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

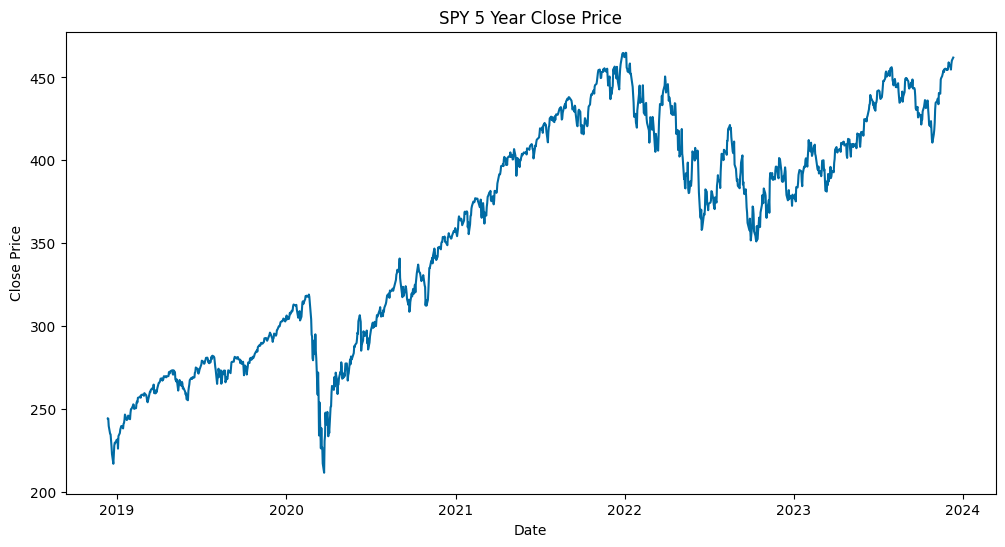
Time Series Eval Metrics Method

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\_percent\_diff))  
print('variance1=%f, variance2=%f, var\_percent\_diff=%f' % (var1, var2, var\_percent\_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%': -2.5680044958343604}

##### 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

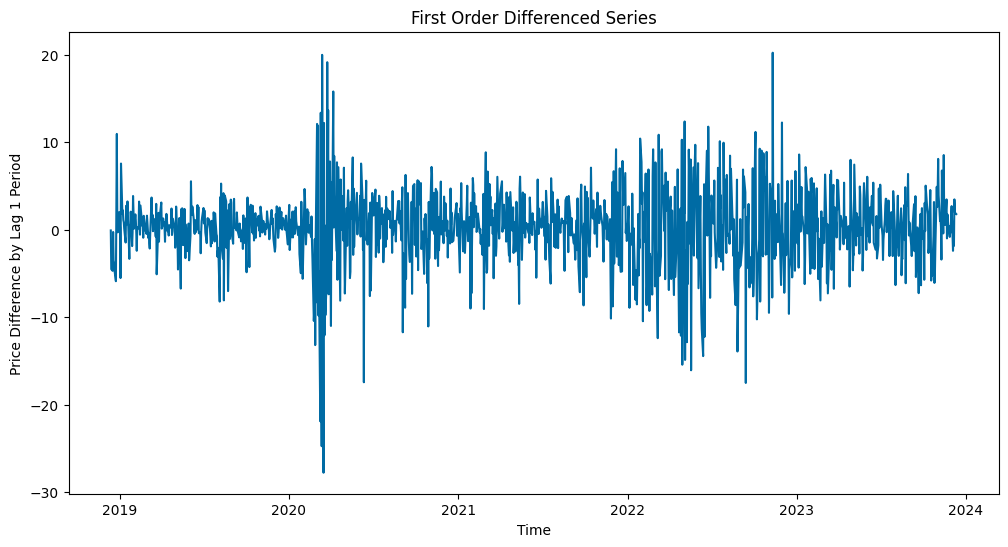
result = kpss(X)  
print(result)  
print('KPSS Test Statistic: %.2f' % result[0])  
print('p-value: %f' % result[1])  
print('Critical Values:')  
print(result[3])

(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%': -2.5680044958343604}

(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, seasonal\_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 time 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

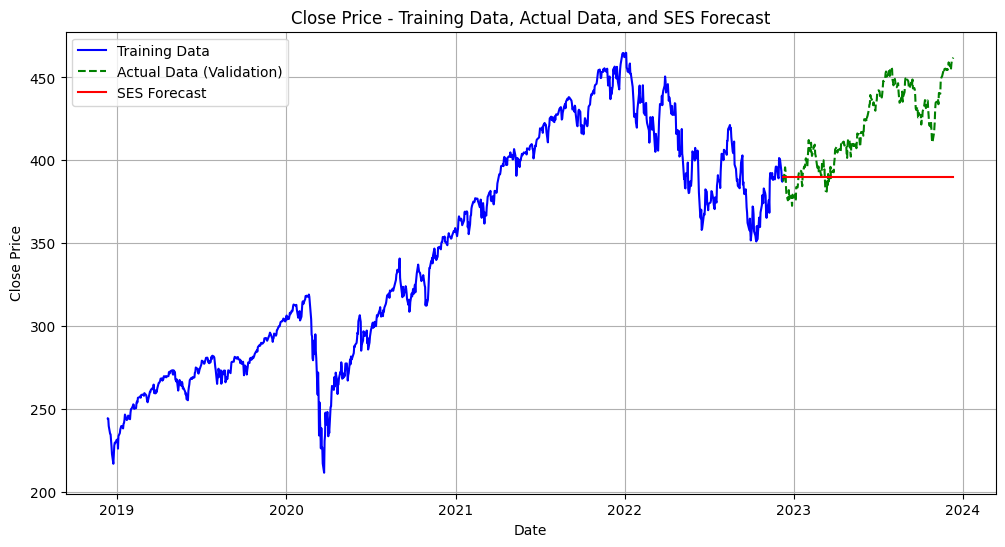
#### Plot an initial time series

##### 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', linestyle='--')  
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()



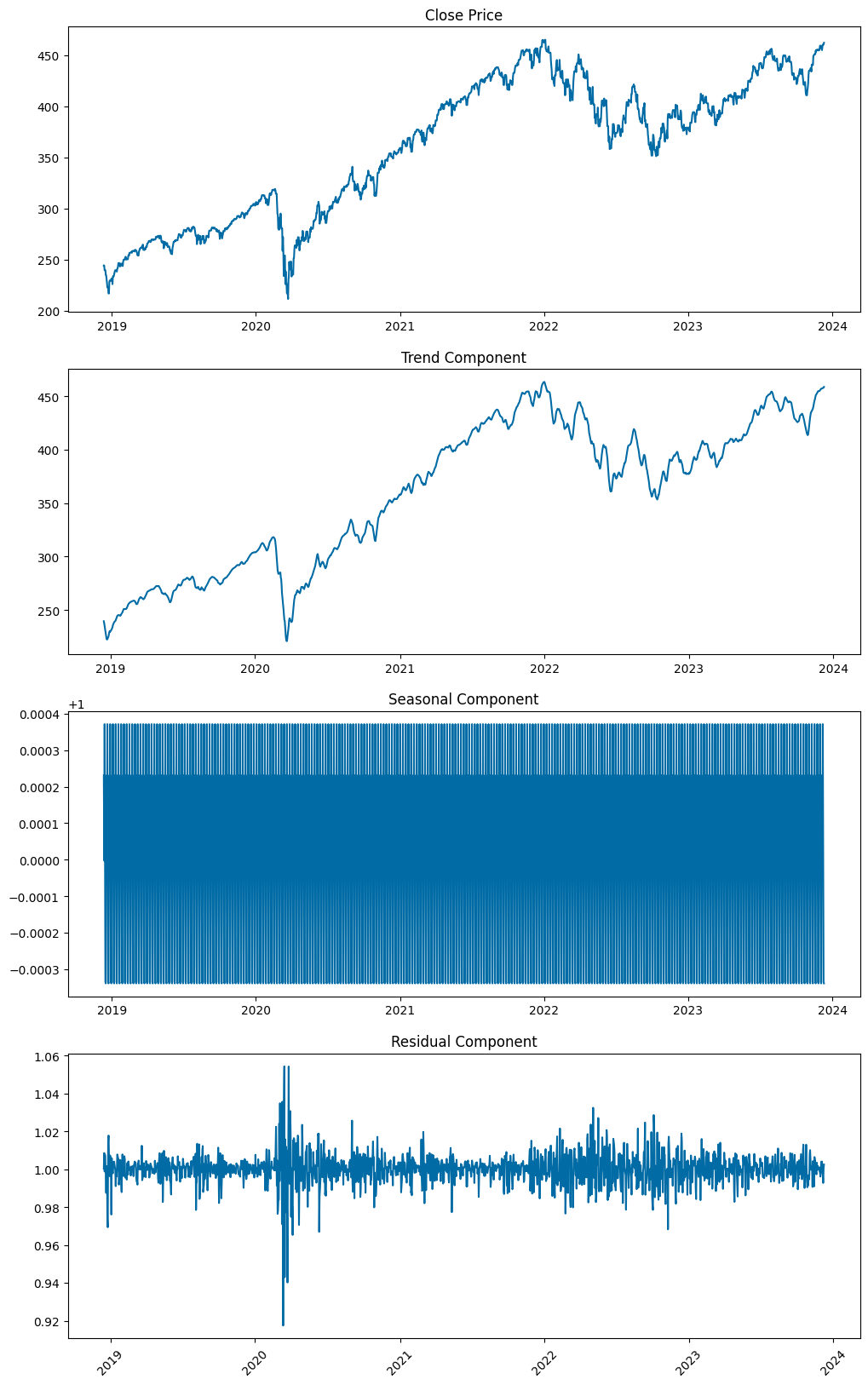
Simple Exponential Smoothing RMSE is 34.56% and it is higher to our success rate criteria. The forecast also doesn't include trend, seasonality and noise

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

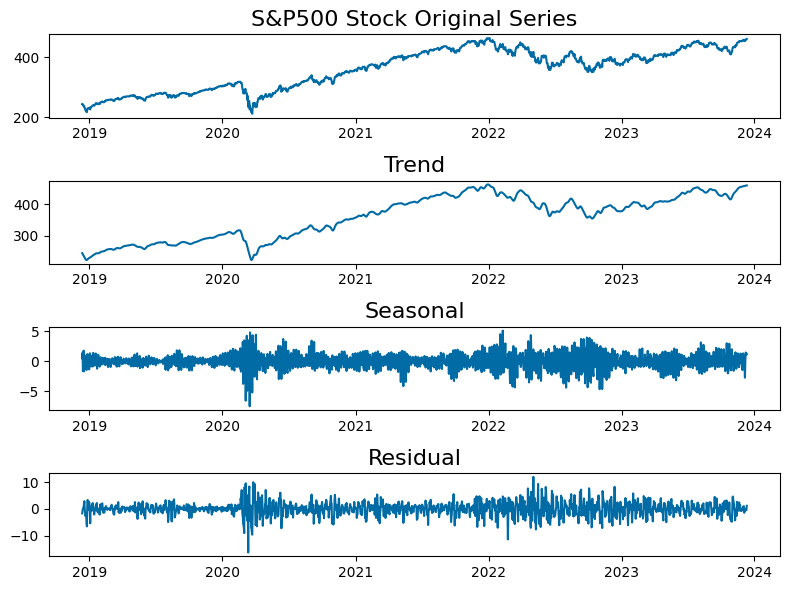
## Decomposition of Raw Values

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>]



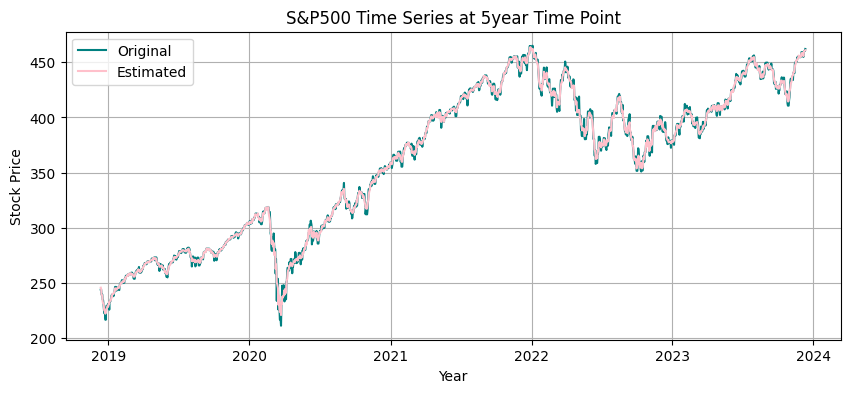
stl\_close = STL(close)  
stl\_close\_f = stl\_close.fit()  
  
  
# Plot decomposition:  
  
plt.figure(figsize=(8,6))  
  
plt.subplot(4,1,1)  
plt.plot(close)  
plt.title('S&P500 Stock Original Series', fontsize=16)  
  
plt.subplot(4,1,2)  
plt.plot(stl\_close\_f.trend)  
plt.title('Trend', fontsize=16)  
  
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

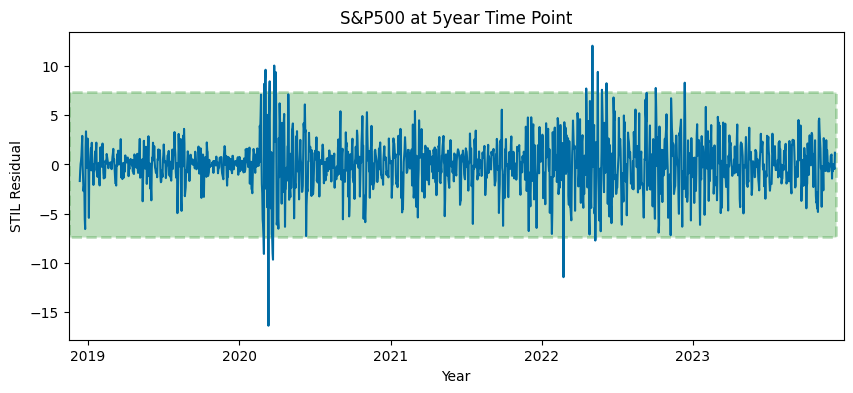
estimated0 = stl\_close\_f.trend + stl\_close\_f.seasonal  
plt.figure(figsize=(10,4))  
plt.plot(close, label='Original', color = 'teal')  
plt.plot(estimated0, label ='Estimated', color = 'pink')  
  
  
plt.xlabel('Year')  
plt.ylabel('Stock Price')  
plt.title('S&P500 Time Series at 5year Time Point')  
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')

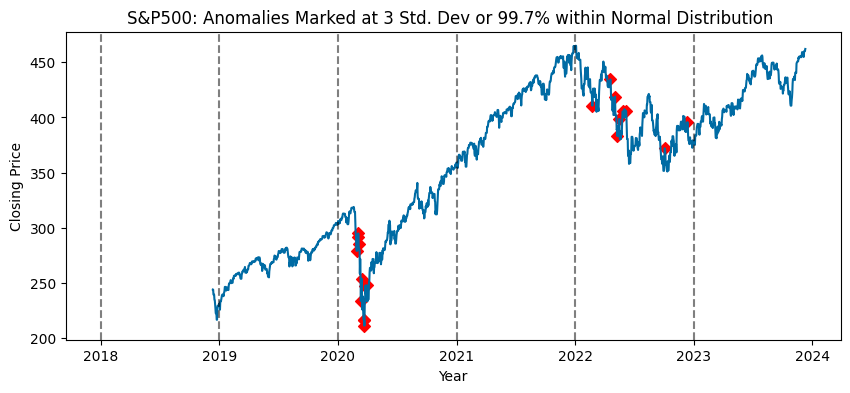
Text(0.5, 1.0, 'S&P500 at 5year Time Point')



# 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 Distribution ')

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

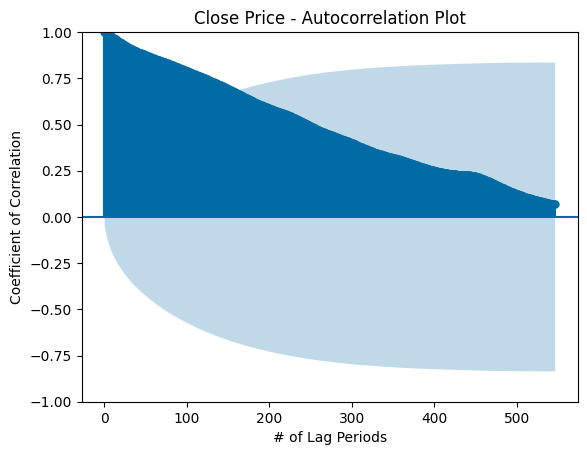
stl\_close.config, stl\_close.period

({'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

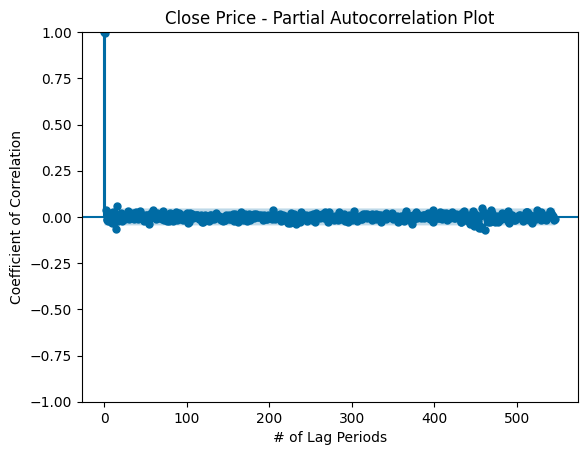
Reference: <https://www.statsmodels.org/devel/graphics.html#time-series-plots>

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()

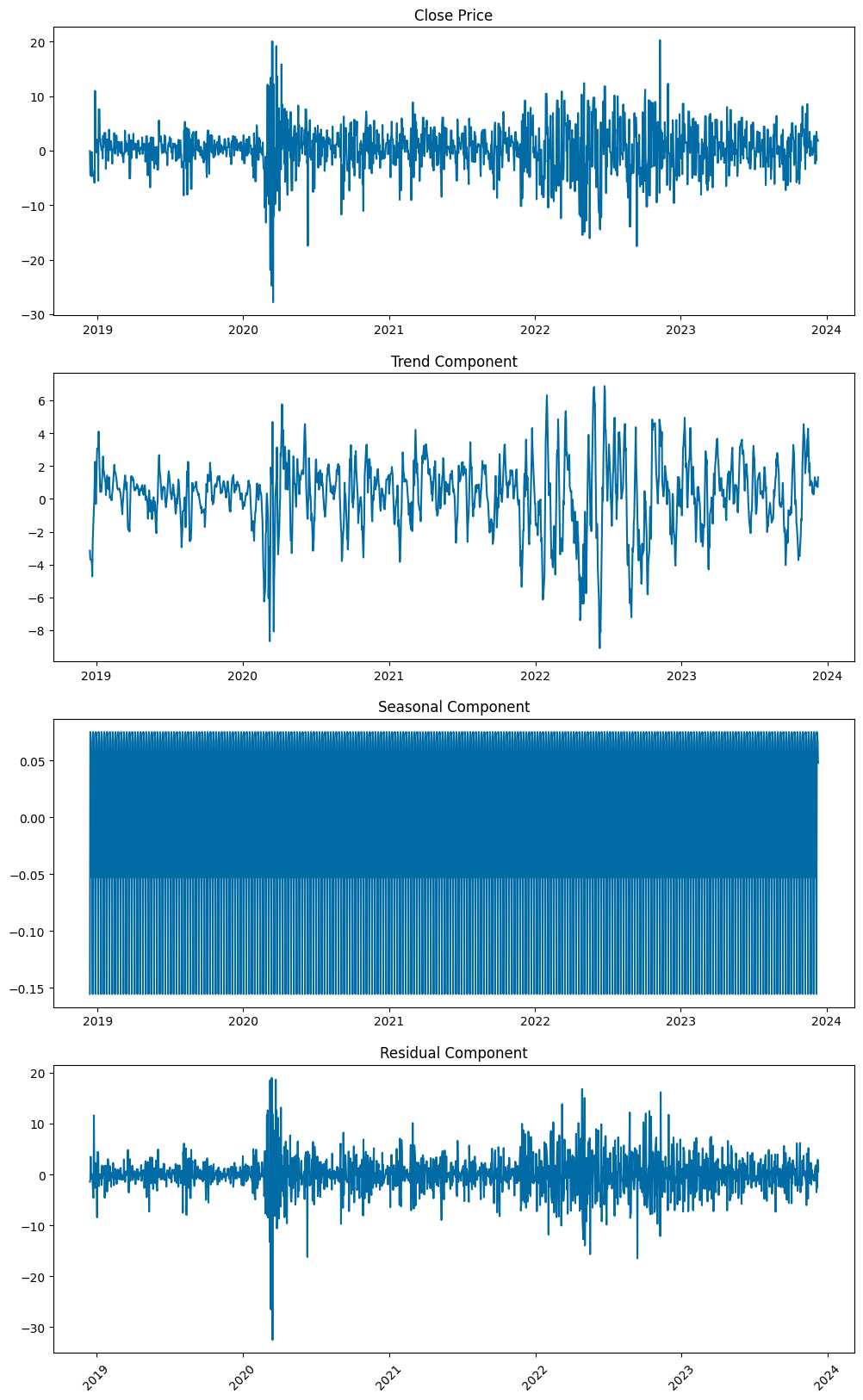


### Therefore, based on PACF plot, we may want to do AR model with lags 1, 2 ~415, ~485, ~510.

## Lag\_1 Decomposition

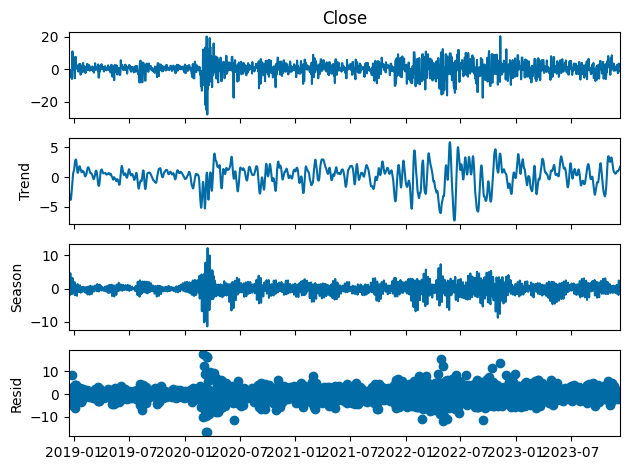
decomposition\_lag\_1 = seasonal\_decompose(ts\_lag\_1, model='additive')  
#decomposition.plot()  
  
trend\_lag\_1 = decomposition\_lag\_1.trend  
seasonal\_lag\_1 = decomposition\_lag\_1.seasonal  
residual\_lag\_1 = decomposition\_lag\_1.resid  
  
fig, axs = plt.subplots(4)  
fig.set\_figheight(20)  
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>]



### Decompose using STL

stl = STL(ts\_lag\_1)  
stl\_plot = stl.fit().plot()



stl.config, stl.period

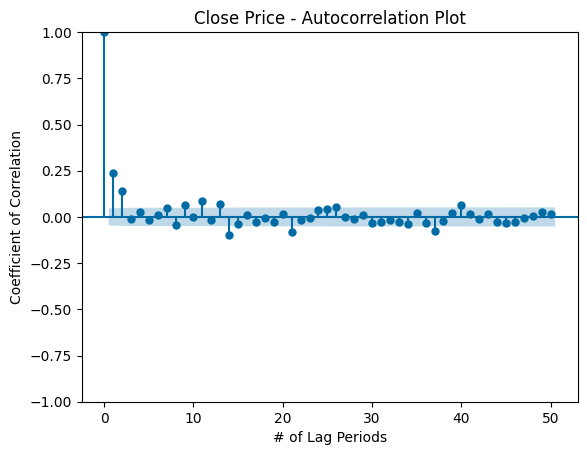
({'period': 7,  
 'seasonal': 7,  
 'seasonal\_deg': 1,  
 'seasonal\_jump': 1,  
 'trend': 15,  
 'trend\_deg': 1,  
 'trend\_jump': 1,  
 'low\_pass': 9,  
 'low\_pass\_deg': 1,  
 'low\_pass\_jump': 1,  
 'robust': False},  
 7)

## 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

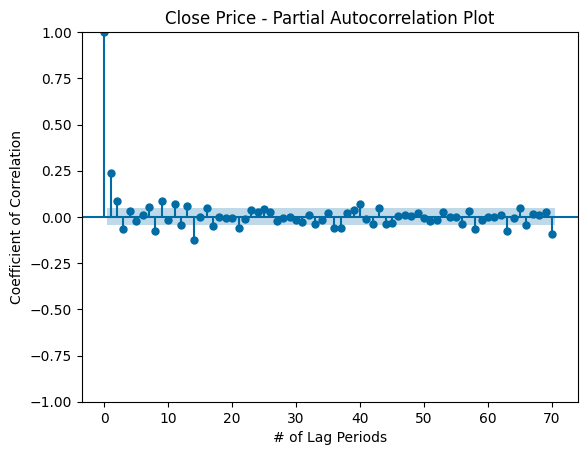
plot\_acf(ts\_lag\_1, lags=50) # Adjust the number of lags as needed  
plt.xlabel('# of Lag Periods')  
plt.ylabel('Coefficient of Correlation')  
plt.title('Close Price - Autocorrelation Plot')  
plt.show()



### 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 PACF  
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 statsmodels.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.holtwinters.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\_square']  
  
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', 'multiplicative', None  
 seasonal\_periods= i,  
 initialization\_method='heuristic'  
 ) #'estimated', 'heuristic', 'legacy-heuristic'  
 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', 'multiplicative', None  
 seasonal\_periods= 3,  
 initialization\_method='heuristic'  
 ) #'estimated', 'heuristic', 'legacy-heuristic'  
 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', 'multiplicative', None  
 #damped\_trend=True, #True, False  
 seasonal= 'add', # 'mul', 'additive', 'multiplicative', None  
 seasonal\_periods= 3,  
 initialization\_method='heuristic'  
 ) #'estimated', 'heuristic', 'legacy-heuristic'  
 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', 'multiplicative', None  
 seasonal\_periods= 3,  
 initialization\_method='heuristic'  
 ) #'estimated', 'heuristic', 'legacy-heuristic'  
 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', 'multiplicative', None  
 seasonal\_periods= 3,  
 initialization\_method=i  
 ) #'estimated', 'heuristic', 'legacy-heuristic'  
 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', 'multiplicative', None  
 damped\_trend=True, #True, False  
 seasonal= 'add', # 'mul', 'additive', 'multiplicative', None  
 seasonal\_periods= 3,  
 initialization\_method='heuristic') #'estimated', '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', linestyle='--')  
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', 'multiplicative', None  
 damped\_trend=True, #True, False  
 seasonal= 'add', # 'mul', 'additive', 'multiplicative', None  
 seasonal\_periods= 2,  
 initialization\_method='heuristic') #'estimated', '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', linestyle='--')  
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 quadratic 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 off it doesnt work  
plt.plot(close\_valid, label='Actual Data (Validation)', color='green', linestyle='--')  
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 difference  
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) # Clean 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 range  
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, index=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=2000, 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.classes\_)  
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\_iter=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.classes\_)  
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, random\_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 = 'pink')  
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 Distribution')  
  
  
# Plot shows anomalies detected outside of +/- 3 Std Dev of normal distribution 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 = - 2

AIC = 2 - 2

() = a log likelihood

() = a number of parameters

() = 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', linestyle='--')  
plt.plot(val\_close.index, arima\_pred1, label='Auto ARIMA Forecast', color='red')  
  
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 dataset.

#### 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', linestyle='--')  
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 compared 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-heuristic']  
  
  
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 poor 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', linestyle='--')  
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\_model1.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\_percentage\_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='Actual', marker='.', color = 'gray')  
plt.plot(amzn\_dfa.index[train\_size + sequence\_length:], y\_pred\_inv, label='Predicted', 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>