = 100000

# create many random portfolios
for single_portfolio in range(num_portfolios):
    weights = np.random.random(num_assets)
    weights /= np.sum(weights)
    returns = np.dot(weights, returns_annual)
    volatility = np.sqrt(np.dot(weights.T, np.dot(cov_annual, weights)))
    sharpe = (returns - 0.01) / volatility # adjust for risk-free rate which is␣
 ↪1%
    sharpe_ratio.append(sharpe)
    port_returns.append(returns)
    port_volatility.append(volatility)
    stock_weights.append(weights)

# dictionary to store the portfolio data
portfolio = {'Returns': port_returns,
             'Volatility': port_volatility,
             'Sharpe Ratio': sharpe_ratio}

# extend original dictionary to include stock weights
for counter, symbol in enumerate(selected):
    portfolio[symbol+' Weight'] = [Weight[counter] for Weight in stock_weights]

# convert dictionary to pandas dataframe
df = pd.DataFrame(portfolio)

# define order of columns
column_order = ['Returns', 'Volatility', 'Sharpe Ratio'] + [stock+' Weight' for␣
 ↪stock in selected]

# reorder dataframe columns
df = df[column_order]

# set x and y limits to focus on feasible set of portfolios
min_volatility = min(port_volatility)
max_volatility = max(port_volatility)
min_return = min(port_returns)
max_return = max(port_returns)

# plot the efficient frontier
plt.figure(figsize=(10,8))
plt.style.use('seaborn-dark')
plt.scatter(port_volatility, port_returns, c=sharpe_ratio, cmap='RdYlGn',␣
 ↪edgecolors='black')
```

```python
plt.colorbar(label='Sharpe Ratio')
plt.xlabel('Volatility (Std. Deviation)')
plt.ylabel('Expected Returns')
plt.title('Efficient Frontier')
plt.xlim([min_volatility, max_volatility+0.02])



# find the index of the portfolio with highest Sharpe ratio
max_sharpe_idx = np.argmax(sharpe_ratio)

# add red dot for max Sharpe ratio point
plt.scatter(port_volatility[max_sharpe_idx], port_returns[max_sharpe_idx],␣
 ↪marker='*', s=300, label='Maximum Sharpe Ratio', edgecolors='red',␣
 ↪facecolors='none')


# find the index of the portfolio with minimum variance
min_vol_idx = np.argmin(port_volatility)

# add blue dot for min volatility point
plt.scatter(port_volatility[min_vol_idx], port_returns[min_vol_idx],␣
 ↪marker='*', s=300, label='Minimum Volatility', edgecolors='blue',␣
 ↪facecolors='none')


# add CML line
risk_free_rate = 0.01
tangency_return = port_returns[max_sharpe_idx]
tangency_volatility = port_volatility[max_sharpe_idx]

# get slope of CML (use tangent of angle between CML and y-axis)
cml_slope = (tangency_return - risk_free_rate) / tangency_volatility

# define range of x-values for CML line
cml_x = np.linspace(0, max_volatility+0.02)

# calculate corresponding y-values for CML line
cml_y = cml_slope * cml_x + risk_free_rate

# add CML line to plot
plt.plot(cml_x, cml_y, color='green', linestyle='--', label='Capital Market␣
 ↪Line')

# add risk-free rate line
plt.axhline(y=risk_free_rate, color='blue', linestyle='-', label='Risk-free␣
 ↪Rate')
```

```python
# set x and y limits
plt.xlim([min_volatility-0.01, max_volatility+0.01])
plt.ylim([min_return-0.01, max_return+0.01])



# finding the tangent portfolio


risk_free_rate = 0.01

# Calculate the slope for each portfolio
slopes = [(ret - risk_free_rate) / vol for ret, vol in zip(port_returns,␣
 ↪port_volatility)]

# Find the index of the portfolio with the highest slope as this is also the␣
 ↪largest sharpe ratio and is tangential to the CML
tangent_idx = np.argmax(slopes)

# Add a green dot for the tangent portfolio
plt.scatter(port_volatility[tangent_idx], port_returns[tangent_idx],␣
 ↪marker='o', s=200, label='Tangent Portfolio', edgecolors='green',␣
 ↪facecolors='none')

# Update the legend and show the plot
plt.legend()
plt.show()
```



[90]:
```
df.iloc[max_sharpe_idx]
```

[90]:
```
Returns          1.111585
Volatility       0.445102
Sharpe Ratio     2.474904
AMZN Weight      0.041347
GOOGL Weight     0.345832
JPM Weight       0.007773
PG Weight        0.030571
MA Weight        0.059224
ABT Weight       0.059919
TSLA Weight      0.053399
HD Weight        0.007739
NFLX Weight      0.057215
MRK Weight       0.336981
Name: 56642, dtype: float64
```

[91]:
```
df.iloc[min_vol_idx]
```

[91]:
```
Returns          0.322849
Volatility       0.290238
Sharpe Ratio     1.077906
AMZN Weight      0.093309
GOOGL Weight     0.219185
JPM Weight       0.027449
PG Weight        0.000076
MA Weight        0.000012
ABT Weight       0.077709
TSLA Weight      0.232236
HD Weight        0.141497
NFLX Weight      0.203438
MRK Weight       0.005088
Name: 9869, dtype: float64
```

[92]:
```
# Get the tangent portfolio's weights
df.iloc[tangent_idx] # as you can notive this is the same as the maximum sharpe␣
↪ratio portfolio as the CML is tangent at the maximum sharpe ratio
```

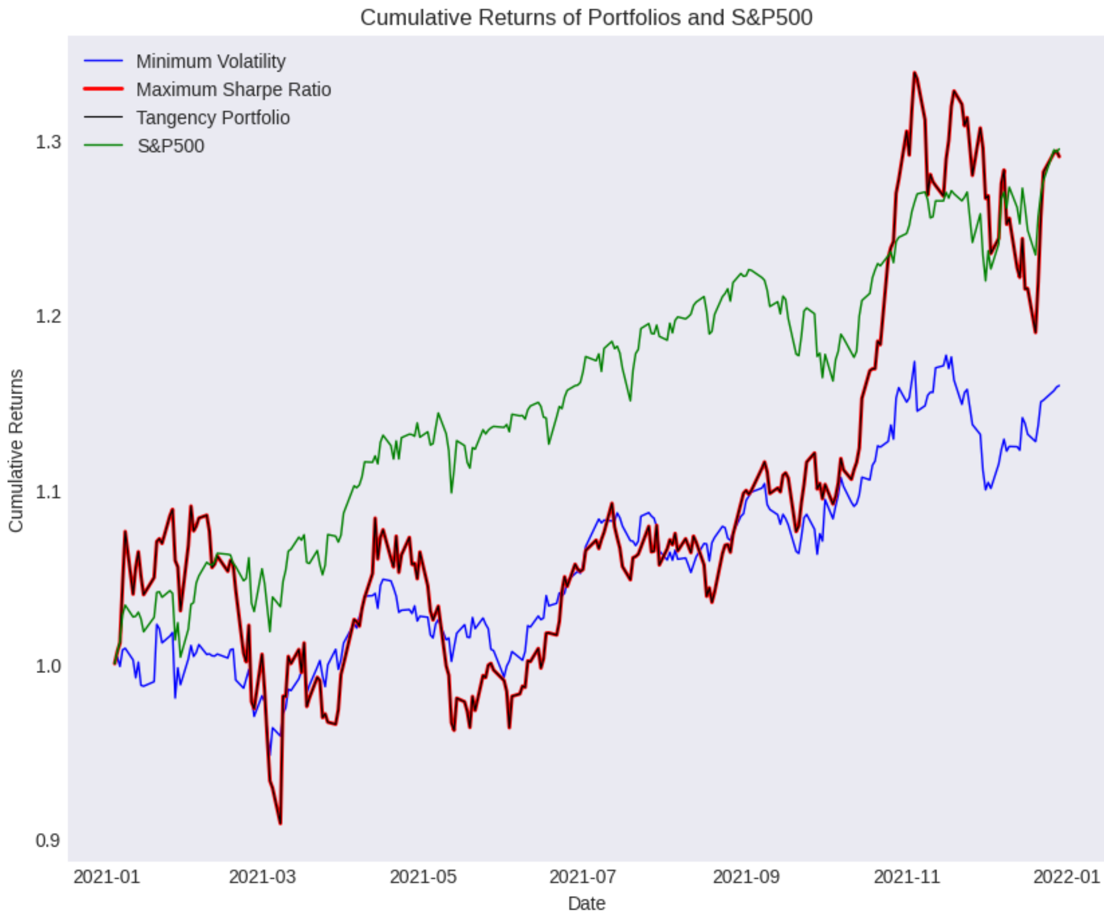
[92]:
```
Returns          1.111585
Volatility       0.445102
Sharpe Ratio     2.474904
AMZN Weight      0.041347
GOOGL Weight     0.345832
JPM Weight       0.007773
PG Weight        0.030571
MA Weight        0.059224
ABT Weight       0.059919
TSLA Weight      0.053399
HD Weight        0.007739
NFLX Weight      0.057215
MRK Weight       0.336981
Name: 56642, dtype: float64
```

[93]:
```
# Get the weight values for the portfolio with maximum Sharpe ratio, minimum␣
↪volatility, and tangent portfolio
max_sharpe_port_weights = stock_weights[max_sharpe_idx]
min_vol_port_weights = stock_weights[min_vol_idx]
tangent_port_weights = stock_weights[tangent_idx]
```

[94]:
```
# Download the historical data for the S&P500 index from Yahoo Finance
sp500_data = pdr.get_data_yahoo('^GSPC', start_test, end_test)

# Filter the S&P500 data to only include the period in the test set
sp500_data = sp500_data['Adj Close']
```

```
# Calculate the cumulative returns of the three portfolios and the S&P500 for
 ↪the test set

# Calculate the cumulative returns for the minimum volatility portfolio
min_volatility_cumulative = (1 + (test_data.pct_change().
 ↪mul(min_vol_port_weights)).sum(axis=1)).cumprod()

# Calculate the cumulative returns for the maximum Sharpe ratio portfolio
max_sharpe_cumulative = (1 + (test_data.pct_change().
 ↪mul(max_sharpe_port_weights)).sum(axis=1)).cumprod()

# Calculate the cumulative returns for the tangency portfolio
tangency_cumulative = (1 + (test_data.pct_change().mul(tangent_port_weights)).
 ↪sum(axis=1)).cumprod()

# Calculate the cumulative returns for the S&P500
sp500_cumulative = (1 + sp500_data.pct_change().fillna(0)).cumprod()

# Plot the cumulative returns of the three portfolios and the S&P500
plt.figure(figsize=(10,8))
plt.plot(min_volatility_cumulative, label='Minimum Volatility',color = 'blue', 
 ↪linewidth=1)
plt.plot(max_sharpe_cumulative, label='Maximum Sharpe Ratio',color = 'red', 
 ↪linewidth=2)
plt.plot(tangency_cumulative, label='Tangency Portfolio', color = 'black', 
 ↪linewidth=0.8)
plt.plot(sp500_cumulative, label='S&P500',color = 'green', linewidth=1)
plt.legend(loc='upper left')
plt.xlabel('Date')
plt.ylabel('Cumulative Returns')
plt.title('Cumulative Returns of Portfolios and S&P500')
plt.show()
```

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



[94]:

Which one is expected to track closer the dynamics of the S&P500 index and why?:

The tangent portfolio is expected to track closest to the dynamics of the S&P500 index because the tangent portfolio is equivalent to the market portfolio for its contingent stocks and the S&P500 is a proxy for said market portfolio. To illustrate this, let's assume there are only 10 stocks in the universe of stocks and one risk-free asset.

Under the Capital Asset Pricing Model (CAPM), market equilibrium necessitates that the aggregate holdings of investors equate to the total supply of assets. Moreover, CAPM presumes that investors are rational and, guided by mean-variance analysis, will choose the same tangency portfolio in conjunction with a position in the risk-free asset.

In this state of equilibrium, each investor maintains identical proportions of assets within their tangency portfolio. Should these proportions diverge, it would signify that at least one investor is not adhering to the unique tangent portfolio, thereby contradicting CAPM's assumptions.

The market portfolio comprises a weighted aggregate of all assets, where the weights correspond to each asset's total market value in relation to the market value of all assets. Consequently, as the sum of each portfolio managers investment in an asset equals the total market value of that asset, and since all investors hold identical proportions of each asset within their tangent portfolios, then these portfolio weightings must reflect the proportionate market capitalization of each respective asset. Thus, the tangent portfolio mirrors the market portfolio in terms of asset allocation.

Hence, the tangent portfolio replicates the market portfolio, as each asset is held in proportion to its market value over the total value of the portfolio. In this context, the tangent portfolio is expected to track closest to the dynamics of the S&P500 index, as the S&P500 serves as a proxy for a market portfolio.

However, in this simplified example the tangent portfolio consists of only 10 stocks, contrastingly, the S&P500 index consists of 500 diverse companies, hence due to the limited scope of the portfolio the tangent portfolio may not provide an accurate representation of such dynamics and so in reality may not track as close as expected.

The tangent portfolio is also by design the portfolio with the greatest sharpe ratio as this would be the point where the CML is tangencial. In this case we also find that the maximum sharpe ratio portfolio is equal to the tangent portfolio and thus we would expect it to track the S&P500 in the same way as the tangent.

Conversely, the minimum volatility portfolio would be expected to exhibit the least resemblance to the S&P500 index. This is primarily because the minimum volatility portfolio focuses on reducing portfolio volatility, which likely results in a portfolio composition that significantly differs from that of the S&P500 index. The construction of the minimum volatility portfolio places emphasis on lower volatility assets, causing a trade-off with lower potential returns. In contrast, the S&P500 index is not designed with such constraints resulting in returns that over time would display higher volatility alongside possibly higher returns than the minimum volatility portfolio.

This would be most prominent in times of tumultuous markets as experienced between 2020 and 2021. For instance, during the bull market of 2021, the S&P500 index experienced substantial gains as investors drove up stock prices in anticipation of future growth. A minimum volatility portfolio, however, might not appreciate as rapidly or to the same extent, as its design prioritizes mitigating downside risk, potentially leading to limited participation in market upswings. Similarly, during the bear markets over the period, a minimum volatility portfolio may not decline as steeply as the S&P500 index, owing to its emphasis on minimizing portfolio volatility and downside risk.

4 Option pricing with the binomial model

[94]:

```

```

Risk neutral

The stock price is currently S0 = £1 and it can either increase to S0 * Zup in the upward state or S0 * Zdown in the downward state. As Zup = 3/2 and Zdown = ½ we have the following

S1up = £1 * 3/2 = £3/2

S1down = £1 * 1/2 = £1/2

In this scenario the probability of an upward state is p and a downward state is q. We can derive the probability of the stock as the possible returns on the stock are either:

(S_0×Z_up)-S_0)/S_0 =Z_up-1

Or

(S_0×Z_down)-S_0)/S_0 =Z_down-1

Therefore, due to the arbitrage free market the expected return of a stock must equal the risk free return, this combined with the assumption that there are only two possible future states, the upward state (U) and the downward state (D) giving us the following conditions:

p(Z_up-1) + q(Z_down-1) = r

p + q = 1

then as q= 1-p

pZ_up+(1-p) Z_down=1+r

p=((1+r)-Z_down)/Z_up-Z_down

q=1-p

Therefore, as r = 5%, p= ((1+5%)-0.5)/(1.5-0.5)=0.55 and hence q = 0.45

Now let us find the payoffs of the upward state and the downward state. The payoff would be the maximum of (S1 − K) and 0, as if S1 went below K then the option would not be exercised and you would not receive a payoff of the option. Given K = 5/4

Payoff in up-state = 3/2 − 5/4 = 0.25 (in the money)

Payoff in down-state = 0 (out the money)

The expected options payoff is therefore:

expected payoff=0.55×0.25+0.45×0=0.1375 The price of the option is then the expected payoff discounted at the risk free rate resulting in the price of the option as:

price of option= 0.1375/(1+0.05)=0.13095

Replicating Portfolio

We can also price this option through the portfolio replication approach. This also adheres to the no arbitrage market as if an option can replicate the expected returns of a portfolio of stocks and bonds then these two assets must be priced the same otherwise one would be able to exploit an arbitrage opportunity. Therefore the current value of the portfolio must equal the price of the option.

Hence we need to create a portfolio that replicates the payoffs of the call option. The portfolio set up is as follows:

We take position _s of shares S

We take position _b of risk free bonds

If this constructed portfolio replicates the option payoff then we can conclude that its current value should be equivalent to the current price of the option (C).

C= S_0 _s+S_0 _b

Therefore as we have the following:

S1up = £1.5 B1up = £1.05

S1down = £0.5 B1down = £1.05

Payoff of up state = £0.25 Payoff of down state = £0

Then we can find the weights of the stock and bond through:

1.5 _s+1.05 _b=0.25 0.5 _s+1.05 _b=0

1.5 _s+1.05 _b-0.5 _s-1.05 _b=0.25

_s=0.25

_b=(-0.5(0.25))/1.05=-0.119

Therefore the present value of the portfolio and thus the price of the option is:

C= 1(0.25)+1(-0.119)=£0.131

[95]:
```
# Risk neutral method


size_of_up_move = 1.5
size_of_down_move = 0.5
S0 = 1
strike_price = 1.25
rf = 0.05
piU = (1+rf-size_of_down_move)/ (size_of_up_move - size_of_down_move)
probability_of_up_move = piU
probability_of_down_move = 1-piU

payoff_U = S0 * size_of_up_move - strike_price
payoff_D = 0 # the payoff from the down move is 0 because we would not use the␣
↪option in this case as S1 = 1 * 0.5 = 0.5 which is less than £1.25

# price of the option should be the discounted value of the expected gain

expected_gain = probability_of_up_move * payoff_U + probability_of_down_move *␣
↪payoff_D

discounted_expected_gain = expected_gain/(1+rf)

discounted_expected_gain
```

[95]: 0.13095238095238096

[96]:
```
# replicating portfolio method

# if the portfolio (of stocks and bonds) is able to replicate the payoff of the␣
↪option under no arbitrage then the current value of the portfolio must equal␣
↪the price of the call option
# therefore we want to create a portfolio that replicates such an option
```

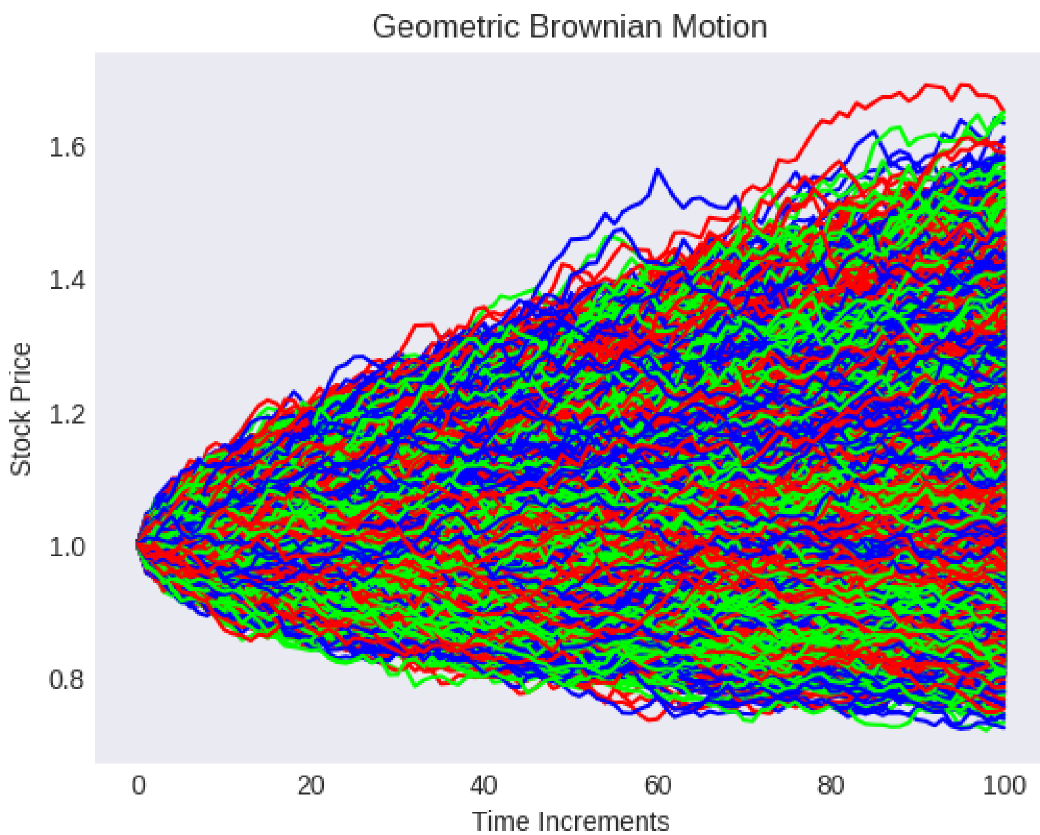
```
"""

# as stated previously the payoff of an option with an upward movement would be␣
↪0.25 (payoff_U = size_of_up_move - strike_price)
# and the option payoff of a downward move would be 0. the price of a bond at␣
↪time B0 is 1 and in the following period it is B1 = B0 * (1+0.05)

# Therefore in the following period:

# upward move

alpha_s * 1.5 + alpha_b * 1.05 = 0.25

# downward move

alpha_s * 0.5 + alpha_b * 1.05 = 0


alpha_s * 1.5 + alpha_b * 1.05 - alpha_s * 0.5 - alpha_b * 1.05 = 0.25

alpha_s = 0.25

alpha_b = -0.119

"""

# Therefore the price of the option should be equal to a current portfolio of␣
↪stocks and bonds with the above weights

alpha_s = 0.25

alpha_b = -0.119

S0 = 1

B0 = 1



price_of_option = alpha_s * S0 + alpha_b * B0

price_of_option
```

[96]: 0.131

option pricing with the Black-Scholes model

[97]:
```
# Finding the option price by creating many possible future price paths and
↪thus finding the value of the option from the times the predicted prices are
↪above the strike


S = 1 #stock price S0
K = 1.1 # strike
T = 100 # time to maturity
r = 0.001 # risk free risk in annual %
mean = 0.002 # annual dividend rate

stdev = 0.01 # annual volatility in %
drift = r - (0.5 * stdev**2)
steps = 100 # time steps
dt = T/steps
N = 100000 # number of trials

# generates a simulation of stock prices for a financial asset over predicted
↪time periods for N simulations.
sim_prices_multi = np.zeros((steps+1, N))
sim_prices_multi[0,:] = S

for i in range (0,N):
    for j in range (1,steps+1):
        sim_prices_multi[j,i]=sim_prices_multi[j-1,i]*np.exp(drift +
↪stdev*sqrt(dt)*np.random.normal(0,1))



plt.plot(sim_prices_multi)
plt.xlabel("Time Increments")
plt.ylabel("Stock Price")
plt.title("Geometric Brownian Motion")
```

[97]:
```
Text(0.5, 1.0, 'Geometric Brownian Motion')
```



[97]:

```

```

[98]:

```
payoffs = np.maximum(sim_prices_multi[-1]-K, 0) # calculating the payoffs, if␣
↪the price is below K then you would not use the option and thus have a␣
↪payoff of 0
option_price = np.mean(payoffs)*np.exp(-r*T) #discounting back to present value
option_price
```

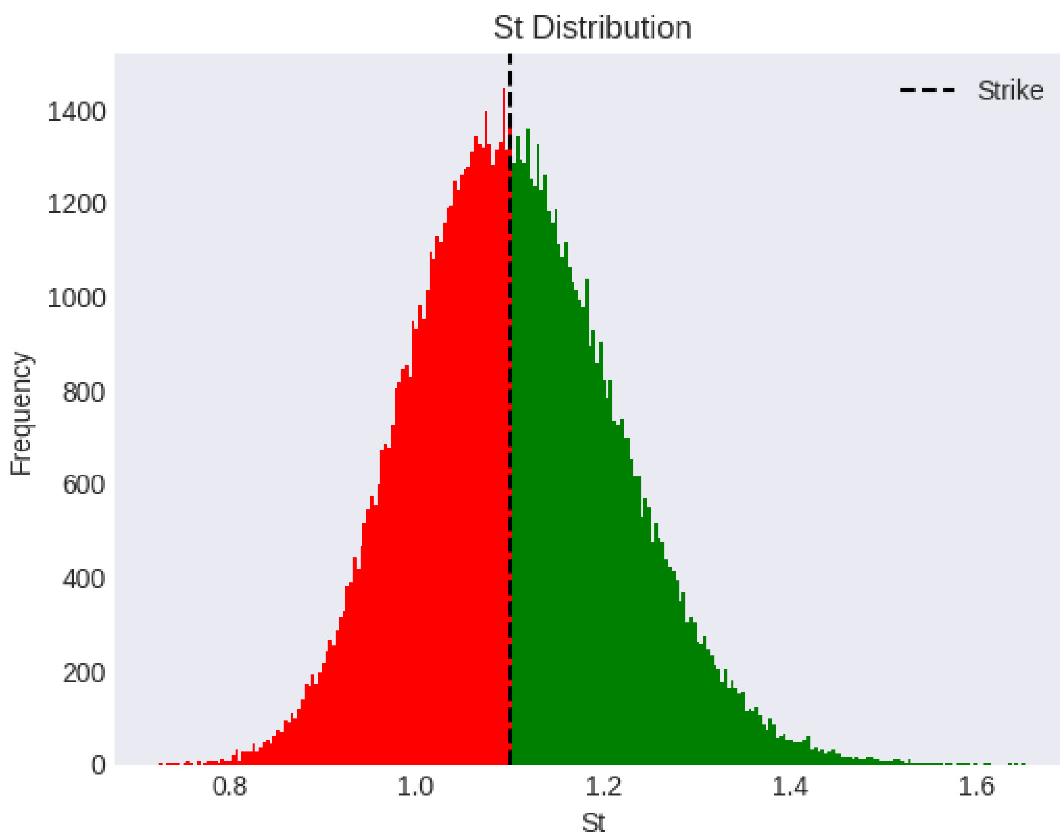
[98]: 0.0424908914112343

[99]:

```
# Plot the histogram of simulated prices at maturity date
n, bins, patches = plt.hist(sim_prices_multi[-1],bins=250);
# Set the color of the bars based on whether the price is greater than the␣
↪strike price or not
for edge, bar in zip(bins, patches):
    if edge > K:
        plt.setp(bar, 'facecolor', 'green')
    else:
        plt.setp(bar, 'facecolor', 'red')
```

```
# Plot graph for visualisation
plt.axvline(K, color='black', linestyle='dashed',label="Strike")
plt.title("St Distribution")
plt.xlabel("St")
plt.ylabel('Frequency')
plt.legend()
```

[99]: <matplotlib.legend.Legend at 0x7f0c32cc2ac0>



[100]:
```
# Auxiliary function for d_one risk-adjusted probability
def d11(S, K, T, r, stdev):
    return (np.log(S0/K) + (r + 0.5 * stdev**2)*T) / (stdev * np.sqrt(T))

# Auxiliary function for d_two risk-adjusted probability
def d21(d1, T, sigma):
    return d1 - stdev * np.sqrt(T)
```

[101]:
```
def black_scholes(S0, K, T, r, stdev):
    d_one = d11(S0, K, T, r, stdev)
    d_two = d21(d_one, T, stdev)
```

```
    return S * norm.cdf(d_one) - np.exp(-r * T) * K * norm.cdf(d_two)
```

[102]:
```
S0 = 1 # current spot price is 1
K = 1.1 # The strike price is 1.1
r = 0.001
T = 100
stdev = 0.01

Black_scholes_option_price = black_scholes(S, K, T, r, stdev)
Black_scholes_option_price
```

[102]: 0.04216744835361197

[ ]:
```
```