Bull Market or Bear Market: Time Series Price Prediction for Q1 2024:

John Vincent Deniega, Ravita Kartawinata, Gabi Rivera

Master of Science in Applied Data Science, University of San Diego

Author Note

We have no known conflict of interest to disclose.

Correspondence concerning this work should be addressed to the authors above. Email: jdeniega@sandiego.edu, rkartawinata@sandiego.edu, gabriellarivera@sandiego.edu

Background Information

Using time series analysis on major market events, such as stock price movement and what factors influence price direction, presents a target-rich opportunity to use models and data-driven methods to inform investment decisions. It is not only beneficial to retail investors, but also governmental and financial organizations that may have a vested interest in garnering the predictive insight borne by these methods. The most recent trend as of this report of downward stock price movement was due to the 2022 inflation crisis. For context, in a Board of Governors of the Federal Reserve System's International Finance Discussion Paper, Ammer (1994) holds that higher inflation is contributes to both lower stock dividends and lower equity returns. In addition, as quoted in Kiley (2023), the word "recession" becomes abundantly searched on the internet typically at the beginning of each prolonged downturn in economic activity. As expected, this search trend was captured during the start of the great recession which was influenced by the housing market crash in 2008 (Hudomiet et. al., 2011), which was then followed by the 2020 COVID-19 recession. This work aims to study how to predict future stock market price movements with a particular focus on the COVID-19 recession due to its significant market-changing dynamics that shifted the global financial ecosystem. Understanding that market sentiment directly drives supply and demand behavior as a result of perceptions on inflation and recession, there may be time series patterns that can be extracted and analyzed in order to capture the overall zeitgeist of the market before it is reflected so in the price.

There have been notable literary datasets derived from 42 recessions in 14 countries using quarterly periods that have high potential for correctly forecasting the next recession (Kroencke, 2022). Stock prices generally declined significantly at about 30% at the start of recession while dividends fell on average by 13% (Kroencke, 2022). Prices of stocks seemed to fall further compared to real dividends during recession periods with the stock price variance behaving relatively the same as dividend growth variance (Kroencke, 2022). Another likely indicator of recession would be the timing of price-dividend ratio drop which happens in two quarters before recession ensues at around 5.6% (Kroencke, 2022). There is also recession variance ratio to look at which is the “recession variance over the pre-recession variance” (Kroencke, 2022). For price changes, the recession variance ratio goes up 2.1-fold which is relatively similar to dividend growth at 1.7-fold (Kroencke, 2022). In this manner, price variance may be a contemporaneous predictor or indicator of how much that period reflects the price variance of known recession periods.

As of this report, the US stock market is experiencing significant volatility into the beginning of the fourth quarter of 2023. As such, using these higher variance price movements provide another opportunity to train models and methods to positive class cases to where we may predict future movement and make data-driven intelligence to inform investment decisions. Through deeper understanding of the underlying characteristics of the markets as a time series and the present environment across multiple endogenous and exogenous dimensions, we may be able to decompose price movements by price level, price trend, quarterly seasonality, and noise, and then identify significant insights from that decomposition. Overall, the goal is to predict with sound data science principles where the US stock market will be by the beginning of the first quarter of 2024 to ultimately manage risk and build capital.

Overview of Main Points

Key components of the stock market include stock exchanges, such as the New York Stock Exchange (NYSE) and NASDAQ, and various financial instruments like equities and exchange-traded funds (ETFs). In this case, the team analyzes the State Street Global Advisors’

Standard and Poor's 500 ETF (SPY) and Amazon.com Incorporated (AMZN) stock as time series to understand and predict the market. Fluctuations of the market often occurred on expected Monday through Friday trading days (excluding holidays). Therefore, the team aims to deploy methods such as moving averages, simple smoothing, exponential smoothing, ARIMA and logistic regression. Additionally, neural networks will be investigated to capture unique patterns derived from the data through transformation. Exploratory Data Analysis (EDA) is based off of historical and derived empirical data to determine predictors for the forecasting model. Throughout the ADS506 course, other methodology would be included in our analysis.

Key Findings

The dataset originated from a real-time Yahoo! Finance-based Application Programming Interface (API) in the Python programming language. The API ticker function retrieves comprehensive daily logs of the stock market movements and metadata. The API returns a Pandas DataFrame with multi-level column names from opening prices, high, low, and closing prices as well as volume. The data types associated with these attributes is a class of a one-dimensional array.

Preliminary EDA findings include the time series of SPDR S&P 500 ETF's stock price exhibiting non-stationarity over the period of six months from May 2023 to November 2023 using the Augmented Dickey-Fuller test statistic method. Upon applying decomposition on the data, the trend and seasonality are considered before providing it as an input into the models.

AMZN was also explored over a trailing 5-year period. The original closing price time series fluctuates between a range of $60 to $190 per share. There was a notable positive trend from 2020 to mid-2021 followed by a negative trend up to 2023's rebounding positive trend. Similar to SPY, AMZN time series displays non-stationarity. The application of LOESS (Locally

Estimated Scatterplot Smoothing) decomposition revealed a 5-year downward concave trend with undetectable seasonality. Anomaly detection to account for noise factor will be applied to residuals and evaluated followed by model fitting and forecasting. This suggests that at this level, AMZN does not have a strong seasonality component.

Data Preprocessing

Data for the respective time series were pulled from Yahoo Finance Library at 5-year span. No missing data values were observed during initial time series plotting approach on market days. However, for continuity of the time series missing values still occurred on weekends and holiday dates, as expected. These missing values on non-market days were imputed from their most recent valid market day. Then, the time series datasets were specified as a daily frequency in order to be used and fed into time series exploratory methods and models.

Exploratory Data Analysis

Determination of stationarity status was the first step of the platform process. Augmented Dickey-Fuller (ADF) test was employed and based off of from the p-values greater than .05 significance level results for SPY and AMZN, the time series datasets were both determined not stationary as depicted in Figures 1a and 2a from the original time series plot. Figures 1a and 2a also shows the next approach which was to subject the time series to (STL) Seasonal-Trend Decomposition using Locally Estimated Scatterplot Smoothing (LOESS) regression. SPY and AMZN’s trend and seasonal components were clearly parsed in both cases. During transformation to remove the trend and seasonal elements for model exploration, first degree differencing of the series was utilized. ADF tests of p-values lower than 0.05 significance level confirms that the time series were converted to stationary datasets. Figures 3a and 3b presents the autocorrelation (ACF) and partial correlation (PACF) plots of SPY and AMZN. For SPY, ACF suggests lags at 1, 2, 3, 8, 9, 11, 14, 21, 58, 63, and 70 periods can be used to explore autoregression *p* parameter value and PACF suggest lags at 1, 2, 3, 8, 9, 11, 14, 21, 58, 63, and 70 periods as candidates for autoregression *p* parameter value. For AMZN, ACF suggests lags at 1, 6, 10, 20, 31, and 32 periods can be used for the moving average *q* parameter value and PACF suggests lags at 1, 6, 10, 20, 31, 32 periods as candidates for the AR *p* parameter value.

As shown in Figures 1b-d and 2b-d, detection of anomaly through the use of STL regression was performed for SPY and AMZN. The residual was taken and subjected to ±3 standard deviation threshold (Figures 1c and 2c) to reveal outliers or anomalous price volatility from the original stock curve (Figures 1d and 2d). For SPY, anomalies were detected during the 2 most recent stock market negative movements in 2020 and 2022. Red marks outside of the 99.7% residual's normal distribution were identified during the times the time series resembled outlier-like volatility. For AMZN, it is noticeable that two anomalies were registered during the 2020 recession when the overall stock market was in dramatic decline. Amazon price performed better during that recession period compared to majority of the market (as SPY) during that period. Throughout calendar year 2022, there appears to have been an increase in the presence of high standard deviation price movements relative to other years in the 5-year period. Overall, analyzing the time series' price distribution provides a good way to detect relative market volatility.

Figures 1e and 2e show additional data exploration on the weekly behavior of the two time series. SPY exhibits central tendency for positive price action on Mondays with price average of $0.96 and median of $0.86. Standard deviation is recorded at $2.09 for that day. Therefore, Mondays are considered the least volatile day of the week for SPY500. In contrast, Thursday price action shows central tendencies of -$0.07 average price, $0.24 median price, and

$3.72 standard deviation. Thursdays are considered to be the most volatile day of the week for SPY with mixed results on positive or negative central tendencies. For AMZN, Tuesdays are the least volatile at -$0.10 average price, $0.05 median price, and $1.58 standard deviation. Mixed central tendencies in mean and median are inconclusive. The following day of the week, Wednesday, exhibits -$0.05 average price, $0.11 median price, and $2.22 standard deviation. For AMZN, this is interpreted as Wednesday being the most volatile day of the week on price action with mixed central tendencies based on average and median price.

Modeling

Selecting Modeling Techniques

Daily stock market logs were mined at the time of API call. Data and associated data types that were retrieved consisted of stock prices (numerical float), volume (integer), and time (Pandas datetime object). An additional variable of interest was derived from stock prices to indicate whether or not the daily price for the time period queried resulted in a positive net increase in price (binary). This derived variable serves as the dependent *'y'* variable for a logistic regression model to be trained to provide a forecast targeting price increase logic.

In order to accomplish this, additional variable predictors were derived as intermediates in the process of fitting the logistic regression model to accomplish this binary outcome. These new predictors are *'open_close'* as well as *'high_low'* which are used to represent the difference between the two original predictors as a combined item in both cases. The *'open_close'* stock price was selected as the most complete predictor of the period that smoothed out intraday price fluctuations. This thereby simplified the numerical difference into a binary outcome field, *'positive'*, as either *'0'* or *'1'* with the latter indicating that the price increased during that discrete period. The binary transformation is used intently to capture the closing stock market price behavior as positive signifying up and negative to down. The time series data – now with additionally constructed features – were then differenced in order to introduce stationarity into the resulting time series for modeling. DataFrames at lag periods one through five were separately constructed for their respective logistic regression model in order to select the best performing lag period and other parameters against the validation set.

In comparison to other methods, the original time series data types pulled from the *yfinance* API were subjected to first-ordered differencing of the closing price predictor before feeding the data into Autoregressive Integrated Moving Average (ARIMA), Simple Exponential Smoothing (SES), and Advanced Exponential Smoothing (AES) methods. The assumption is that the time series exhibits stationarity to remove the influences of both trend and seasonality from the methods. The goal of these aforementioned methods is to predict the closing stock prices at a given target future date solely based off of historical data. Inspection of the SPY data's autocorrelation plot in Figure 3a revealed potentially significant moving average orders at periods 0, 1, 2, 11, 13, 14, 21, 37, and 40. Subsequent inspection of the SPY data's partial autocorrelation plot in Figure 3a revealed potentially significant autoregression orders at periods 1, 2, 3, 8, 9, 11, 14, 21, 58, 63, and 70. This information was key to determining which parameters to iterate through in order to discover the highest performing parameters for lag and moving average orders of ARIMA and AES's individual models. Same goes with Amazon, periods 1, 6, 10, 20, 31, 32 from ACF and PACF were used to determine the *p*,*d*,*q* parameters of ARIMA.

Generating a Test Design

As mentioned earlier, both model-based and data-driven methods (e.g. ARIMA & AES) were used to develop price predictors and logistic regression as well as neural network models to represent stock market movement through price prediction and binary classification. For the price prediction objective, statistical measurements such as root mean square error (RMSE) and mean absolute percentage error (MAPE) were used to evaluate predictive performance between model predictions and truly observed values due to their interpretable values in being in the same unit as the original series as well as a relative proportional measurement as a percentage of the error, respectively.

Additionally, in order to evaluate with consideration for both fitness of the data with complexity of the method used, Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) were considered when selecting the best method to deploy on unseen data. Of these two criteria, a lower number of parameters and a higher log likelihood of fitting the data aim to mitigate the effects of overfitting a model to the training data.

For binary forecasting, a confusion matrix provided performance metrics for the purpose of evaluating the predictive performance of the classifier against the validation data. Precision, sensitivity, F1 score, and accuracy were included in the confusion matrix to assist in different models matching different investment risk profiles. For more conservative risk profiles, high precision models may be preferred due to their focus on higher probability positive price movements. For more risk-tolerant and higher return risk profiles, high accuracy models may be preferred so as to optimize for higher recall on positive signals, but counter-balanced with the higher specificity on non-positive signals (assuming equally weighted costs on both positive and non-positive periods).

Building the Models

SES, AES, ARIMA, logistic regression, and neural network forecasting methods were explored throughout the model building process. SES and ARIMA were omitted out for their insignificant predictive performance. For SPY, the data-driven AES method exhibited the highest performance based on RMSE and MAPE among the price predictor models. Among binary forecasters, Cross-sectional MLP (Neural Network) Model exhibited the highest performance based on confusion matrix results. With respect to their respective parameters for SPY, AES shown in Figure 4c was set to have multiplicative damped trend, additive seasonality, three periods per season, and a heuristic initiation method. Fit was set at 0.1 for both smoothing level and trend parameters. These were determined by running AES through multiple combinations of each parameter such that for both trend and seasonal parameters '*add, mul, additive, multiplicative, and none*' were iterated through to find the combination that yielded the best performing metrics. The same process was performed for parameters in damped trend, seasonal period length, initiation methods, and multiple fits for SPY. Then, each parameter was subjected to predictive performance criteria sequentially. The validation dataset used for the calculation of the performance criteria is set from the last year period and ending on November 27, 2023. The results are further discussed under assessment and evaluation.

As for SPY logistic regression, the *'positive'* field was set as the dependent outcome variable. The stock price predictors along with the new predictors derived by lagged periods were differenced at three periods before feeding into the logistic regression model. Then, validation data was set to the next 200 periods after the training model, resulting in the confusion matrix found in Figure 4d. Figure 4e shows the resulting confusion matrix of the cross sectional MLP (Neural Network) model on validation data. By iterating through the different possible parameter options, those with the highest performing accuracy were selected and collated to build the highest performing model based on hidden layer sizes, activation methods, solver methods, and maximum iteration for convergence. The tanh activation function, hidden layers of

3 and then 2, 2000 maximum iterations, and stochastic gradient descent as the solver method were identified as the highest performing parameter values for the MLP.

For AMZN, auto-ARIMA and ARIMA (Figures 5a-b) were ruled out due to low performance statistics in RMSE and MAPE. The AES trend and initialization method was subjected to different iterations of additive and multiplicative as well as None, estimated, heuristic, and legacy-heuristic options, respectively. Generally, either no trend and heuristic method parameters performed the best on smoothing. Multiplicative seasonality parameters for AES were also selected as optimal. The forecast was set to 252 periods in the future, with that number being the typical number of trading days the market is open in a given year. The model fit smoothing level and trend were both set to 0.5 to correct for the model's visual location against the training data using the plot as shown in Figure 5c. AMZN's logistic regression model building approach was the same to SPY written above with the confusion matrix result shown in Figure 5d. The final model for AMZN was recurrent neural network (RNN) forecast model as shown in Figure 5e. The time series data was first normalized using min-max scaling before feeding the dataset into the model. Then the data frame was partitioned for training and test datasets and run into sequential model with simple RNN and dense method added. After fitting the model and generating a forecast, the results were transformed back to their original scale using inverse transformation on the min-max scaler.

Model Assessment and Evaluation

Figure 4a shows SPY SES model forecast with poor predictive curve spanning across the trailing year range. The AIC score was recorded at 3042 while BIC was recorded at 3051, which is to be compared with models throughout this discussion. RMSE was recorded at 31.71 and

MAPE was recorded at .06. This may be interpreted as an error of $31.71 USD and a 6% price deviation.

Figure 4b shows SPY ARIMA model forecast performing poorly in predicting the validation set similar with SES plot. The forecast curve flattened out throughout the last year prediction range. For performance metrics review, optimal '*pdq'* parameters of 14, 1, 1, respectively, yielded an AIC of 10377 and a BIC of 10466. The RMSE was recorded at 418.23 and the MAPE was recorded at 1.0. This may be interpreted as an error of $418.23 USD and a 100% price deviation, suggesting that even with parameter optimization, ARIMA may be unfeasible for price prediction purposes. For AMZN, Figures 5a-b shows the plot for auto-ARIMA and ARIMA with parameters p,d,q manually iterated for best performing outcome. Both models performed poorly as illustrated with a flat forecast.

As shown in Figure 4c, the SPY AES method was able to forecast the exponential pattern compared to the validation data. Although, the trend and seasonality patterns were not explicitly reflected to also match the actual validation results it appears that the smoothing sufficiently compensated for increased predictive performance. Among all of the price prediction models, the data-driven AES method performed the best against the validation set with half of the error of SES for only a one-fifth higher information criterion with AIC at 3884 and BIC at 3923. The RMSE was recorded at 15.06 and the MAPE was recorded at .03. This may be interpreted as an impressive minimal error of $15.06 USD and a 3% price deviation as of November 27, 2023's data retrieval depicted in Figure 10. For AMZN, the AES model performed significantly better compared to ARIMA. As shown in Figure 5c, the weekly price action closing prices are captured. However, AES is not able to forecast the steep incline of the validation dataset, which is to be expected as an inherent feature of the smoothing method absent of seasonality.

Figure 4d depicts SPY's logistic regression model with resulting confusion matrix shown in Figure 9. The overall accuracy is recorded at .62 with precision at .77 and sensitivity at .61. This may be interpreted as a model that is likely best suited for more conservative risk profiled investors due to its higher precision performance. For AMZN, logistic regression performed poorly at around 51% accuracy, which is effectively similar to a random predictor model and should be ruled out or further reinvestigated.

Figure 4e shows SPY's cross-sectional MLP (Neural Network) confusion matrix with recorded accuracy of .93. Recurrent Neural Network was performed to forecast AMZN. MAPE score was recorded at .02 with MSE of 8.2. Looking at Figure 5e, RNN forecast model outperformed all the methods used for AMZN as it was able to predict the validation dataset as closely as possible.

Conclusion

Time series analysis reveals that through modeling and data-driven methods, it is possible to predict within 20% MAPE future price movements when applied to a market in the form of the S&P 500 or to an individual stock in Amazon.com Inc. Neural networks were especially effective at predicting following positive or negative days with specific lags at 3 days performing optimally on validation data. This is likely due to the increased weight that buyers and sellers place on more recent stock market data, where older data and longer historical timeframes rapidly decay in their usefulness as a predictor. Finally, we understand that market sentiment directly drives stock prices as a result of future market outlook rather than a purely-data driven approach including financial and economic data. In the future, additional work should be made to capture sentiment using large language models or text analysis of news feeds and message boards in order to capture the overall zeitgeist of the market and lead market predictions.

Figure 1a

STL Decomposision of SPY500 Time Series Using LOESS Technique.

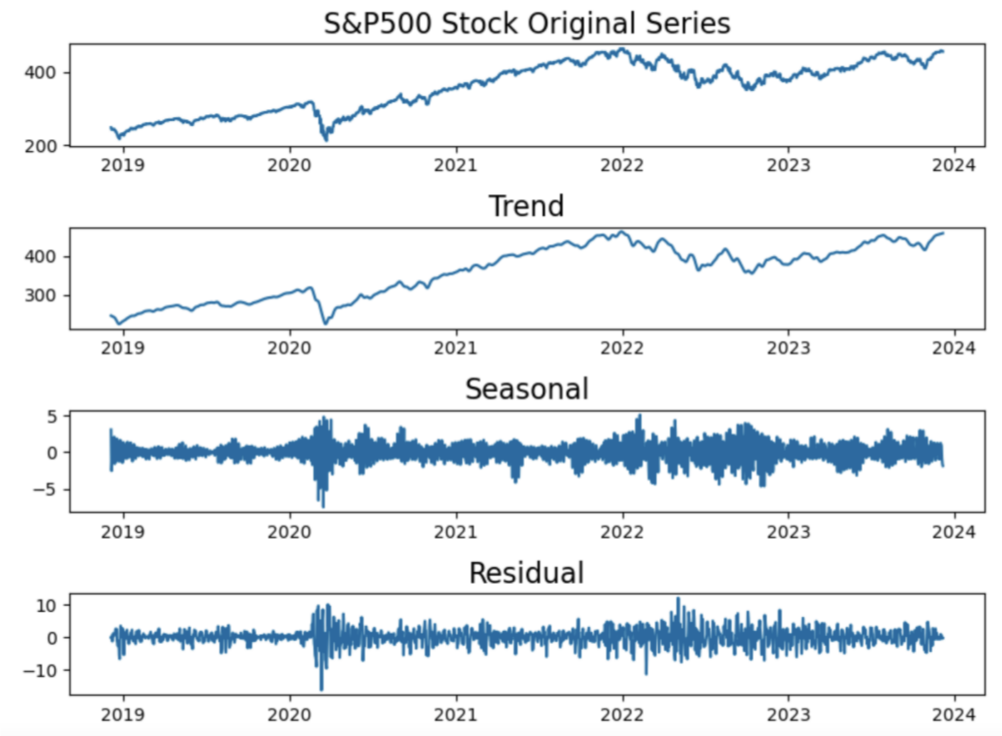
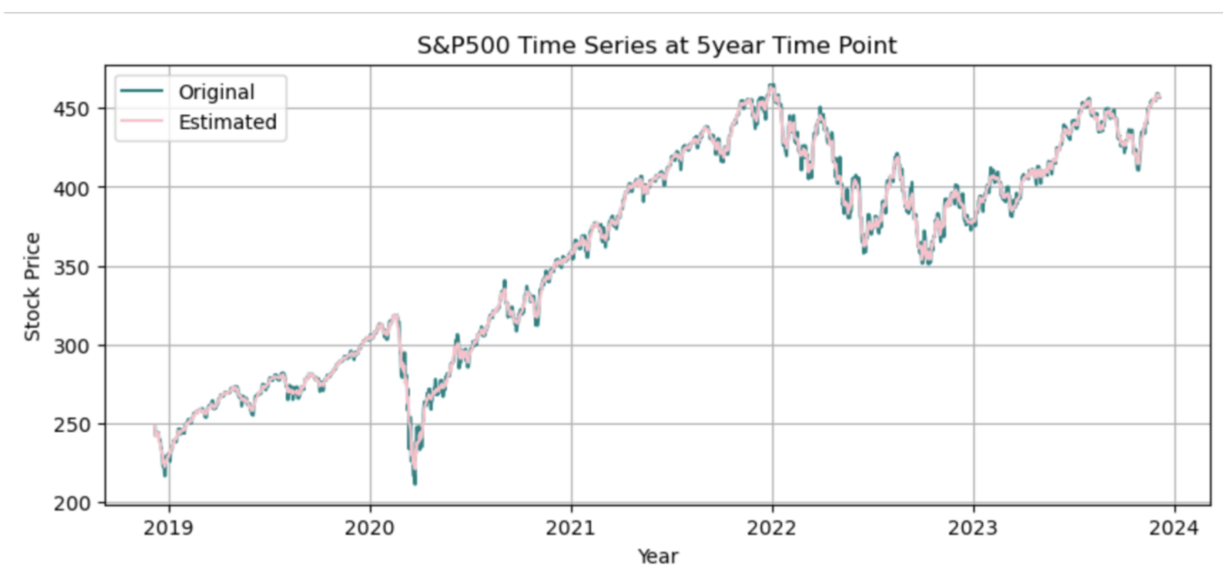


Figure 1b

SPY500 Closing Price (USD) Time Series in 5-year Span.



Note. Orignal and Estimated (STL's Trend & Seasonal) projection.

Figure 1c

SPY500: STL decomposision Residual at Thresholds of ± 3 Std Dev of Normal Distribution.

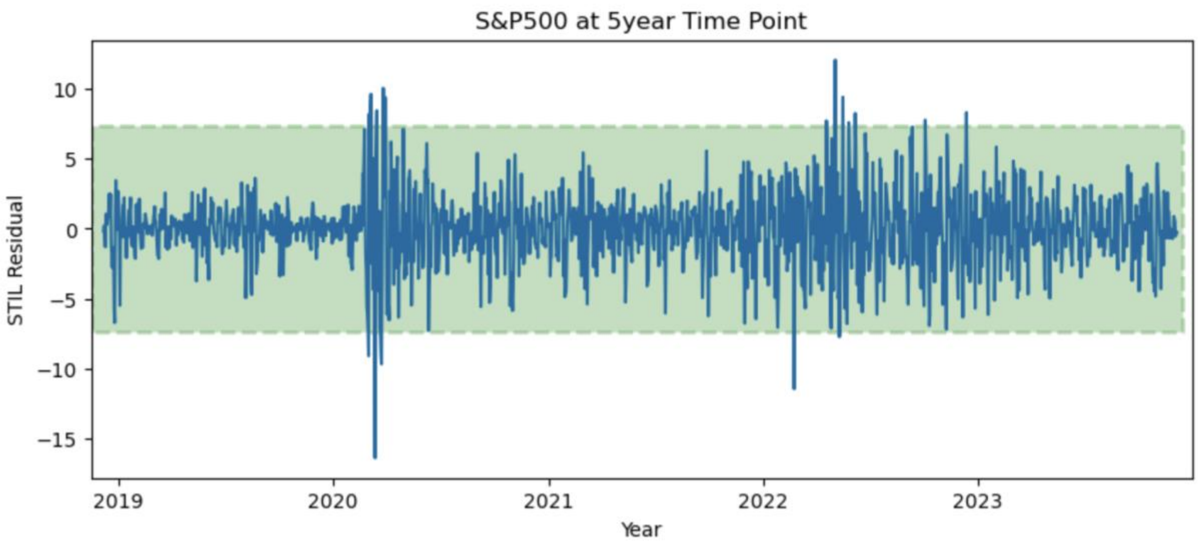


Figure 1d

SPY500 5-yr Time Series with Marked Anomalies Outside of 99.7% Normal Distribution.

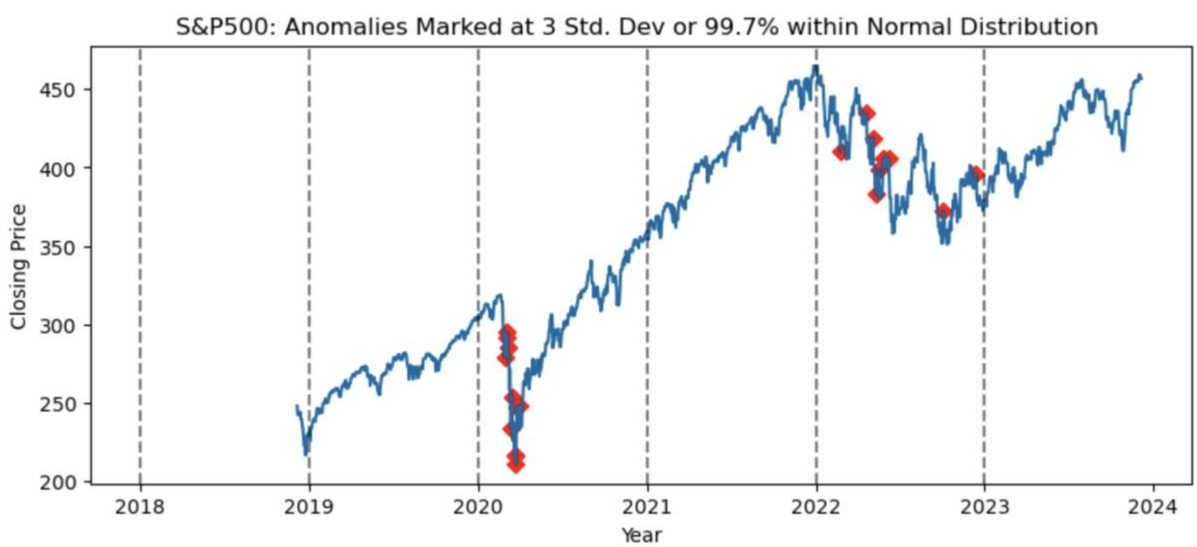


Figure 1e

SPY500 Price Change Vs. Days of the Week Boxplots.

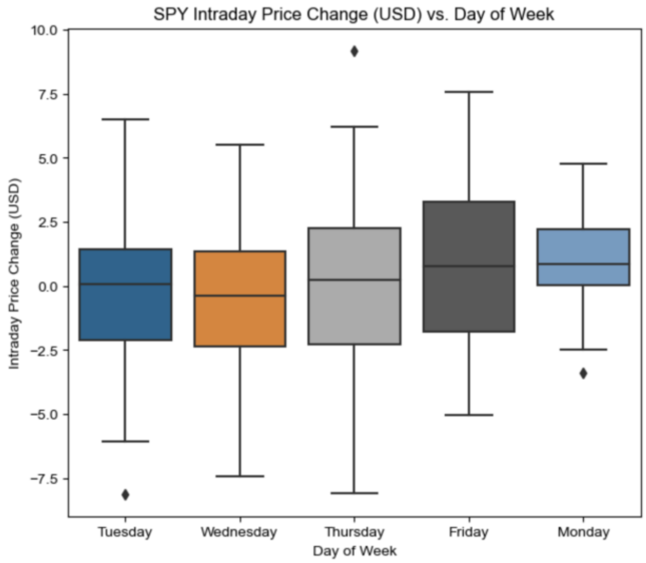


Figure 2a

STL Decomposision of Amazon Time Series Using LOESS Technique

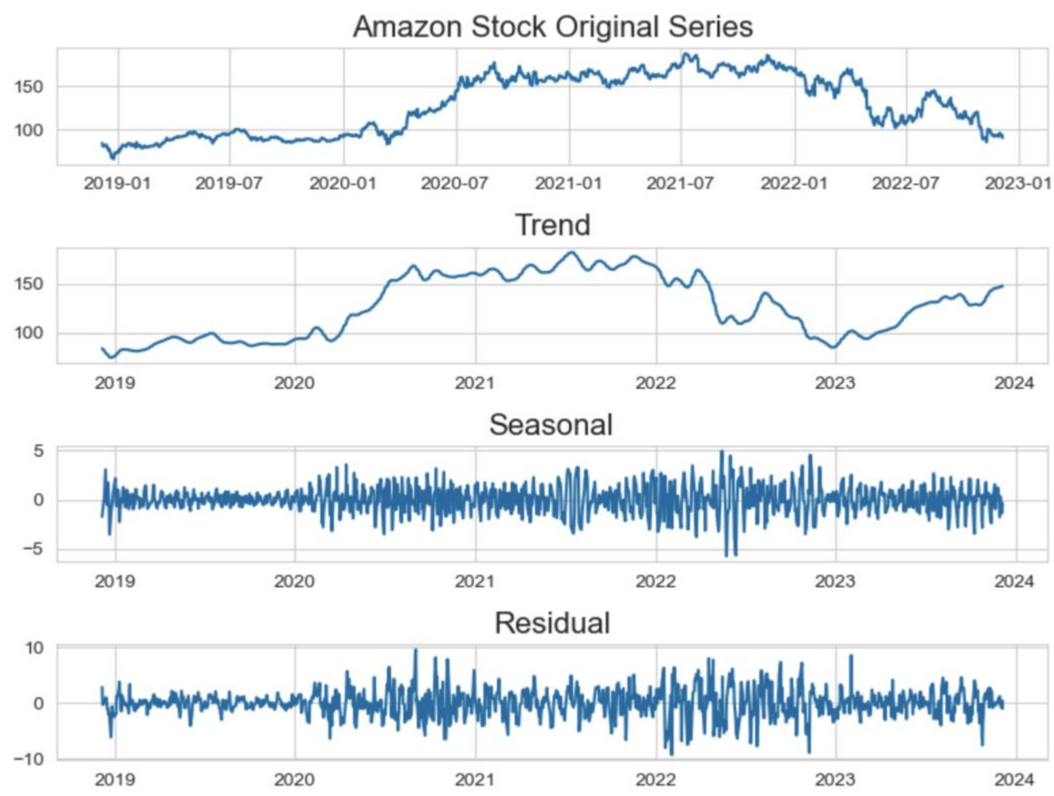


Figure 1b

Amazon Closing Price (USD) Time Series in 5-year Span.

Note. Orignal and Estimated (STL's Trend & Seasonal) projection.

Figure 1c

Amazon: STL decomposision Residual at Thresholds of ± 3 Std Dev of Normal Distribution.

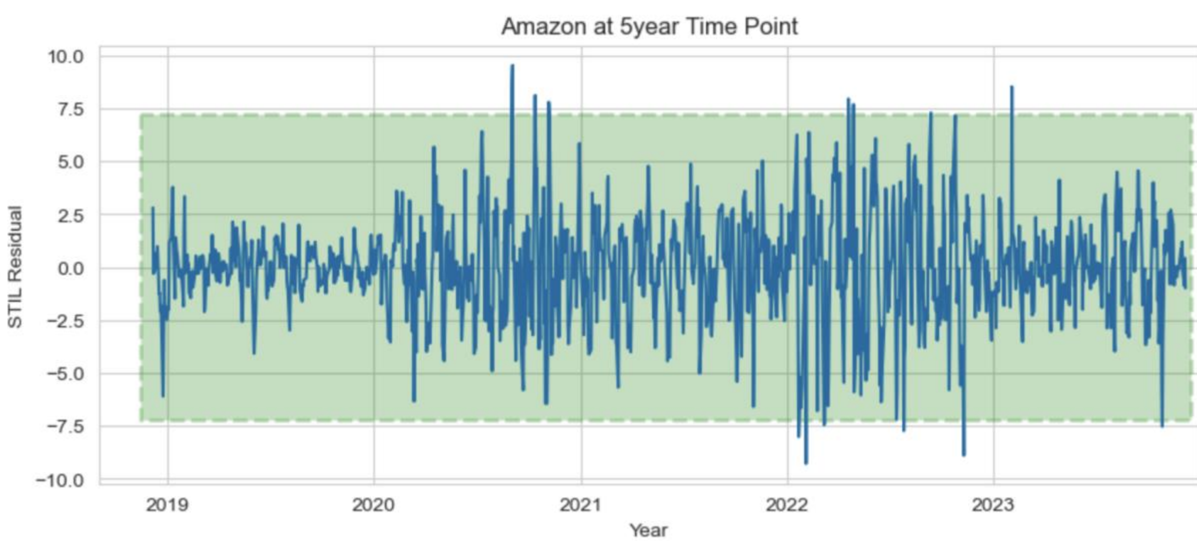


Figure 1d

Amazon 5-yr Time Series with Marked Anomalies Outside of 99.7% Normal Distribution.

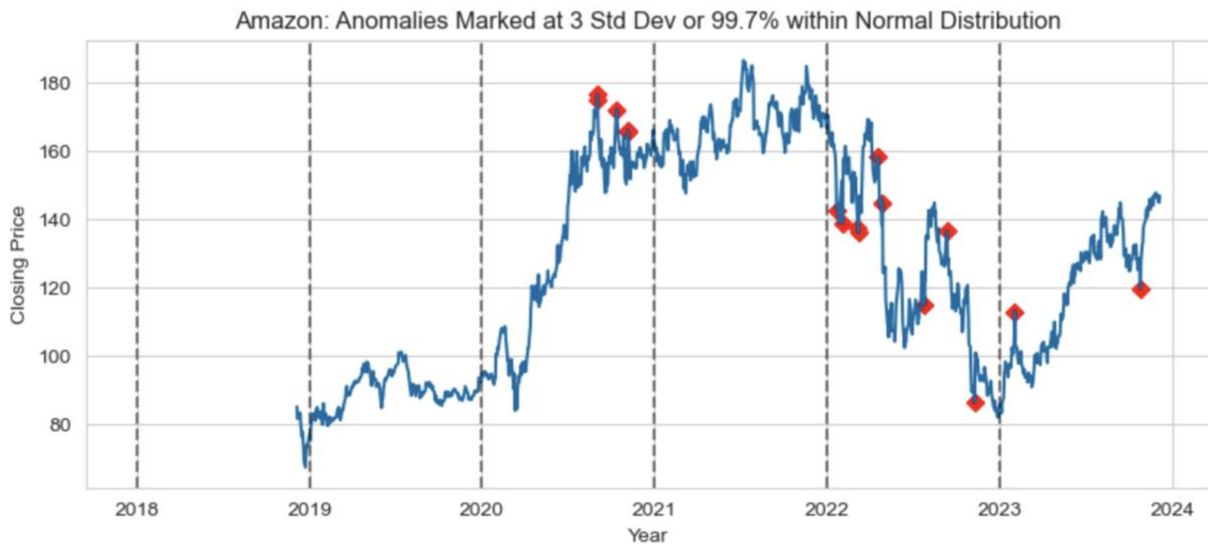


Figure 1e

Amazon Price Change Vs. Days of the Week Boxplots.

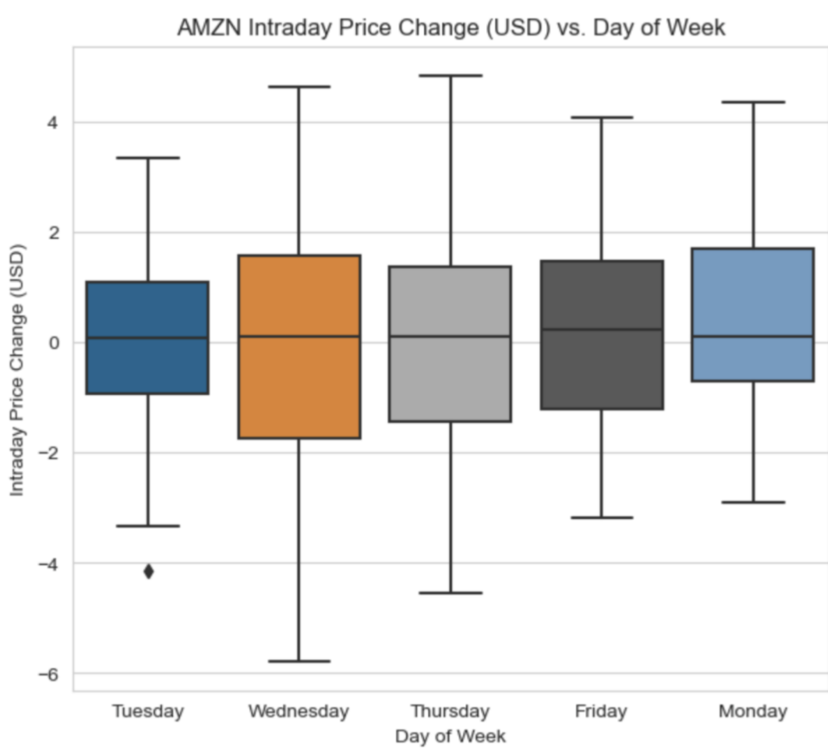


Figure 3a

SPY ACF and PACF Plots.

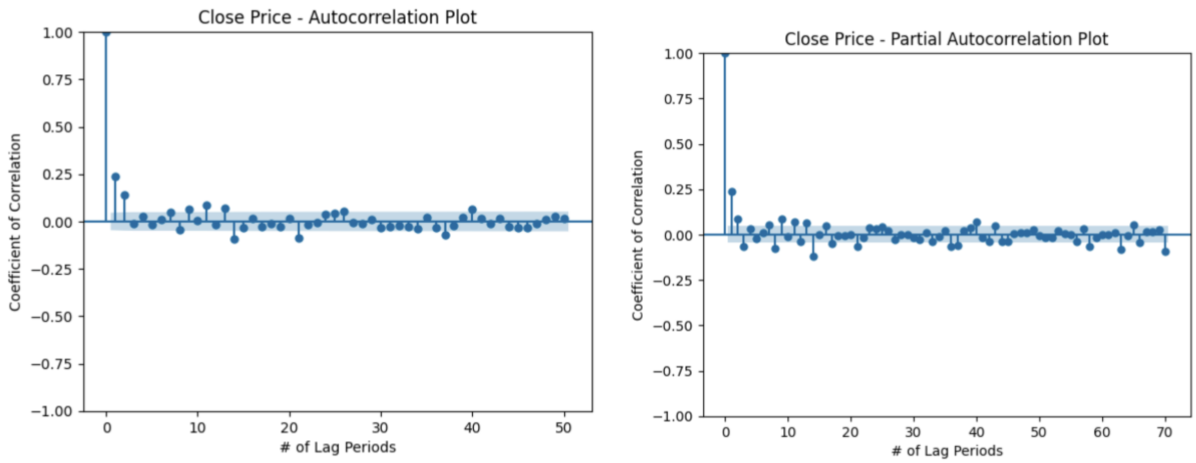


Figure 3b

AMZN ACF and PACF Plots.

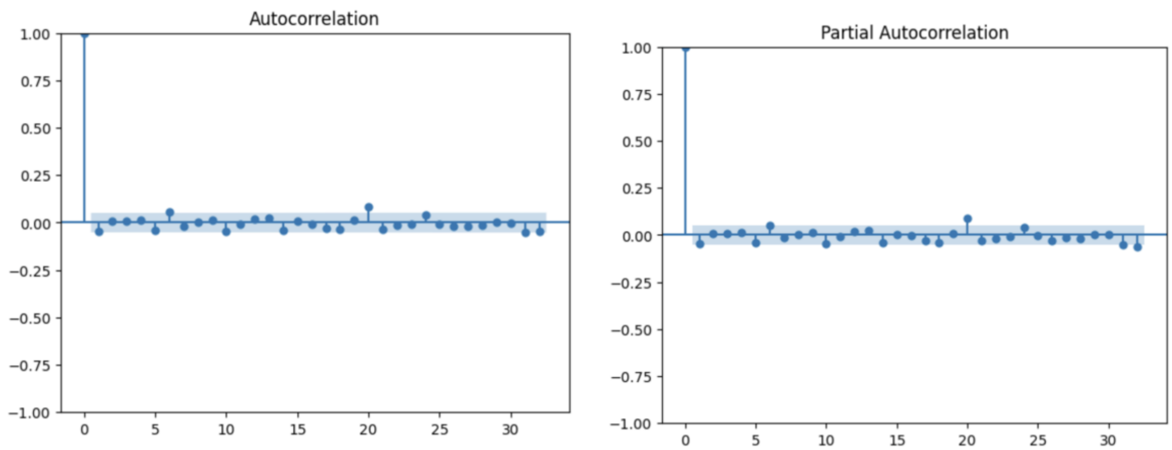


Figure 4a

SPY: SES Forecast Model Plot.

Figure 4b

SPY: ARIMA Forecast Model Plot.

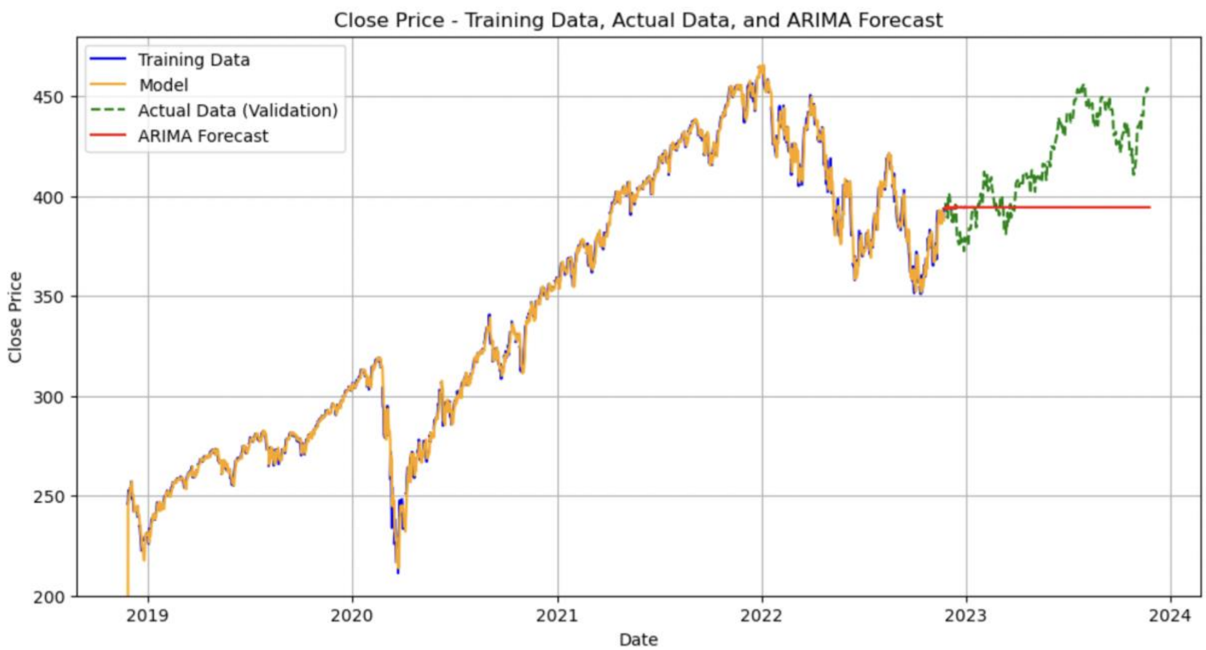


Figure 4c

SPY: AES Forecast Model Plot.

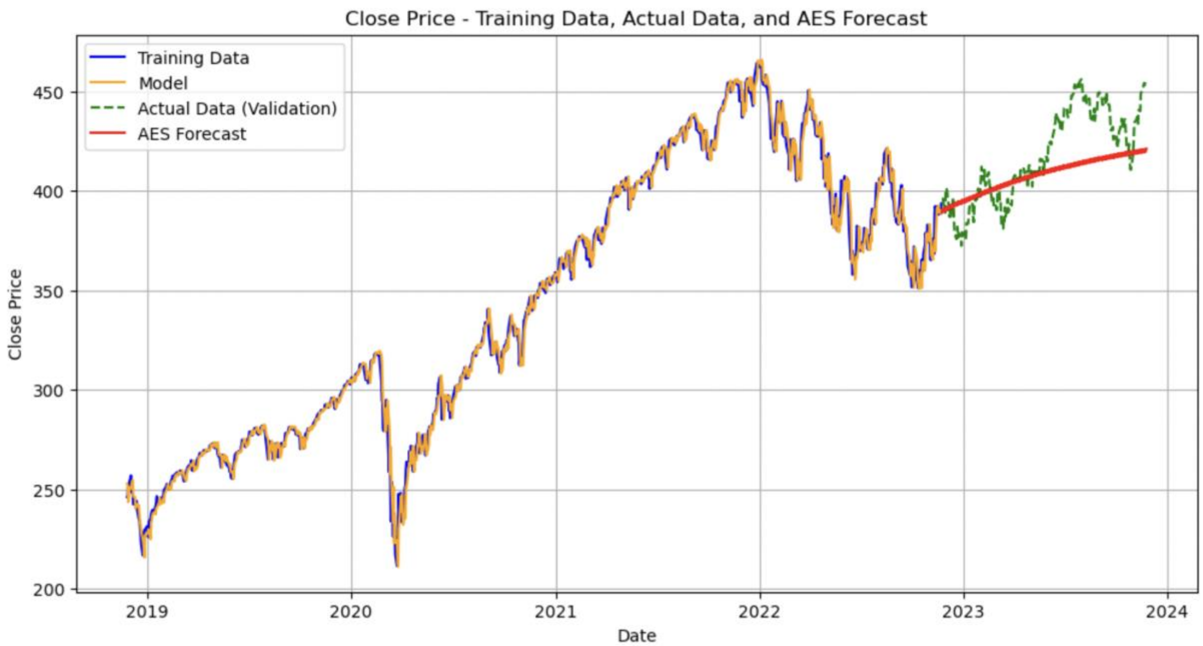


Figure 4d

SPY: Logistic Regression Confusion Matrix.

	precision	recall	f1-score	support
0	0.50	0.71	0.59	14
1	0.83	0.66	0.73	29
accuracy			0.67	43
macro avg	0.66	0.68	0.66	43
weighted avg	0.72	0.67	0.68	43

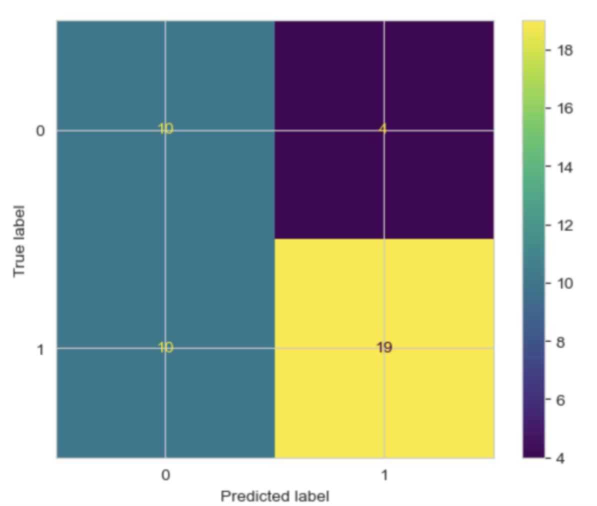


Figure 4e

SPY: Cross-sectional MLP (Neural Network) Model Confusion Matrix.

	precision	recall	f1-score	support
0	0.87	0.93	0.90	14
1	0.96	0.93	0.95	29
accuracy			0.93	43
macro avg	0.92	0.93	0.92	43
weighted avg	0.93	0.93	0.93	43

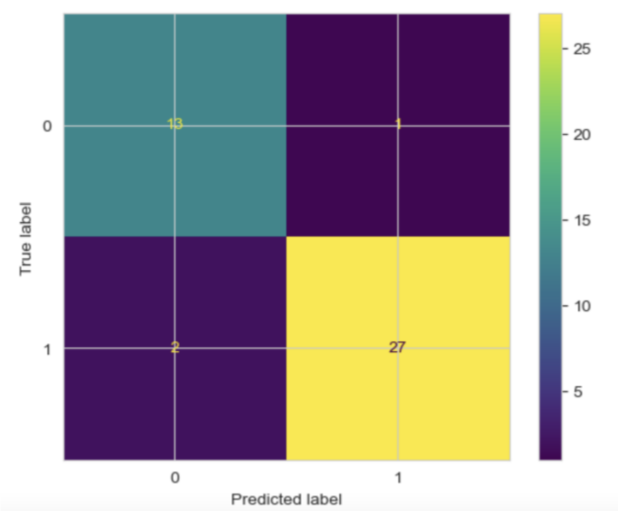


Figure 5a

AMZN: Auto-ARIMA Forecast Model Plot.

Figure 5b

AMZN: ARIMA Forecast Model Plot.

Figure 5c

AMZN: AES Forecast Model Plot.

Figure 5d

AMZN: Logistic Regression Confusion Matrix.

	precision	recall	f1-score	support
0	0.33	0.40	0.36	15
1	0.64	0.57	0.60	28
accuracy			0.51	43
macro avg	0.49	0.49	0.48	43
weighted avg	0.53	0.51	0.52	43

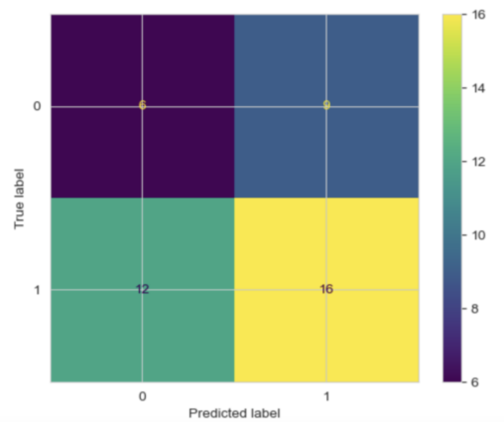
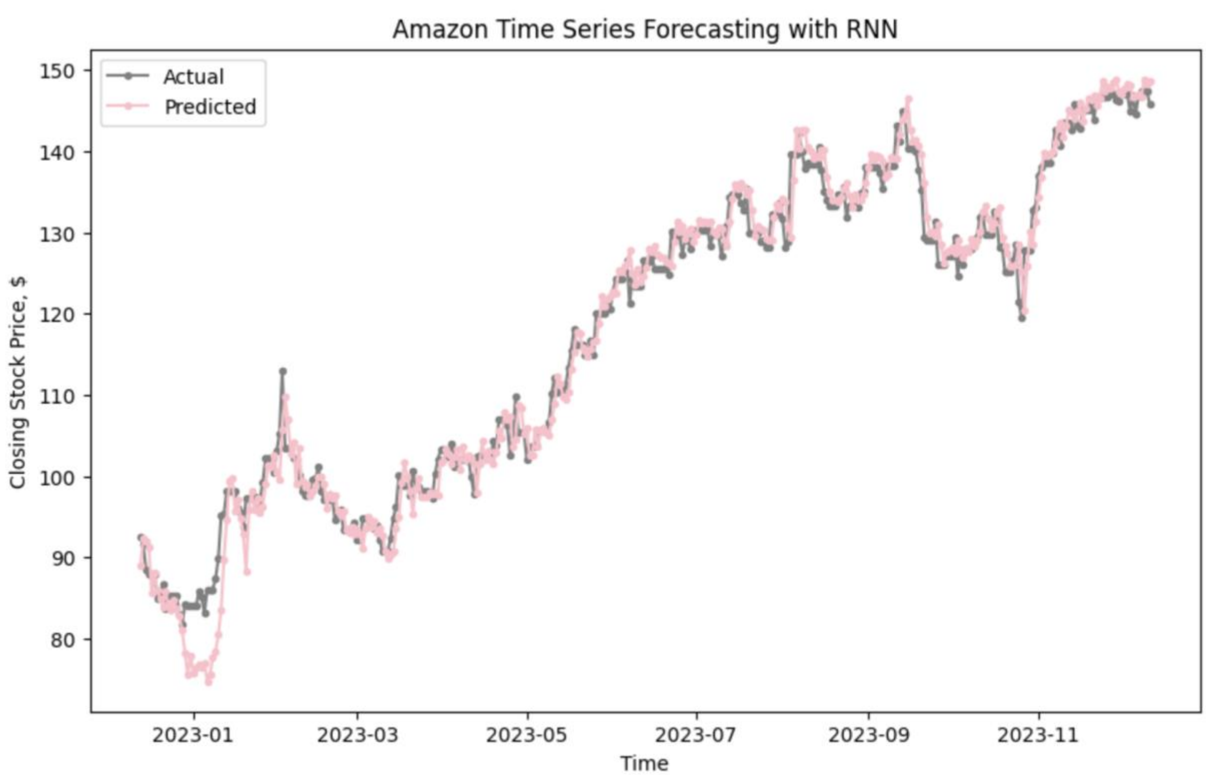


Figure 5e

AMZN: Recurrent Neural Network (RNN) Forecast Model Plot.



References

Ammer, J. (1994, April). Inflation, inflation risk, and stock returns - Federal Reserve Board. **https://www.federalreserve.gov/pubs/ifdp/1994/464/ifdp464.pdf**

Hudomiet, P., Kézdi, G., & Willis, R. J. (2011). Stock market crash and expectations of American households. *Journal of Applied Economics*, *26*(3), 393–415. **https://doi.org/10.1002/jae.1226**

Industrial Business Machines Corporation (2021). *Introduction to CRISP-DM*. Industrial Business Machines Corporation. **https://www.ibm.com/docs/en/spss-modeler/saas?topic=guide-introduction-crisp-dm**

Kiley, M. T. (2023, January 16). Recession Signals and Business Cycle Dynamics: Tying the Pieces Together. *Finance and Economics Discussion Series 2023-008*. **https://doi.org/10.17016/FEDS.2023.008**

Kroencke, T. A. (2022, July 7). Recessions and the stock market. *Journal of Monetary Economics*. **https://www.sciencedirect.com/science/article/pii/S0304393222000976**

OpenAI. (2023). ChatGPT. (Version 3.5). OpenAI. **http://www.openai.com/product/chatgpt**

Appendix

Jupyter Notebook: Team 1 Python Code and Output

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import statsmodels.api as sm
import seaborn as sns
import pandas_datareader.data as web
import yfinance as yf
import datetime
import dateutil.parser

from datetime import date, datetime
from pmdarima.arima import auto_arima
from sklearn import metrics
from sklearn.preprocessing import RobustScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import ParameterGrid, GridSearchCV
from sklearn.neural_network import MLPClassifier, MLPRegressor
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import STL
from sklearn.metrics import (
    mean_absolute_percentage_error,
    confusion_matrix,
    classification_report,
    ConfusionMatrixDisplay,
    accuracy_score
)
from statsmodels.tsa.api import SimpleExpSmoothing, Holt, seasonal_decompose
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tools.eval_measures import rmse, meanabs


import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

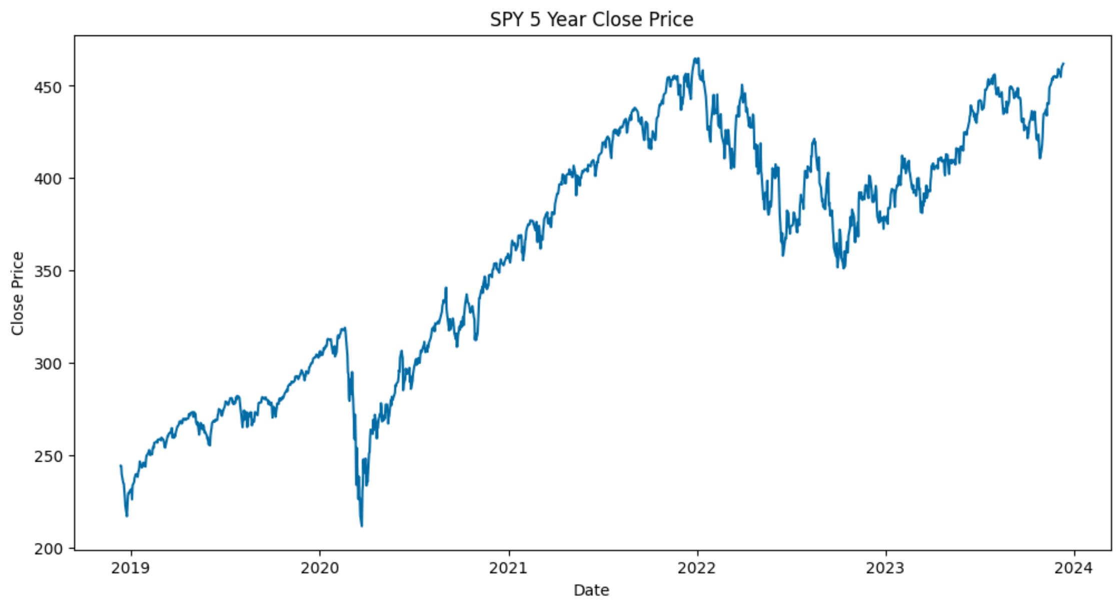
Time Series Eval Metrics Method

```
def ts_eval_metrics(y_true, y_pred):
    print('Time Series Evaluation Metrics')
    print(f'MSE = {metrics.mean_squared_error(y_true, y_pred)}')
    print(f'MAE = {metrics.mean_absolute_error(y_true, y_pred)}')
    print(f'RMSE = {np.sqrt(metrics.mean_squared_error(y_true, y_pred))}')
```

```
    print(f'MAPE = {mean_absolute_percentage_error(y_true, y_pred)}')
    print(f'r2 = {metrics.r2_score(y_true, y_pred)}', end='\n\n')

aapl = yf.Ticker("SPY")
```

Plot an initial time series

```
plt.style.use('tableau-colorblind10') #https://matplotlib.org/stable/gallery/
style_sheets/style_sheets_reference.html
plt.figure(figsize=(12, 6))
close = aapl.history(period='5y')['Close']
plt.plot(close)
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('SPY 5 Year Close Price')
plt.show()
```



Check stationarity

Split Price into halves for statistical analysis

Chaudhari, S. (2021). Stationarity in time series analysis explained using Python. Mathematics and Econometrics. https://blog.quantinsti.com/stationarity

```
X = close.copy()
split = round(len(X)/2)
X1, X2 = X[0:split], X[split:]
mean1, mean2 = X1.mean(), X2.mean()
mean_percent_diff = (mean2 - mean1) / mean1 * 100
var1, var2 = X1.var(), X2.var()
var_percent_diff = (var2 - var1) / var1 * 100
print('mean1=%f, mean2=%3f, mean_percent_diff=%f' % (mean1, mean2, mean_perce
nt_diff))
```

```
print('variance1=%f, variance2=%f, var_percent_diff=%f' % (var1, var2, var_pe
rcent_diff))
```

```
mean1=304.155321, mean2=416.232425, mean_percent_diff=36.848641
variance1=2205.038146, variance2=712.309164, var_percent_diff=-67.696288
```

Augmented Dickey-Fuller Test

Check for stationarity where H0 = time series !stationary; H1 = time series = stationary

If p-value <= .05 or abs(test statistic)>critical value, reject H0

```
result = adfuller(X)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
print(result[4])
```

```
ADF Statistic: -1.529443
p-value: 0.518899
Critical Values:
{'1%': -3.4356006420838963, '5%': -2.8638586845641063, '10%': -2.568004495834
3604}
```

P-value > .05; therefore, time series is not stationary (as expected).

abs(ADF Statistic) < abs(critical value) -> fail to reject H0

Therefore: Time series is not stationary for p-values .01, .05, and .1

Kwiatkowski-Phillips-Schmidt-Shin test

KPSS is opposite of ADF where H0 = time series = stationary; H1 = time series !stationary

If p-value <= .05 || abs(KPSS test statistic)>critical value -> reject H0 -> therefore, !stationary

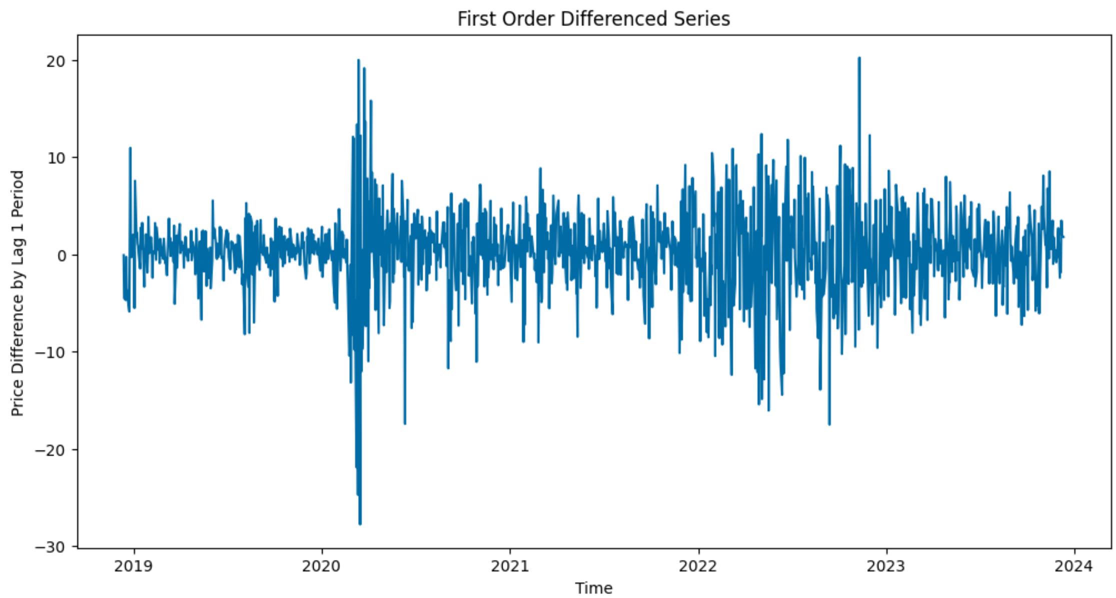
```
result = kpss(X)
print(result)
print('KPSS Test Statistic: %.2f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
print(result[3])
```

```
(4.901644547281778, 0.01, 20, {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%'
: 0.739})
KPSS Test Statistic: 4.90
p-value: 0.010000
Critical Values:
{'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
```

P-value < .05; therefore, not stationary

Transform into stationary series

```
X['lag_1'] = X.diff() # periods=1 by default
X['lag_14'] = X.diff(periods=14)
plt.figure(figsize=(12,6))
plt.plot(X['lag_1'])
plt.title('First Order Differenced Series')
plt.xlabel('Time')
plt.ylabel('Price Difference by Lag 1 Period')
plt.show()
```



X['lag_1'], therefore, is the first-ordered differenced stationary series to use.

```
ts_lag_1 = X.lag_1.dropna()
ts_lag_14 = X.lag_14.dropna()
result = adfuller(ts_lag_1)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
print(result[4])
```

```
ADF Statistic: -10.965788
p-value: 0.000000
Critical Values:
{'1%': -3.4356006420838963, '5%': -2.8638586845641063, '10%': -2.568004495834
3604}
```

(ADF) P-value < .05; therefore, AAPL price series is a difference-stationary series.

Smoothing Methods

Reference

Brownlee, J. (2020, April 12). A gentle introduction to exponential smoothing for time series forecasting in Python. Machine Learning Mastery. https://machinelearningmastery.com/exponential-smoothing-for-time-series-forecasting-in-python/

Triple Exponential Smoothing

Use this because we assume that these time series will have level, trends, seasonality, and noise

Brownlee, J. (2020, August 28). How to grid search triple exponential smoothing for time series forecasting in Python. Machine Learning Mastery. https://machinelearningmastery.com/how-to-grid-search-triple-exponential-smoothing-for-time-series-forecasting-in-python/

```
# Using method from Brownlee, J. (2020).
def exp_smoothing_forecast(history, config):
    t,d,s,p,b,r = config
    # define model model
    history = array(history)
    model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seas
onal_periods=p)
    # fit model
    model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
    # make one step forecast
    yhat = model_fit.predict(len(history), len(history))
    return yhat[0]
```

Time Series Prediction

Data partition

2 years train; last 1 year validation

```
ts_lag_1 = ts_lag_1.asfreq('D')
ts_lag_1 = ts_lag_1.ffill()

past_year = ts_lag_1.iloc[-252:] # Typically 252 trading days per year
before_past_year = ts_lag_1.iloc[:-len(past_year)] # Beginning of selected ti
me series until before 'past_year'

close = aapl.history(period='5y')['Close']
close_train = close.iloc[:-len(past_year)]
close_valid = close.iloc[-252:]
```

Simple Exponential forecaster

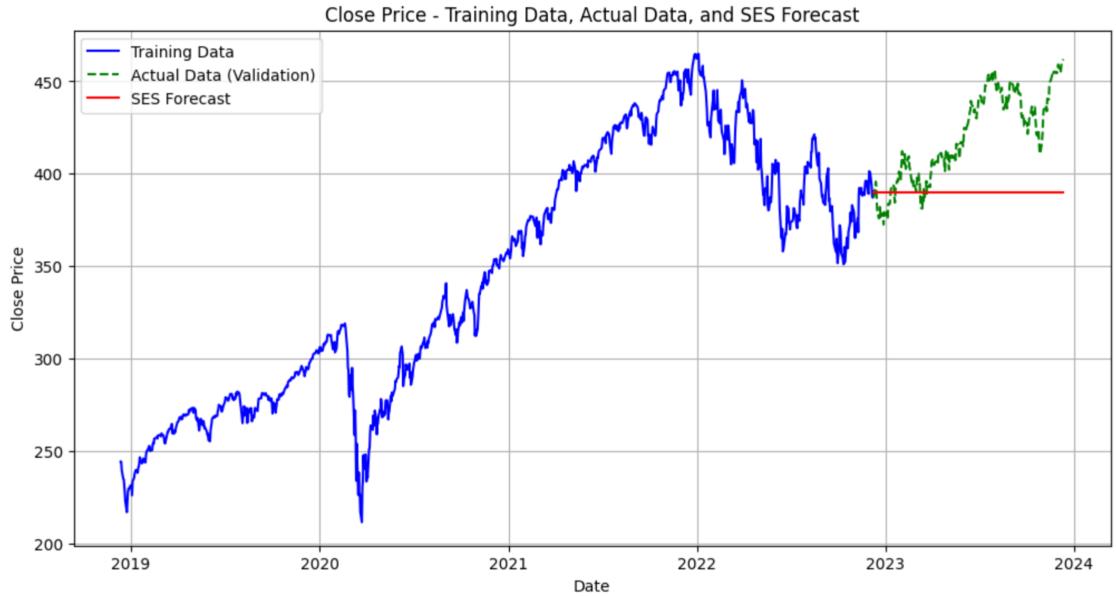
Plot an initial time series

Reference: ADS506 - Module 1 and https://www.statsmodels.org/stable/tsa.html

```
# Forecast 12 months ahead
ses_model = SimpleExpSmoothing(close_train).fit()

ses_pred = ses_model.forecast(steps=len(close_valid))
print('AIC = %s' %(ses_model.aic))
print('BIC = %s' %(ses_model.bic))
ses_eval_metrics = ts_eval_metrics(close_valid, ses_pred)
```

```
AIC = 3052.771470562817
BIC = 3062.598945264136
Time Series Evaluation Metrics
MSE = 1487.6883685940447
MAE = 32.266149118876086
RMSE = 38.570563498528834
MAPE = 0.07425825054942216
r2 = -1.6675002269332975
```

```
plt.figure(figsize=(12, 6))
plt.plot(close_train, label='Training Data', color='blue')
plt.plot(close_valid, label='Actual Data (Validation)', color='green', linest
yle='--')
plt.plot(close_valid.index, ses_pred, label='SES Forecast', color='red')

plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('Close Price - Training Data, Actual Data, and SES Forecast')
plt.legend()
plt.grid(True)
plt.show()
```



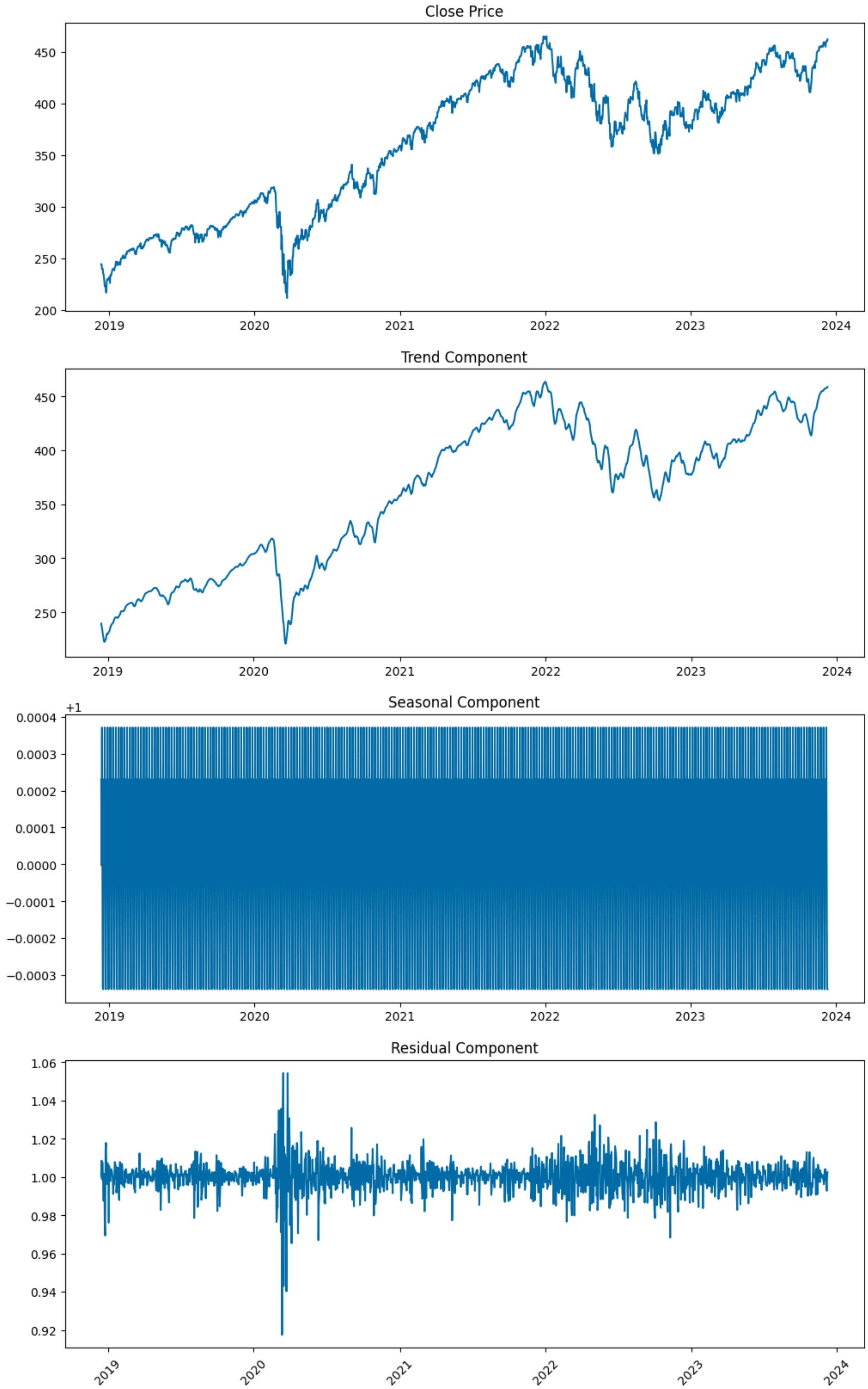
Simple Exponential Smoothing RMSE is 34.56% and it is higher to our success rate criteria. The forecast also doesn't include trend, seasonality and noise

```
# #impute to decompose
close = close.asfreq('D')
close = close.ffill()
```

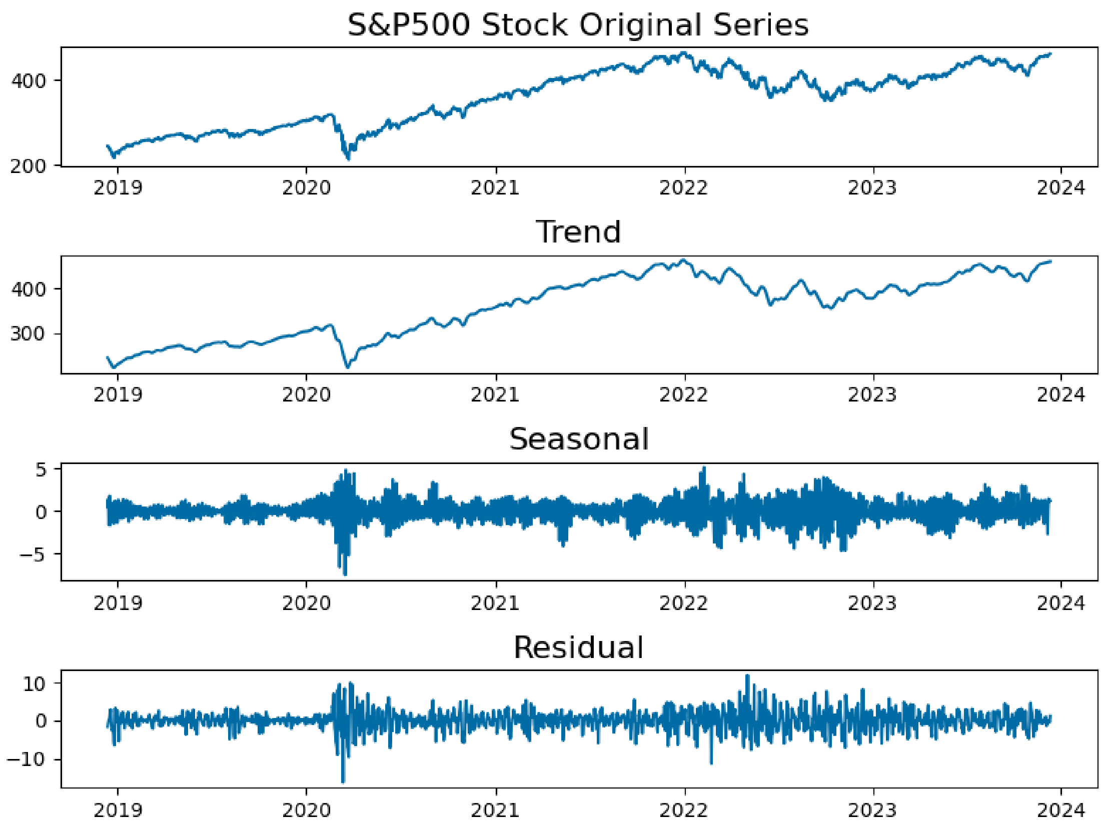
Decomposition of Raw Values

```
decomposition = seasonal_decompose(close, model='multiplicative')
#decomposition.plot()

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

fig, axs = plt.subplots(4)
fig.set_figheight(20)
fig.set_figwidth(12)
plt.xticks(rotation=45)
axs[0].title.set_text('Close Price')
axs[1].title.set_text('Trend Component')
axs[2].title.set_text('Seasonal Component')
axs[3].title.set_text('Residual Component')
axs[0].plot(close)
axs[1].plot(trend)
axs[2].plot(seasonal)
axs[3].plot(residual)
```

```
[<matplotlib.lines.Line2D at 0x283f72c50>]
```


Close Price
450
400
350
300
250
200
2019
2020
2021
2022
2023
2024
Trend Component
450
400
350
300
250
2019
2020
2021
2022
2023
2024
Seasonal Component
+1
0.0004
0.0003
0.0002
0.0001
0.0000
−0.0001
−0.0002
−0.0003
2019
2020
2021
2022
2023
2024
Residual Component
1.06
1.04
1.02
1.00
0.98
0.96
0.94
0.92
2019
2020
2021
2022
2023
2024

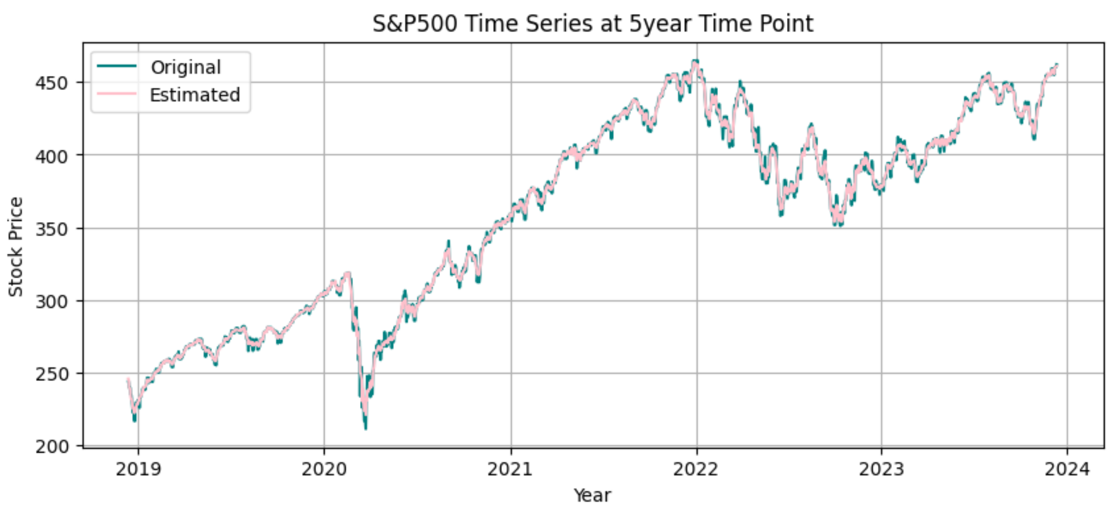
```
stl_close = STL(close)
stl_close_f = stl_close.fit()


# Plot decomposition:

plt.figure(figsize=(8,6))

plt.subplot(4,1,1)
plt.plot(close)
plt.title('S&P500 Stock Original Series', fontsize=16)

plt.subplot(4,1,2)
plt.plot(stl_close_f.trend)
plt.title('Trend', fontsize=16)

plt.subplot(4,1,3)
plt.plot(stl_close_f.seasonal)
plt.title('Seasonal', fontsize=16)

plt.subplot(4,1,4)
plt.plot(stl_close_f.resid)
plt.title('Residual', fontsize=16)

plt.tight_layout()
```



Seasonal-Trend decomposition using LOESS (STL) shows a positive trend on SPY and seasonal around quarterly basis

Anomaly Detection from STL Decomposition

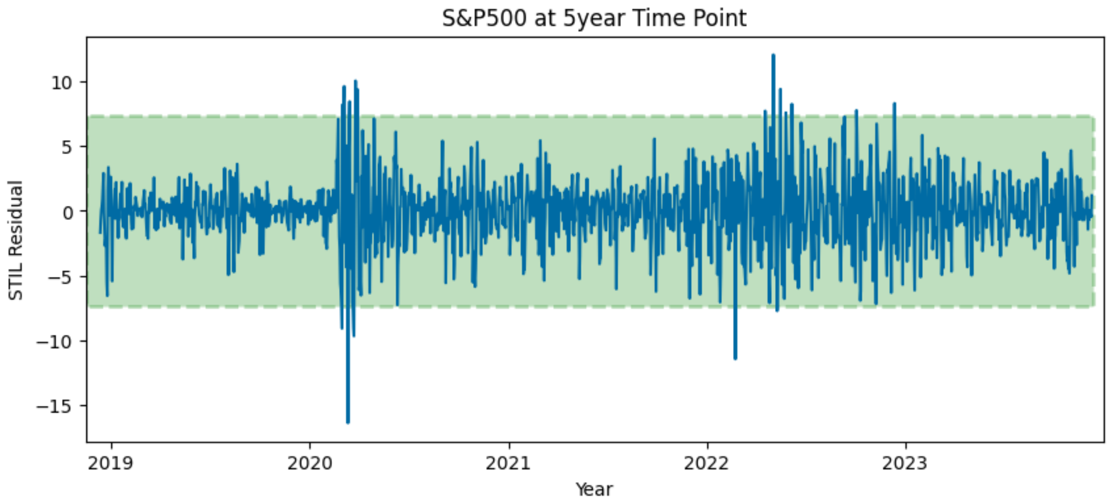
```
estimated0 = stl_close_f.trend + stl_close_f.seasonal
plt.figure(figsize=(10,4))
plt.plot(close, label='Original', color = 'teal')
plt.plot(estimated0, label ='Estimated', color = 'pink')


plt.xlabel('Year')
plt.ylabel('Stock Price')
plt.title('S&P500 Time Series at 5year Time Point')
plt.legend()
plt.grid(True)
plt.show()
```




```
# Taking residuals and detecting anomaly at 3std. dev:

resid_mu0 = stl_close_f.resid.mean()
resid_dev0 = stl_close_f.resid.std()

lower0 = resid_mu0 - 3*resid_dev0
upper0 = resid_mu0 + 3*resid_dev0

# Plot residual threshold:

plt.figure(figsize=(10,4))
plt.plot(stl_close_f.resid)

plt.fill_between([datetime(2018,11,15), datetime(2023,12,15)], lower0, upper0
, color='g', alpha=0.25, linestyle='--', linewidth=2)
plt.xlim(datetime(2018,11,15), datetime(2024,1,1))
```

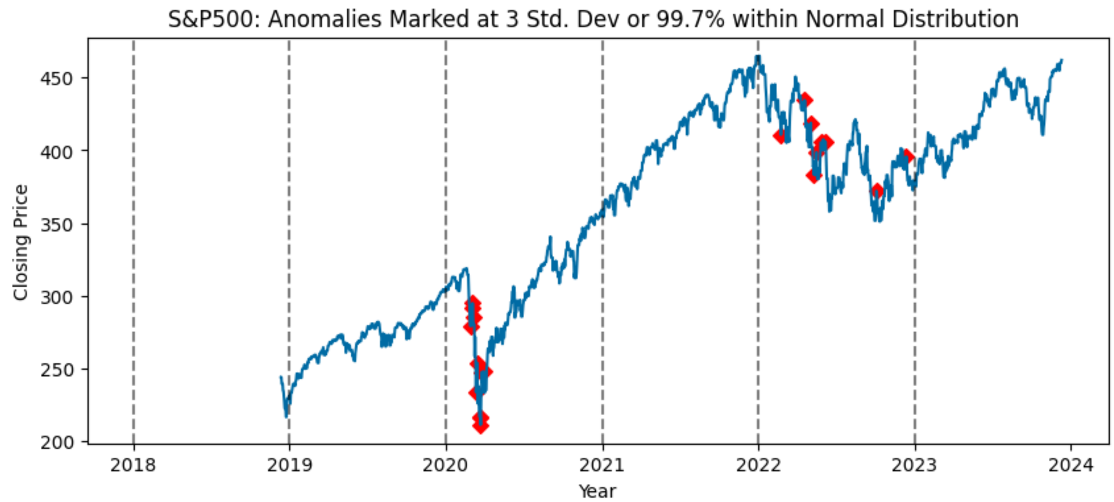
```
plt.xlabel('Year')
plt.ylabel('STIL Residual')
plt.title('S&P500 at 5year Time Point')
```

```
Text(0.5, 1.0, 'S&P500 at 5year Time Point')
```




```
# Identify anomalies by setting the residuals upper and lower limits:

anomalies0 = close[(stl_close_f.resid < lower0) | (stl_close_f.resid > upper0
)]
anomalies0 = pd.DataFrame(anomalies0)

# Plot identified residual anomalies:

plt.figure(figsize=(10,4))
plt.plot(close)

for year in range(2018,2024):
    plt.axvline(datetime(year,1,1), color='k', linestyle='--', alpha=0.5)

plt.scatter(anomalies0.index, anomalies0.Close, color='r', marker='D')
plt.xlabel('Year')
plt.ylabel('Closing Price')
plt.title('S&P500: Anomalies Marked at 3 Std. Dev or 99.7% within Normal Dist
ribution ')
```

```
Text(0.5, 1.0, 'S&P500: Anomalies Marked at 3 Std. Dev or 99.7% within Normal
Distribution ')
```




```
# Anomalies identified outside 3std dev of residuals:

anomalies0.head()
```

```
                                Close
Date
2020-03-01 00:00:00-05:00  279.321136
2020-03-02 00:00:00-05:00  291.417511
2020-03-04 00:00:00-05:00  294.971954
2020-03-05 00:00:00-05:00  285.166504
2020-03-12 00:00:00-04:00  233.924072
```

Gather parameters from decomposition

```
stl_close.config, stl_close.period
```

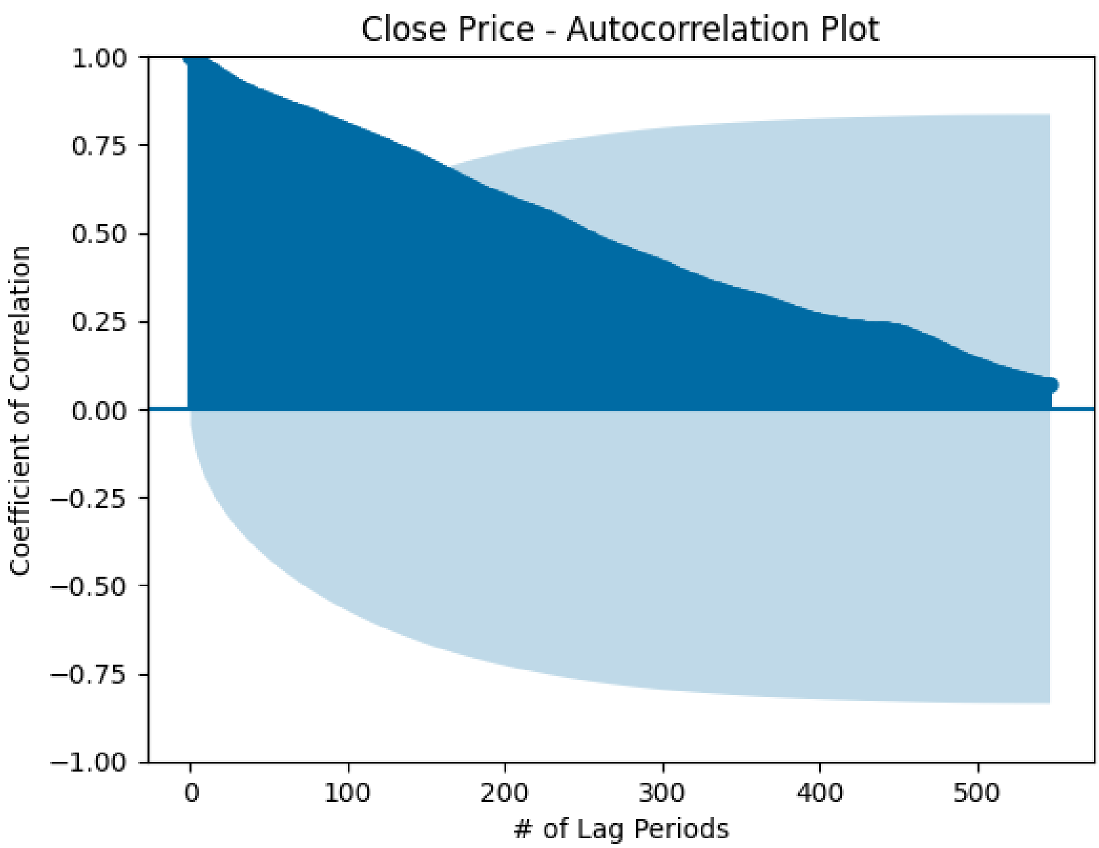
```
({'period': 7,
  'seasonal': 7,
  'seasonal_deg': 1,
  'seasonal_jump': 1,
  'trend': 15,
  'trend_deg': 1,
  'trend_jump': 1,
  'low_pass': 9,
  'low_pass_deg': 1,
  'low_pass_jump': 1,
  'robust': False},
 7)
```

Autocorrelation - Raw Values

Reference: https://www.statsmodels.org/devel/graphics.html#time-series-plots

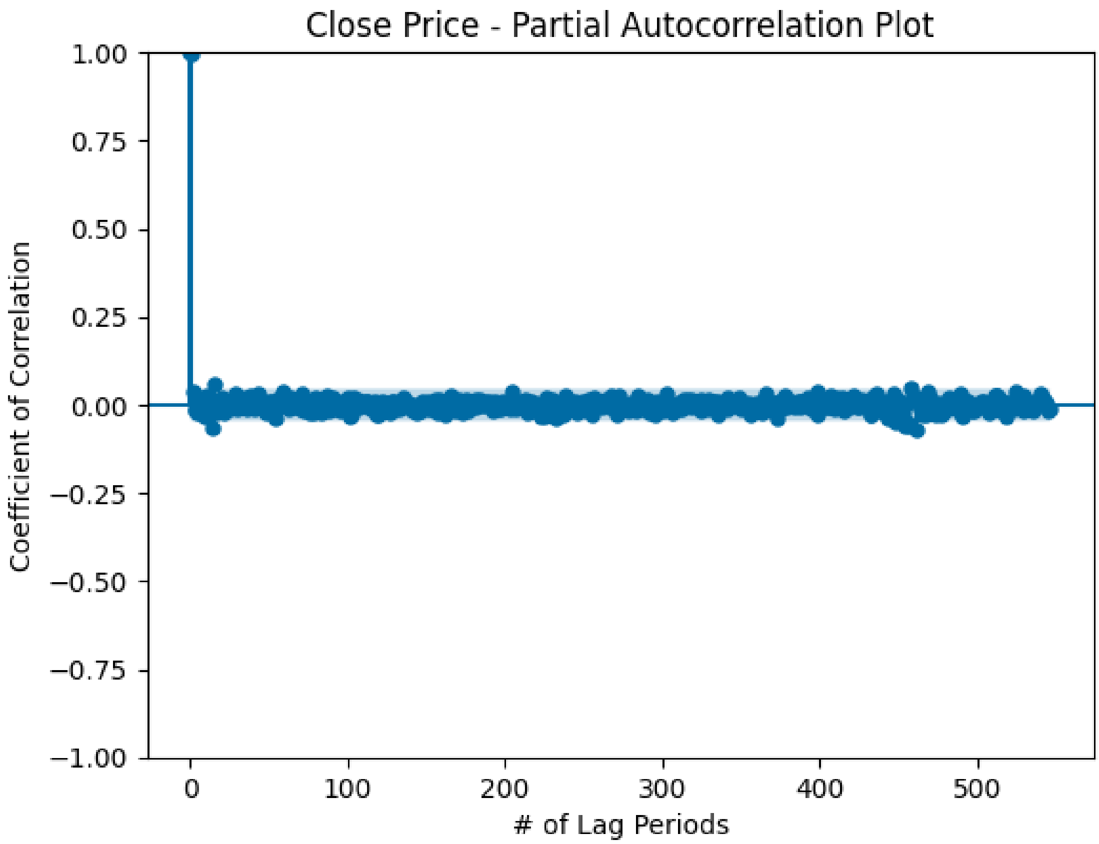
```
plot_acf(close, lags=546)  # Adjust the number of lags as needed
plt.xlabel('# of Lag Periods')
plt.ylabel('Coefficient of Correlation')
```

```
plt.title('Close Price - Autocorrelation Plot')
plt.show()
```



Partial Autocorrelation Plot - Raw Values

```
plot_pacf(close, lags=546)
plt.xlabel('# of Lag Periods')
plt.ylabel('Coefficient of Correlation')
plt.title('Close Price - Partial Autocorrelation Plot')
plt.show()
```

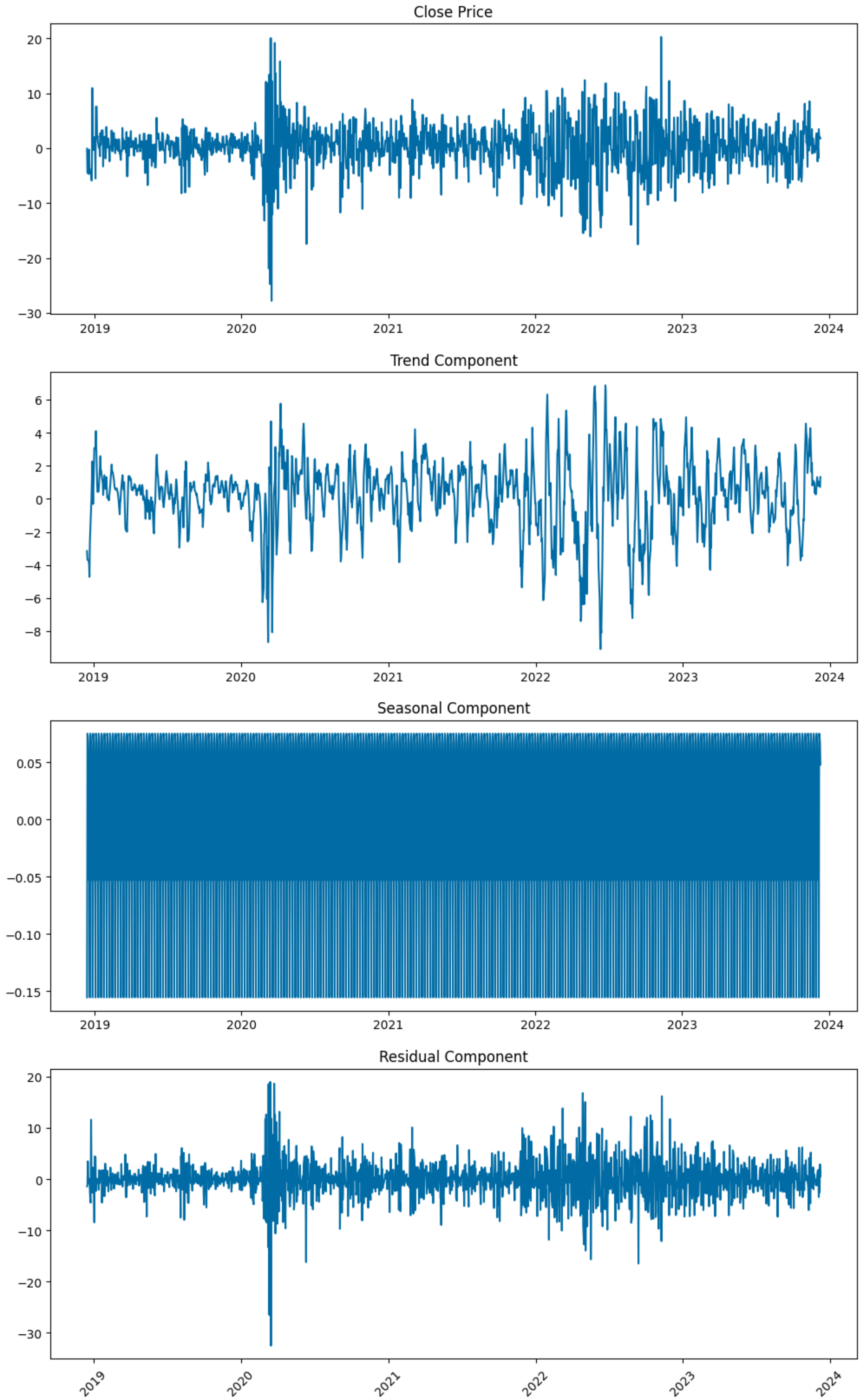


Therefore, based on PACF plot, we may want to do AR model with lags 1, 2 ~415, ~485, ~510.

Lag_1 Decomposition

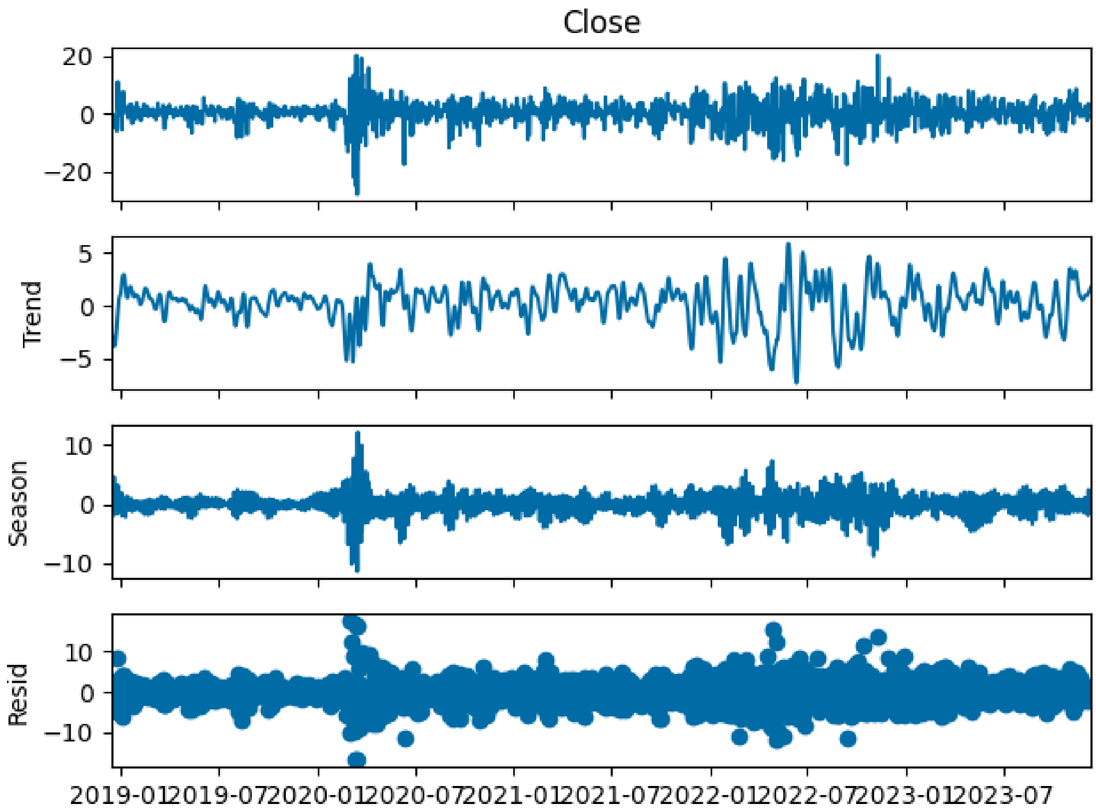
```
decomposition_lag_1 = seasonal_decompose(ts_lag_1, model='additive')
#decomposition.plot()

trend_lag_1 = decomposition_lag_1.trend
seasonal_lag_1 = decomposition_lag_1.seasonal
residual_lag_1 = decomposition_lag_1.resid

fig, axs = plt.subplots(4)
fig.set_figheight(20)
fig.set_figwidth(12)
plt.xticks(rotation=45)
axs[0].title.set_text('Close Price')
axs[1].title.set_text('Trend Component')
axs[2].title.set_text('Seasonal Component')
axs[3].title.set_text('Residual Component')
axs[0].plot(ts_lag_1)
axs[1].plot(trend_lag_1)
axs[2].plot(seasonal_lag_1)
axs[3].plot(residual_lag_1)
```

```
[<matplotlib.lines.Line2D at 0x284b080d0>]
```


Close Price
20
10
0
−10
−20
−30
2019
2020
2021
2022
2023
2024
Trend Component
6
4
2
0
−2
−4
−6
−8
2019
2020
2021
2022
2023
2024
Seasonal Component
0.05
0.00
−0.05
−0.10
−0.15
2019
2020
2021
2022
2023
2024
Residual Component
20
10
0
−10
−20
−30
2019
2020
2021
2022
2023
2024

Decompose using STL

```
stl = STL(ts_lag_1)
stl_plot = stl.fit().plot()
```




```
stl.config, stl.period
```

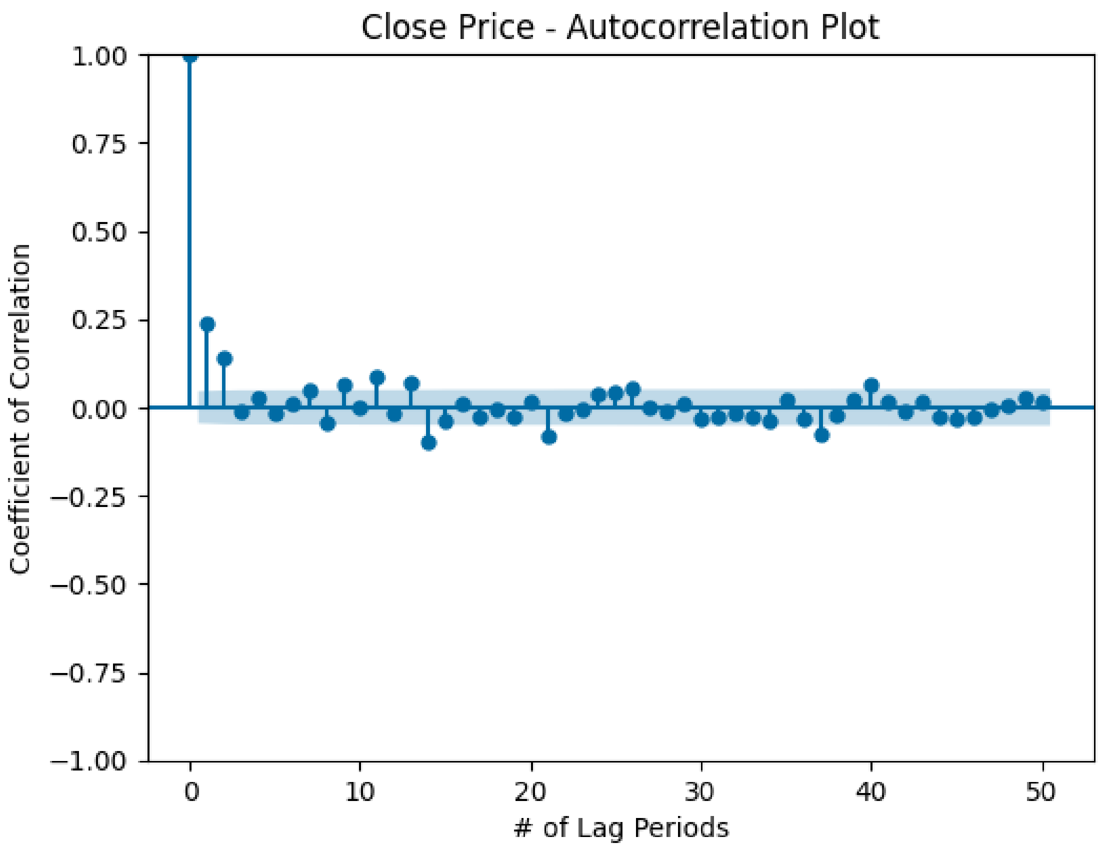
```
({'period': 7,
  'seasonal': 7,
  'seasonal_deg': 1,
  'seasonal_jump': 1,
  'trend': 15,
  'trend_deg': 1,
  'trend_jump': 1,
  'low_pass': 9,
  'low_pass_deg': 1,
  'low_pass_jump': 1,
  'robust': False},
 7)
```

ARIMA Parameter Selection

Reference: Shmueli, G. (2016). ARIMA models [Youtube Video]. https://www.youtube.com/watch?v=0xHf-SJ9Z9U&list=PLoK4oIB1jeK0LHLbZW3DTT05e4srDYxFq&index=29 and

ACF on lag_1 period

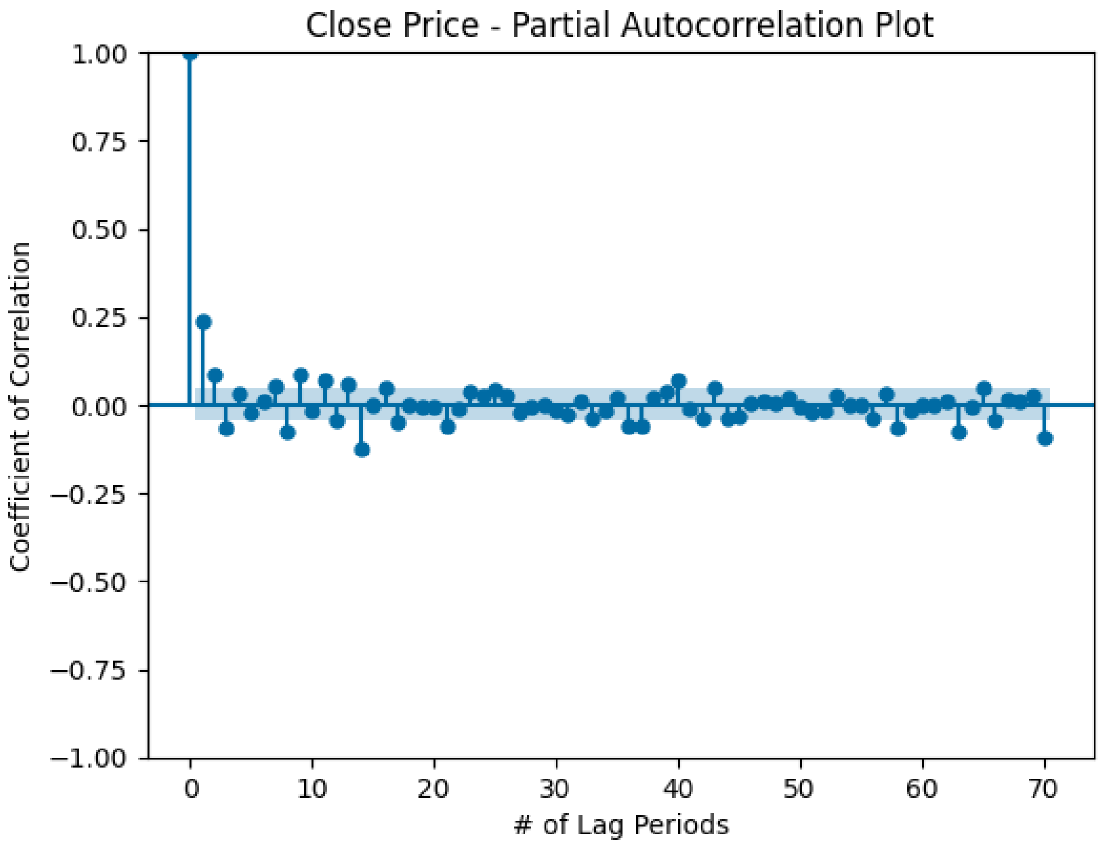
```
plot_acf(ts_lag_1, lags=50)  # Adjust the number of lags as needed
plt.xlabel('# of Lag Periods')
plt.ylabel('Coefficient of Correlation')
plt.title('Close Price - Autocorrelation Plot')
plt.show()
```



Therefore, based on ACF plot, we see a positive pattern in the quarterly basis. We may want to do MA at 0, 1, 2, 11, 13, 14, 21, 37, 40.

Plot PACF on lag_1 period

```
plot_pacf(ts_lag_1, lags=70)
plt.xlabel('# of Lag Periods')
plt.ylabel('Coefficient of Correlation')
plt.title('Close Price - Partial Autocorrelation Plot')
plt.show()
```



Therefore, based on PACF plot, we may want to do AR at 1, 2, 3, 8, 9, 11, 14, and 21, 58, 63, 70.

Iterate through different AR and MA orders to find best AIC and BIC model

Reference: ritvikmath (2020, Oct 7). Time series model selection (AIC & BIC): Time series talk [YouTube]. https://www.youtube.com/watch?v=McEN54l3EPU

Finding AR_orders code would take a small of time

```
ar_orders = [1, 2, 3, 8, 9, 11, 14, 21]#, 58, 63]#, 70] # based on PACF
#ar_orders = [58, 63]#, 70] # based on PACF # attempting higher order from PA
CF
ma_orders = [1, 2, 11, 13, 14, 21] # based on ACF
fitted_model_dict = {}
for i, ar_order in enumerate(ar_orders):
    ar_model = sm.tsa.arima.ARIMA(ts_lag_1, order=(ar_order,1,1),trend='n') #
import statsmodels.api as sm for ARIMA
    ar_model_fit = ar_model.fit()
    fitted_model_dict[ar_order] = ar_model_fit
for ar_order in ar_orders:
    print('AIC for AR(%s): %s' %(ar_order, fitted_model_dict[ar_order].aic))
    print('BIC for AR(%s): %s' %(ar_order, fitted_model_dict[ar_order].bic))
    print('\n')
```

```
AIC for AR(1): 10441.163784804496
BIC for AR(1): 10457.690146316398


AIC for AR(2): 10429.296535840556
BIC for AR(2): 10451.331684523093


AIC for AR(3): 10423.509613447599
BIC for AR(3): 10451.05354930077


AIC for AR(8): 10414.93689094752
BIC for AR(8): 10470.024762653862


AIC for AR(9): 10403.30415380969
BIC for AR(9): 10463.900812686667


AIC for AR(11): 10398.310238029026
BIC for AR(11): 10469.924471247272


AIC for AR(14): 10366.413230061316
BIC for AR(14): 10454.553824791465


AIC for AR(21): 10365.577237817095
BIC for AR(21): 10492.279342741684
```

The lower AIC and BIC is the better model selection; AR(14) has the lowest AIC and BIC

Rerun with AR(14) as default and iterate through different MA orders based on ACF

Reference: ritvikmath (2020, Oct 7). Time series model selection (AIC & BIC): Time series talk [YouTube].

https://www.youtube.com/watch?v=McEN54l3EPU

Finding AR_orders code (q) would take some time

```
#ar_orders = [1, 2, 3, 8, 9, 11, 14, 21]
ar_orders = [0, 1, 2, 11, 13, 14, 21, 37, 40] #actually MA orders, but using same var name for simplicity
fitted_model_dict = {}
for i, ar_order in enumerate(ar_orders):
    ar_model = sm.tsa.arima.ARIMA(ts_lag_1, order=(14,1,ar_order)) #import statsmodels.api as sm for ARIMA
```

```
    ar_model_fit = ar_model.fit()
    fitted_model_dict[ar_order] = ar_model_fit
for ar_order in ar_orders:
    print('AIC for MA(%s): %s' %(ar_order, fitted_model_dict[ar_order].aic))
    print('BIC for MA(%s): %s' %(ar_order, fitted_model_dict[ar_order].bic))
    print('\n')
```

AIC and BIC minimization suggest order=(14,1,1) is the optimal 3-tuple

Measure error statistics on validation set

```
arima_model = sm.tsa.arima.ARIMA(ts_lag_1, order=(14,1,1)).fit() #import stat
smodels.api as sm for ARIMA
print('AIC = %s' %(arima_model.aic))
print('BIC = %s' %(arima_model.bic))
arima_pred = arima_model.forecast(steps=len(close_valid))
arima_metrics = ts_eval_metrics(close_valid, arima_pred)
```

Result: Even with minimum AIC and BIC, ARIMA optimal pdq based on ACF and PACF performs very poorly

Find optimal AES model parameters

```
# Reference: https://www.statsmodels.org/stable/generated/
# statsmodels.tsa.holtwinters.ExponentialSmoothing.html#statsmodels.tsa.holtw
inters.ExponentialSmoothing

aes_param_trend = ['add', 'mul', None]
aes_param_damped_trend = [True, False]
aes_param_seasonal = ['add', 'mul', None]
aes_param_seasonal_periods = [2, 3, 8, 9, 11, 14, 21, 58, 59, 60, 61, 62, 63,
64, 65, 70] # Informed by PACF
aes_param_initial_method = [None, 'estimated', 'heuristic', 'legacy-heuristic
']

fit_param_smoothing_level = [0, .1, .2, .3, .4, .5, .6, .7, .8, .9, 1]
fit_param_smoothing_trend = [0, .1, .2, .3, .4, .5, .6, .7, .8, .9, 1]
fit_param_smoothing_seasonal = [0, .1, .2, .3, .4, .5, .6, .7, .8, .9, 1]
fit_param_damping_trend = [0, .1, .2, .3, .4, .5, .6, .7, .8, .9, 1]
fit_optimzied = [True, False]
fit_method = ['L-BFGS-B', 'TNC', 'SLSQP', 'Powell', 'trust-constr', 'least_sq
uare']

fitted_model_dict = {}
```

Searching for ideal seasonal period parameter

```
for i in aes_param_seasonal_periods:
    aes_model = ExponentialSmoothing(close_train,
                                     trend='mul', # 'add', 'mul', 'additive',
'multiplicative', None
                                     damped_trend=True, #True, False
                                     seasonal= 'mul', # 'mul', 'additive', 'm
```

```
ultiplicative', None
                                         seasonal_periods= i,
                                         initialization_method='heuristic'
                                        ) #'estimated', 'heuristic', 'legacy-heur
istic'
    aes_model = aes_model.fit(smoothing_level=.1,
                              smoothing_trend=.1,
                              #smoothing_seasonal=.1,
                              #damping_trend=.002
                             )
    print('Results for Seasonal Period %s' % (i))
    print('AIC = %s' %(aes_model.aic))
    print('BIC = %s' %(aes_model.bic))
    aes_pred = aes_model.forecast(steps=len(close_valid))
    aes_eval_metrics = ts_eval_metrics(close_valid, aes_pred)
```

Seasonal Periods at 2, 3, and 60 appear to be locally optimal candidate parameter values, but accounting for AIC and BIC, Seasonal Period 2 or 3 may be ideal.

Searching for ideal seasonal parameter

```
close = aapl.history(period='5y')['Close']
close_train = close.iloc[:-len(past_year)]

for i in aes_param_seasonal:
    aes_model = ExponentialSmoothing(close_train,
                                         trend='mul', # 'add', 'mul', 'additive',
'multiplicative', None
                                         damped_trend=False, #True, False
                                         seasonal= i, # 'mul', 'additive', 'multi
plicative', None
                                         seasonal_periods= 3,
                                         initialization_method='heuristic'
                                        ) #'estimated', 'heuristic', 'legacy-heur
istic'
    aes_model = aes_model.fit(smoothing_level=.1,
                              smoothing_trend=.1,
                              #smoothing_seasonal=.1,
                              #damping_trend=.002
                             )
    print('Results for Seasonal %s' % (i))
    print('AIC = %s' %(aes_model.aic))
    print('BIC = %s' %(aes_model.bic))
    aes_pred = aes_model.forecast(steps=len(close_valid))
    aes_eval_metrics = ts_eval_metrics(close_valid, aes_pred)
```

Validation statistics suggest additive seasonality is optimal where it has the lowest RMSE

Searching for ideal trend parameter

```
close = aapl.history(period='5y')['Close']
close_train = close.iloc[:-len(past_year)]
```

```
for i in aes_param_trend:
    aes_model = ExponentialSmoothing(close_train,
                                     trend=i, # 'add', 'mul', 'additive', 'mu
ltiplicative', None
                                     #damped_trend=True, #True, False
                                     seasonal= 'add', # 'mul', 'additive', 'm
ultiplicative', None
                                     seasonal_periods= 3,
                                     initialization_method='heuristic'
                                    ) #'estimated', 'heuristic', 'legacy-heur
istic'
    aes_model = aes_model.fit(smoothing_level=.1,
                              smoothing_trend=.1,
                              #smoothing_seasonal=.1,
                              #damping_trend=.002
                             )
    print('Results for Trend %s' % (i))
    aes_pred = aes_model.forecast(steps=len(close_valid))
    print('AIC = %s' %(aes_model.aic))
    print('BIC = %s' %(aes_model.bic))
    aes_eval_metrics = ts_eval_metrics(close_valid, aes_pred)
```

Validation statistics suggest multiplicative trend is optimal. Upon further investigation, the trend parameter varies on which value is optimal as new data is rolled into the dataframe. Further exploration is needed to determine which trend parameter value is optimal for the greatest likelihood on a rolling 3-day basis.

Searching for ideal aes_param_damped_trend parameter

```
close = aapl.history(period='5y')['Close']
close_train = close.iloc[:-len(past_year)]

for i in aes_param_damped_trend:
    aes_model = ExponentialSmoothing(close_train,
                                     trend='mul', # 'add', 'mul', 'additive',
'multiplicative', None
                                     damped_trend=i, #True, False
                                     seasonal= 'add', # 'mul', 'additive', 'm
ultiplicative', None
                                     seasonal_periods= 3,
                                     initialization_method='heuristic'
                                    ) #'estimated', 'heuristic', 'legacy-heur
istic'
    aes_model = aes_model.fit(smoothing_level=.1,
                              smoothing_trend=.1,
                              #smoothing_seasonal=.1,
                              #damping_trend=.002
                             )
    print('Results for Damped Trend %s' % (i))
    print('AIC = %s' %(aes_model.aic))
```

```
    print('BIC = %s' %(aes_model.bic))
    aes_pred = aes_model.forecast(steps=len(close_valid))
    aes_eval_metrics = ts_eval_metrics(close_valid, aes_pred)
```

Validation statistics suggest trend should be damped.

Searching for optimal initialization method

```
close = aapl.history(period='5y')['Close']
close_train = close.iloc[:-len(past_year)]

for i in aes_param_initial_method:
    aes_model = ExponentialSmoothing(close_train,
                                     trend='add', # 'add', 'mul', 'additive',
'multiplicative', None
                                     damped_trend=False, #True, False
                                     seasonal= 'add', # 'mul', 'additive', 'm
ultiplicative', None
                                     seasonal_periods= 3,
                                     initialization_method=i
                                    ) #'estimated', 'heuristic', 'legacy-heur
istic'
    aes_model = aes_model.fit(smoothing_level=.1,
                              smoothing_trend=.1,
                              #smoothing_seasonal=.1,
                              #damping_trend=.002
                             )
    print('Results for Initialization Method %s' % (i))
    print('AIC = %s' %(aes_model.aic))
    print('BIC = %s' %(aes_model.bic))
    aes_pred = aes_model.forecast(steps=len(close_valid))
    aes_eval_metrics = ts_eval_metrics(close_valid, aes_pred)
```

Validation statistics suggest initialization should be heuristic.

Final pre-fit Advanced Exponential Smoothing Model w/ Parameters

```
aes_model = ExponentialSmoothing(close_train,
                                 trend= 'mul', # 'add', 'mul', 'additive', 'm
ultiplicative', None
                                 damped_trend=True, #True, False
                                 seasonal= 'add', # 'mul', 'additive', 'multi
plicative', None
                                 seasonal_periods= 3,
                                 initialization_method='heuristic') #'estimat
ed', 'heuristic', 'legacy-heuristic'

aes_model = aes_model.fit(smoothing_level=.1,
                          smoothing_trend=.1,
                          #smoothing_seasonal=.1,
                          #damping_trend=.002
                         )
```

```
aes_pred = aes_model.forecast(steps=len(close_valid))
print('AIC = %s' %(aes_model.aic))
print('BIC = %s' %(aes_model.bic))
aes_eval_metrics = ts_eval_metrics(close_valid, aes_pred)
print(aes_eval_metrics)

plt.figure(figsize=(12, 6))
plt.plot(close_train, label='Training Data', color='blue')
plt.plot(aes_model.fittedvalues, label="Model", color = 'orange')
plt.plot(close_valid, label='Actual Data (Validation)', color='green', linest
yle='--')
plt.plot(close_valid.index, aes_pred, label='AES Forecast', color='red')

plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('Close Price - Training Data, Actual Data, and AES Forecast')
plt.legend()
plt.grid(True)
plt.show()
```

Replicate the above, but with seasonal_periods=2 for lower AIC and BIC

```
aes_model = ExponentialSmoothing(close_train,
                                 trend='mul', # 'add', 'mul', 'additive', 'mu
ltiplicative', None
                                 damped_trend=True, #True, False
                                 seasonal= 'add', # 'mul', 'additive', 'multi
plicative', None
                                 seasonal_periods= 2,
                                 initialization_method='heuristic') #'estimat
ed', 'heuristic', 'legacy-heuristic'

aes_model = aes_model.fit(smoothing_level=.1,
                          smoothing_trend=.1,
                          #smoothing_seasonal=.1,
                          #damping_trend=.002
                         )

aes_pred = aes_model.forecast(steps=len(close_valid))
print('AIC = %s' %(aes_model.aic))
print('BIC = %s' %(aes_model.bic))
aes_eval_metrics = ts_eval_metrics(close_valid, aes_pred)
print(aes_eval_metrics)

plt.figure(figsize=(12, 6))
plt.plot(close_train, label='Training Data', color='blue')
plt.plot(aes_model.fittedvalues, label="Model", color = 'orange')
plt.plot(close_valid, label='Actual Data (Validation)', color='green', linest
yle='--')
plt.plot(close_valid.index, aes_pred, label='AES Forecast', color='red')
```

```
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('Close Price - Training Data, Actual Data, and AES Forecast')
plt.legend()
plt.grid(True)
plt.show()
```

Autoregression Integrated Moving Average (ARIMA)

Reference:

Brownlee, J. (2020). How to create an ARIMA model for time series forecasting in Python. Machine Learning Mastery. https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python

https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima.model.ARIMA.html

```
# Auto regression integrated moving average; Find best (p,d,q) by using auto_
arima function
# p = number of lag observations, lag order
# d = number of raw observations differenced, degree of differencing
# q = size of moving average window, order of moving average

close_train = close_train.asfreq('D')
arima_model = sm.tsa.ARIMA(close_train, order=(14,1,1)).fit() #use '2' for qu
adratic trend
print(arima_model.summary())
arima_pred = arima_model.forecast(steps=len(close_valid))

# auto_arima_model.plot_diagnostics(figsize=(12, 8))
# arima_pred.head

# plt.plot(close_valid.index, arima_pred, label="Predicted", color='red')
plt.figure(figsize=(12, 6))
ax = plt.gca()
ax.set_ylim([200, 480])
plt.plot(close_train, label='Training Data', color='blue')
plt.plot(arima_model.fittedvalues, label="Model", color = 'orange') # turn of
f it doesnt work
plt.plot(close_valid, label='Actual Data (Validation)', color='green', linest
yle='--')
plt.plot(close_valid.index, arima_pred, label='ARIMA Forecast', color='red')

plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('Close Price - Training Data, Actual Data, and ARIMA Forecast')
plt.legend()
plt.grid(True)
plt.show()
```

Logistic Regression Model on SPY

Add fields on open-close difference

```
hist = aapl.history(period = '1y')

# Add columns for open-close difference, positive/negative, and high-low diff
erence
hist['open_close'] = hist['Close'] - hist['Open']
hist['positive'] = np.where(hist['open_close'] > 0, 1, 0)
hist['high_low'] = hist['High'] - hist['Low']
hist = hist.drop(['Dividends', 'Stock Splits', 'Capital Gains'], axis=1) # Cl
ean out sparse columns
hist.head()

spy_desc = hist.copy()
spy_desc['Date'] = pd.to_datetime(spy_desc.index)
spy_desc.insert(0, 'day_of_week', spy_desc['Date'].dt.day_name())
spy_desc.head()

# From Deniega (2023) ADS 505 Final Project
target_y = 'open_close'
column_x = 'day_of_week'

plt.figure(figsize=(7, 6))
sns.boxplot(x=column_x, y=target_y, data=spy_desc)
sns.set_style("whitegrid")
plt.title("SPY Intraday Price Change (USD) vs. Day of Week")
plt.xlabel("Day of Week")
plt.ylabel("Intraday Price Change (USD)")
plt.show()

day_week_stats = spy_desc.groupby('day_of_week').describe().transpose()
display(day_week_stats.loc['open_close'])
```

S&P 500 appears to exhibit the least volatility on Mondays (using standard deviation) and the most volatility on Thursdays. This volatility in price should be used to determine typical risk/reward relative to other days of the week.

```
hist_lag = hist.copy()

lag = 3

hist_lag = hist_lag.diff(periods=lag)
#for lag in range(1, 6):
#    globals()[f'hist_lag_{lag}'] = hist_lag.diff(periods=lag)

# Inspired by Deniega, J. (2023) ADS 505 Final Project
# Add lagged columns to same index
for i in range(1, lag+1):
    for col in hist_lag.columns:
        lag_col_name = f'{col}_lag{i}'
```

```
        hist_lag[lag_col_name] = hist_lag[col].shift(i)

hist_lag = hist_lag.dropna()

pd.set_option('display.max_columns', 70)
display(hist_lag.head())
```

Reference: https://scikit-learn.org/

```
## Change n to lag the data
#for n in range(1, 6):

# Data partition
y = hist['positive'] # binary values should not be differenced]['positive'] # binary values should not be differenced
X = hist_lag.drop(['positive'], axis=1)
y = y.reindex(X.index)

end_train_index = 200
X_train = X.iloc[:end_train_index]
X_valid = X.iloc[end_train_index:]

y_train = y.iloc[:end_train_index]
y_valid = y.iloc[end_train_index:]


# Model and fitting
logreg_model = LogisticRegression()
logreg_model.fit(X_train,y_train)

# Model Performance
logreg_pred = logreg_model.predict(X_valid)
logreg_pred = pd.Series(logreg_pred, index=X_valid.index)
y_valid = y_valid.reindex(logreg_pred.index)
cm = confusion_matrix(y_valid, logreg_pred, labels=logreg_model.classes_)
cmd = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=logreg_model
.classes_)
cmd.plot()
print(classification_report(y_valid, logreg_pred))
```

Cross-sectional MLP (Neural Network) Model

Using cross-sectional since the dataframe that will be used already statically assigns the lagged values to its respective column. Shuffling across records does not dynamically change the values of the lagged columns.

Make copy of historical data (differenced at lag=3)

```
hist_diff = hist.copy()
lag = 3
hist_diff = hist_diff.diff(periods=lag)
```

```
# Inspired by Deniega, J. (2023) ADS 505 Final Project
# Add lagged columns to same index
for i in range(1, lag+1):
    for col in hist_diff.columns:
        lag_col_name = f'{col}_lag{i}'
        hist_diff[lag_col_name] = hist_diff[col].shift(i)

hist_diff = hist_diff.dropna() # Remove missing values due to lags out of ran
ge
hist_diff.head()
```

Preprocess dataframes for RobustScaler due to expected outlier stock price movements

```
hist_diff_scale = RobustScaler().fit_transform(hist_diff)
hist_diff_scale = pd.DataFrame(hist_diff_scale, columns=hist_diff.columns, in
dex=hist_diff.index)

# Reset positive column to correct for differencing on all columns
hist_diff_scale['positive'] = hist['positive']
hist_diff_scale.head()
```

Partition

```
y = hist['positive'] # binary values should not be differenced]['positive'] #
binary values should not be differenced
X = hist_diff_scale.drop(['positive'], axis=1)
y = y.reindex(X.index)

# Data partition
end_train_index = 200
X_train = X.iloc[:end_train_index]
X_valid = X.iloc[end_train_index:]

y_train = y.iloc[:end_train_index]
y_valid = y.iloc[end_train_index:]

X_train.shape, X_valid.shape, y_train.shape, y_valid.shape #check lengths of
data partitions
```

Cross-sectional MLP (Neural Network) Model Fitting and Confusion Matrix

Gridsearch for the best set of parameters

To run and find best parameter takes some time

```
# Inspired by Deniega (2023) ADS 505 Final Project
param_grid = {
    'hidden_layer_sizes': [1, 2, 4, 8, 16,
                           '(2,2)', '(3,3)', '(4,4)', '(5,5)', '(6,6)', '(7,7
)', '(8,8)', '(9,9)', '(10,10)',
                           '(2,2,2)', '(3,3,3)', '(4,4,4)', '(5,5,5)', '(6,6,
6)', '(7,7,7)', '(8,8,8)'],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
```

```
    'solver': ['lbfgs', 'sgd', 'adam'],
    'max_iter': [500, 1000, 2000, 4000]
}

grid_search = GridSearchCV(MLPClassifier(random_state=14), param_grid, cv=5,
n_jobs=-1)
grid_search.fit(X_train,y_train)

best = grid_search.best_estimator_
best
```

Fit parameters to MLP model

```
# Model and fitting
mlp_model = MLPClassifier(activation='tanh', hidden_layer_sizes=2, max_iter=2
000, solver='sgd')
mlp_model.fit(X_train,y_train)
```

Evaluate model with Confusion Matrix

```
mlp_pred = mlp_model.predict(X_valid)
mlp_pred = pd.Series(mlp_pred, index=X_valid.index)
y_valid = y_valid.reindex(mlp_pred.index)
cm = confusion_matrix(y_valid, mlp_pred, labels=mlp_model.classes_)
cmd = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=mlp_model.cl
asses_)
cmd.plot()
print(classification_report(y_valid, mlp_pred))
```

TEST: Try different parameters (hidden layers, solvers, etc.)

```
# Model and fitting
mlp_model = MLPClassifier(activation='tanh', hidden_layer_sizes=(3,2), max_it
er=4000, solver='sgd',
                          random_state=14)
mlp_model.fit(X_train,y_train)
mlp_pred = mlp_model.predict(X_valid)
mlp_pred = pd.Series(mlp_pred, index=X_valid.index)
y_valid = y_valid.reindex(mlp_pred.index)
cm = confusion_matrix(y_valid, mlp_pred, labels=mlp_model.classes_)
cmd = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=mlp_model.cl
asses_)
cmd.plot()
print(classification_report(y_valid, mlp_pred))
```

Hidden Layers (3,2) shows .95 accuracy!!!

MLP Regressor

```
mlpr = hist.copy()

y = hist['Close']
X = mlpr.drop(['Close'], axis=1)
y = y.reindex(X.index)
```

```
# Data partition
end_train_index = 200
X_train = X.iloc[:end_train_index]
X_valid = X.iloc[end_train_index:]

y_train = y.iloc[:end_train_index]
y_valid = y.iloc[end_train_index:]

X_train.shape, X_valid.shape, y_train.shape, y_valid.shape #check lengths of
data partitions
```

MLP Regressor Parameter search

```
# Inspired by Deniega (2023) ADS 505 Final
# Finding best estimator will take some time
param_grid = {
    'hidden_layer_sizes': [1, 2, 4, 8, 16,
                           '(2,2)', '(3,2)', '(4,2)', '(5,2)', '(3,3)', '(3,4
)', '(3,5)', '(4,3)', '(4,4)',
                           '(4,5)', '(5,4)', '(5,5)', '(2,2,2)', '(2,2,3)', '
(2,3,2)', '(3,2,2)'],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['lbfgs', 'sgd', 'adam'],
    'alpha': [0.0001, 0.001, 0.01],
    'learning_rate': ['constant', 'invscaling', 'adaptive'],
    'max_iter': [500, 1000, 2000, 4000]
}

grid_search = GridSearchCV(MLPRegressor(random_state=14), param_grid, cv=5, n
_jobs=-1)
grid_search.fit(X_train,y_train)

best = grid_search.best_estimator_
best

# Model and fitting
mlpr_model = MLPRegressor(alpha=0.01, hidden_layer_sizes=8, max_iter=500, ran
dom_state=14,
             solver='lbfgs')
mlpr_model.fit(X_train,y_train)

mlpr_pred = mlpr_model.predict(X_valid)
mlpr_pred = pd.Series(mlpr_pred, index=X_valid.index)
y_valid = y_valid.reindex(mlpr_pred.index)
#close_valid.shape, mlpr_pred.shape
ts_eval_metrics(y_valid, mlpr_pred)
```

High coefficent of determination and low error scores suggests the 8-neuron regressor may be overfitting the validation data. Further analysis required in future to iterate through lower-order neuron models to mitigate overfit risk.

Amazon Closing Stock Price Analysis and Forecasting

```
# Download market data for Amazon:

amzn = yf.Ticker("AMZN")
#amzn.history_metadata

# Import Amazon stock dataset:

amzn = amzn.history(period="5y")
amzn_df = pd.DataFrame(amzn)

display(amzn_df.head(5))
display(amzn_df.tail(5))
display(amzn_df.describe())

# Plot initial Amazon stock time series at 5y time point:

plt.figure(figsize=(10, 5))
plt.plot(amzn_df['Open'], label='Open', color='green', linestyle='--')
plt.plot(amzn_df['High'], label='High', color='blue', linestyle='dotted')
plt.plot(amzn_df['Low'], label='Low', color='blue', linestyle='dashdot')
plt.plot(amzn_df['Close'], label='Close', color='pink')

for year in range(2019,2024):
    plt.axvline(datetime(year,1,1), color='k', linestyle='--', alpha=0.5)


plt.xlabel('Year')
plt.ylabel('Stock Price')
plt.title('Amazon Stock Time Series at 5year Time Point')
plt.legend()
plt.grid(True)
plt.show()
```

Partition train and validation datasets

```
# Partition train and validation datasets:

past_year0 = amzn_df.iloc[-252:] # 252 trading days per year
b_past_year = amzn_df.iloc[:-len(past_year0)]

amzn_dfa = amzn_df['Close'].asfreq('D')
amzn_dfa = amzn_dfa.ffill()

train = b_past_year['Close'].asfreq('D')
amzn_train = train.ffill()
```

```
valid = amzn_df.iloc[-len(past_year0):]
val_close = valid['Close'].asfreq('D')
val_close = val_close.ffill()
```

Test of Stationarity Through Augmented Dickey–Fuller Method

```
# Determine dataset stationarity:
# H0 = time series not stationary; H1 = time series is stationary

result = adfuller(amzn_train )
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
print(result[4])
print('Time series is not stationary')
```

STL Decoposition Using Locally Estimated Scatterplot Smoothing (LOESS)

```
# Fit close stock price dataset to STL:

stl = STL(amzn_df['Close'], period=12)
result = stl.fit()


# Identify seasonal, trend, resid:

seasonal, trend, resid = result.seasonal, result.trend, result.resid

# Plot decomposition:

plt.figure(figsize=(8,6))

plt.subplot(4,1,1)
plt.plot(amzn_df)
plt.title('Amazon Stock Original Series', fontsize=16)

plt.subplot(4,1,2)
plt.plot(trend)
plt.title('Trend', fontsize=16)

plt.subplot(4,1,3)
plt.plot(seasonal)
plt.title('Seasonal', fontsize=16)

plt.subplot(4,1,4)
plt.plot(resid)
plt.title('Residual', fontsize=16)

plt.tight_layout()
```

Holt-Winters Smoothing

```
# Looking at overall trend with Holt's Winter Smoothing

hw_model = ExponentialSmoothing(amzn_train,
                    trend='add', seasonal='add', seasonal_periods=4)
result_hw = hw_model.fit()

amzn_smo_fore = amzn_train.copy()
amzn_smo_fore['Forecast'] = result_hw.fittedvalues
amzn_smo_fore = pd.to_numeric(amzn_smo_fore, errors='coerce')
amzn_smo_fore.dropna(inplace=True)

# Plot Holt's Winter Smoothing:

plt.figure(figsize=(10, 5))
plt.plot(amzn_smo_fore, label='Actual Sales', color = 'Teal', marker='')
plt.plot(result_hw.fittedvalues, label="Holt's Winter Smoothing", color = 'pi
nk')
plt.xlabel('Time')
plt.ylabel('Amazon Close Price')
plt.title('Triple Exponential Smoothing Forecast')
plt.legend()
plt.show()
```

Anomaly Detection Using STL Decomposition

```
# Plot original Amazon Close time series vs Forecasted time series:

estimated = trend + seasonal # from STL
plt.figure(figsize=(10,4))
plt.plot(amzn_df['Close'], label='Original', color = 'teal')
plt.plot(estimated, label ='Estimated', color = 'pink')

plt.xlabel('Year')
plt.ylabel('Stock Price')
plt.title('Amazon Stock Time Series at 5year Time Point')
plt.legend()
plt.grid(True)
plt.show()

# Taking residuals and detecting anomaly at 3std. dev:

resid_mu = resid.mean()
resid_dev = resid.std()

lower = resid_mu - 3*resid_dev
upper = resid_mu + 3*resid_dev

# Plot residual threshold:

plt.figure(figsize=(10,4))
```

```
plt.plot(resid)

plt.fill_between([datetime(2018,11,15), datetime(2023,12,15)], lower, upper,
color='g', alpha=0.25, linestyle='--', linewidth=2)
plt.xlim(datetime(2018,9,1), datetime(2024,1,1))

plt.xlabel('Year')
plt.ylabel('STIL Residual')
plt.title('Amazon at 5year Time Point')

# Identify anomalies by setting the residuals upper and lower limits:

anomalies = amzn_df['Close'][(resid < lower) | (resid > upper)]
anomalies = pd.DataFrame(anomalies)

# Plot identified residual anomalies:    ********In Progress*******

plt.figure(figsize=(10,4))
plt.plot(amzn_df['Close'])

for year in range(2018,2024):
    plt.axvline(datetime(year,1,1), color='k', linestyle='--', alpha=0.5)

plt.scatter(anomalies.index, anomalies.Close, color='r', marker='D')
plt.xlabel('Year')
plt.ylabel('Closing Price')
plt.title('Amazon: Anomalies Marked at 3 Std Dev or 99.7% within Normal Distr
ibution')


# Plot shows anomalies detected outside of +/- 3 Std Dev of normal distributi
on in red.
# Anomaly detection successful in detecting times market is most volitile.

# Anomalies identified outside 3std dev of residuals:

anomalies.head()
```

Transforming Time Series to Stationary

```
# Removing trend by applying the first Difference:

diff_ts = amzn_train.diff().dropna()

# Plot first difference:

plt.figure(figsize=(10,4))
plt.plot(diff_ts)

plt.xlabel('Years', fontsize=10)
plt.ylabel('Amazon Stock Closing Price \n(First Diff.)', fontsize=10)
```

```
# Determine dataset stationarity:
# H0 = time series not stationary; H1 = time series is stationary

result = adfuller(diff_ts)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
print(result[4])
print('Time series is stationary')
```

Selecting a Model

```
# ACF suggest MA Lag 1, 6, 10, 20, 31, 32

plot_acf(diff_ts)
display(plt.show())

# PACF suggest AR Lag 1, 6, 10, 20, 31, 32

plot_pacf(diff_ts, method='ywm')
display(plt.show())
```

Model Selection Criteria:

BIC = $\ln(n)k$ - 2l

AIC = 2k - 2l

(l) = a log likelihood

(k) = a number of parameters

(n) = a number of samples used for fitting

Auto-ARIMA Model

```
# Auto ARIMA Model:

auto_arima_model = auto_arima(amzn_train, d=1, seasonal=True, stepwise=True,
trace=True)
auto_arima_model.summary()

arima_pred0 = auto_arima_model.predict(n_periods=len(val_close))

# ARIMA Model and Forecast at ARIMA(2,1,2):

# p = number of lag observations, lag order
# d = number of raw observations differenced, degree of differencing
# q = size of moving average window, order of moving average

arima_m = sm.tsa.ARIMA(amzn_train, order=(2,1,2)).fit()
print(arima_m.summary())

arima_pred1 = arima_m.forecast(steps=len(val_close))
```

```
# Plot Auto ARIMA Result:

plt.figure(figsize=(12, 6))
plt.plot(amzn_train, label='Training Data', color='blue')
plt.plot(arima_m.fittedvalues, label="Model", color = 'orange')
plt.plot(val_close, label='Actual Data (Validation)', color='green', linestyl
e='--')
plt.plot(val_close.index, arima_pred1, label='Auto ARIMA Forecast', color='re
d')

plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('Close Price - Training Data, Actual Data, and Auto ARIMA Forecast'
)
plt.legend()
plt.grid(True)
plt.show()

# Plot shows best performing ARIMA parameter at (2, 1, 2) values of p,d,q.
# Although, the auto forecast did very poorly in predicting the validation da
taset.
```

ARIMA Model

```
# AR lag optimization:

ar_orders0 = [1, 6, 10, 20, 31, 32]
fitted_model_dict = {}

for i, ar_order0 in enumerate(ar_orders0):
    ar_model0 = sm.tsa.arima.ARIMA(amzn_train, order=(ar_order0,1,1),trend='n
')
    ar_model_fit0 = ar_model0.fit()
    fitted_model_dict[ar_order0] = ar_model_fit0

for ar_order0 in ar_orders0:
    print('AIC for AR(%s): %s' %(ar_order0, fitted_model_dict[ar_order0].aic)
)
    print('BIC for AR(%s): %s' %(ar_order0, fitted_model_dict[ar_order0].bic)
)
    print('\n')
```

Result:

AR order 1 has the lowest AIC and BIC scores.

```
# MA lag optimization:

ma_orders0 = [1, 6, 10, 20, 31, 32]
fitted_model_dict = {}
```

```
for i, ma_order0 in enumerate(ma_orders0):
    ma_model0 = sm.tsa.arima.ARIMA(amzn_train, order=(1,1,ma_order0),trend='n
')
    ma_model_fit0 = ma_model0.fit()
    fitted_model_dict[ma_order0] = ma_model_fit0

for ma_order0 in ma_orders0:
    print('AIC for AR(%s): %s' %(ma_order0, fitted_model_dict[ma_order0].aic)
)
    print('BIC for AR(%s): %s' %(ma_order0, fitted_model_dict[ma_order0].bic)
)
    print('\n')
```

Result:

MA order 1 has the lowest AIC and BIC scores.

```
# ARIMA Model and Forecast at ARIMA(1,1,1):

# p = number of lag observations, lag order
# d = number of raw observations differenced, degree of differencing
# q = size of moving average window, order of moving average

arima_m0 = sm.tsa.ARIMA(amzn_train, order=(1,1,1)).fit()
print(arima_m0.summary())

arima_pred2 = arima_m0.forecast(steps=len(val_close))

# Statistical Metrics:

print('AIC = %s' %(arima_m0.aic))
print('BIC = %s' %(arima_m0.bic))
arima0_metrics = ts_eval_metrics(val_close, arima_pred2)

# Plot ARIMA Result:

plt.figure(figsize=(12, 6))
plt.plot(amzn_train, label='Training Data', color='blue')
plt.plot(arima_m0.fittedvalues, label="Model", color = 'orange')
plt.plot(val_close, label='Actual Data (Validation)', color='green', linestyl
e='--')
plt.plot(val_close.index, arima_pred2, label='ARIMA Forecast', color='red')

plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('Close Price - Training Data, Actual Data, and ARIMA Forecast')
plt.legend()
plt.grid(True)
plt.show()

# Same with auto-ARIMA, ARIMA with (1,1,1) parameters selected through metric
```

```
scores performance did very poorly.
# Manual selection of ARIMA parameters did not predict closing prices well co
mpared to the validation dataset.
```

AES Model

```
# Define AES parameters for optimization:

aes_param_trend0 = ['add', 'mul', None]
aes_param_seasonal0 = ['add', 'mul', None] # set to mul by default
aes_param_initial_method0 = [None, 'estimated', 'heuristic', 'legacy-heuristi
c']


fitted_model_dict0 = {}

# Trend parameter optimization:

for i in aes_param_trend0:
    aes_model2 = ExponentialSmoothing(amzn_train,
                                     trend=i,
                                     damped_trend=False, # Error message: Can
only dampen the trend component
                                     seasonal= 'mul',
                                     seasonal_periods= 252,
                                     initialization_method='heuristic'
                                    )
    aes_model2 = aes_model2.fit(
                               )

    print('Results for Trend %s' % (i))
    print('AIC = %s' %(aes_model2.aic))
    print('BIC = %s' %(aes_model2.bic))
    aes_pred2 = aes_model2.forecast(steps=len(val_close))
    aes_eval_metrics2 = ts_eval_metrics(val_close, aes_pred2)
```

Result

No trend parameter has the lowest AIC and BIC scores.

```
# Initialization method optimization:

for i in aes_param_initial_method0:
    aes_model3 = ExponentialSmoothing(amzn_train,
                                     trend= None,
                                     seasonal= 'mul',
                                     seasonal_periods= 252,
                                     initialization_method=i
                                    )
    aes_model3 = aes_model3.fit(
                               )
```

```
    print('Results for Initialization Method %s' % (i))
    print('AIC = %s' %(aes_model3.aic))
    print('BIC = %s' %(aes_model3.bic))
    aes_pred3 = aes_model3.forecast(steps=len(val_close))
    aes_eval_metrics3 = ts_eval_metrics(val_close, aes_pred3)
```

Result

For initialization method, heuristic and estimated methods performed the best.

```
# Final AES Model on train dataset:

aes_modelf = ExponentialSmoothing(amzn_train,
                                  trend= None,
                                  seasonal= 'mul', # set by default, add is po
or performer visually
                                  seasonal_periods= 252, #252 trading days per
year forecast
                                  initialization_method='heuristic')

aes_modelf = aes_modelf.fit(smoothing_level=.5,
                            smoothing_trend=.5)


aes_predf = aes_modelf.forecast(steps=len(val_close))
aes_eval_metricsf = ts_eval_metrics(val_close, aes_predf)

print('AIC = %s' %(aes_modelf.aic))
print('BIC = %s' %(aes_modelf.bic))

# Plot AES Result:

plt.figure(figsize=(12, 6))
plt.plot(amzn_train, label='Training Data', color='blue')
plt.plot(aes_modelf.fittedvalues, label="Model", color = 'orange')
plt.plot(val_close, label='Actual Data (Validation)', color='green', linestyl
e='--')
plt.plot(val_close.index, aes_predf, label='AES Forecast', color='red')

plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('Close Price - Training Data, Actual Data, and AES Forecast')
plt.legend()
plt.grid(True)
plt.show()


# AES forecasting model performed better than ARIMA model.
# But compared to the validation dataset, AES did poorly overall.
```

Logistic Regression

```
# Pull 1 year Amazon history log:

amzn_1 = yf.Ticker("AMZN")
hist0 = amzn_1.history(period = '1y')


# Create new predictors and outcome variables:

hist0['open_close'] = hist0['Close'] - hist0['Open']
hist0['positive'] = np.where(hist0['open_close'] > 0, 1, 0)

hist0['high_low'] = hist0['High'] - hist0['Low']
hist0 = hist0.drop(['Dividends', 'Stock Splits'], axis=1)

hist0.head()

amzn_desc = hist0.copy()
amzn_desc['Date'] = pd.to_datetime(amzn_desc.index)
amzn_desc.insert(0, 'day_of_week', amzn_desc['Date'].dt.day_name())
amzn_desc.head()

target_y = 'open_close'
column_x = 'day_of_week'

plt.figure(figsize=(7, 6))
sns.boxplot(x=column_x, y=target_y, data=amzn_desc)
sns.set_style("whitegrid")
plt.title("AMZN Intraday Price Change (USD) vs. Day of Week")
plt.xlabel("Day of Week")
plt.ylabel("Intraday Price Change (USD)")
plt.show()

day_week_stats = amzn_desc.groupby('day_of_week').describe().transpose()
display(day_week_stats.loc['open_close'])
```

Wednesdays appear to have the most volatile price movements at $2.26 standard deviation with Monday and Tuesday showing the lowest volatility.

Recommendation: Buy/Sell Amazon.com stock early in the week to minimize likelihood of Wednesday volatility.

```
hist0_lag = hist0.copy()

lag = 3

hist0_lag = hist0_lag.diff(periods=lag)

for i in range(1, lag+1):
    for col in hist0_lag.columns:
        lag_col_name = f'{col}_lag{i}'
```

```
        hist0_lag[lag_col_name] = hist0_lag[col].shift(i)

hist0_lag = hist0_lag.dropna()

pd.set_option('display.max_columns', 70)
display(hist0_lag.head())

# Data partition for logistic regression:

y1 = hist0['positive']
X1 = hist0_lag.drop(['positive'], axis=1)
y1 = y1.reindex(X1.index)

end_train_index1 = 200
X1_train = X1.iloc[:end_train_index1]
X1_valid = X1.iloc[end_train_index1:]

y1_train = y1.iloc[:end_train_index1]
y1_valid = y1.iloc[end_train_index1:]


# Logistic regression model and fitting:

logreg_model1 = LogisticRegression()
logreg_model1.fit(X1_train,y1_train)


# Model Performance

logreg_pred1 = logreg_model1.predict(X1_valid)
logreg_pred1 = pd.Series(logreg_pred1, index=X1_valid.index)
y1_valid = y1_valid.reindex(logreg_pred1.index)

cm1 = confusion_matrix(y1_valid, logreg_pred1, labels=logreg_model1.classes_)
cmd1 = ConfusionMatrixDisplay(confusion_matrix=cm1, display_labels=logreg_mod
el1.classes_)
cmd1.plot()

print(classification_report(y1_valid, logreg_pred1))
```

Recurrent Neutal Network: Simple RNN and Dense

```
#!pip install tensorflow

import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN
from tensorflow.keras.optimizers import Adam
```

```
# Normalize the data using Min-Max scaling
scaler = MinMaxScaler(feature_range=(0, 1))
ts_scaled = scaler.fit_transform(amzn_dfa.values.reshape(-1, 1))

# Prepare the data for training
def create_sequences(amzn_dfa, seq_length):
    sequences = []
    targets = []
    for i in range(len(amzn_dfa) - seq_length):
        seq = amzn_dfa[i:i+seq_length]
        target = amzn_dfa[i+seq_length]
        sequences.append(seq)
        targets.append(target)
    return np.array(sequences), np.array(targets)

sequence_length = 10  # You can adjust this parameter based on your needs
X, y = create_sequences(ts_scaled, sequence_length)

# Split the data into training and testing sets
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Build the RNN model
model = Sequential()
model.add(SimpleRNN(units=50, activation='relu', input_shape=(sequence_length
, 1)))
model.add(Dense(units=1, activation='linear'))


# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001), loss='mean_absolute_percen
tage_error')

# Train the model
model.fit(X_train, y_train, epochs=50, batch_size=16, verbose=1)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Inverse transform the predictions and actual values to the original scale
y_pred_inv = scaler.inverse_transform(y_pred)
y_test_inv = scaler.inverse_transform(y_test.reshape(-1, 1))


# Calculate and print the metrics
aes_eval_metricsf = ts_eval_metrics(y_test_inv, y_pred_inv)

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(amzn_dfa.index[train_size + sequence_length:], y_test_inv, label='Ac
```

```
tual', marker='.', color = 'gray')
plt.plot(amzn_dfa.index[train_size + sequence_length:], y_pred_inv, label='Pr
edicted', marker='.', color = 'pink')
plt.title('Amazon Time Series Forecasting with RNN')
plt.xlabel('Time')
plt.ylabel('Closing Stock Price, $')
plt.legend()
plt.show()

# RNN forecast model did really well in predicting the test dataset as shown
below.
```

References:

Brownlee J. (2020, April 12). A gentle introduction to exponential smoothing for time series forecasting in Python. Machine Learning Mastery. https://machinelearningmastery.com/exponential-smoothing-for-time-series-forecasting-in-python/

Brownlee, J. (2020, August 28). How to grid search triple exponential smoothing for time series forecasting in Python. Machine Learning Mastery. https://machinelearningmastery.com/how-to-grid-search-triple-exponential-smoothing-for-time-series-forecasting-in-python/

OpenAI. (2023). ChatGPT.). http://www.openai.com/product/chatgpt

Chaudhari, S. (2021, February 11). Stationarity in time series analysis explained using Python. Mathematics and Econometrics. https://blog.quantinsti.com/stationarity

ritvikmath. (2020, August 27). Seasonal-trend decomposition using LOESS. Github. https://github.com/ritvikmath/Time-Series-Analysis/blob/master/STL%20Decomposition.ipynb

Shmueli, G. (2016). ARIMA models [Video]. YouTube. https://youtu.be/0xHf-SJ9Z9U?feature=shared

statsmodels. (2023, May 05). statsmodels.tsa.holtwinters.ExponentialSmoothing. statsmodels. https://www.statsmodels.org/stable/generated/statsmodels.tsa.holtwinters.ExponentialSmoothing.html#

statsmodels. (2023, May 05). Time series analysis. statsmodels. https://www.statsmodels.org/stable/tsa.html

statsmodels. (2023, December 08). Graphics. statsmodels. https://www.statsmodels.org/devel/graphics.html#time-series-plots

TensorFlow. (2023, September 27). tf.keras.Sequential. TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/Sequential

University of San Diego. (n.d.). Lab 1.2: Model selection. University of San Diego. https://sandiego.instructure.com/courses/847/pages/lab-1-dot-2-model-selection?module_item_id=226903