In [2]:
```
!pip install pmdarima
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting pmdarima
  Downloading pmdarima-2.0.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.manylinux_2_28_x86_64.whl (1.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 59.0 MB/s eta 0:00:00
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (1.2.0)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (1.26.15)
Requirement already satisfied: setuptools!=50.0.0,>=38.6.0 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (67.7.2)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (1.22.4)
Requirement already satisfied: statsmodels>=0.13.2 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (0.13.5)
Requirement already satisfied: Cython!=0.29.18,!=0.29.31,>=0.29 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (0.29.34)
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (1.2.2)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (1.10.1)
Requirement already satisfied: pandas>=0.19 in /usr/local/lib/python3.10/dist-packages (from pmdarima) (1.5.3)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.19->pmdarima) (2022.7.1)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.19->pmdarima) (2.8.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.22->pmdarima) (3.1.0)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.10/dist-packages (from statsmodels>=0.13.2->pmdarima) (23.1)
Requirement already satisfied: patsy>=0.5.2 in /usr/local/lib/python3.10/dist-packages (from statsmodels>=0.13.2->pmdarima) (0.5.3)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from patsy>=0.5.2->statsmodels>=0.13.2->pmdarima) (1.16.0)
Installing collected packages: pmdarima
Successfully installed pmdarima-2.0.3
```

In [3]:
```
import os
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose

from pmdarima.arima import auto_arima
from statsmodels.tsa.arima.model import ARIMA

from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
```

```
from sklearn.metrics import mean_squared_error, mean_absolute_error
import math
```

In [137…
```
dateparse = lambda dates: pd.datetime.strptime(dates, '%Y-%m-%d')
stock_data=pd.read_table('/content/sample_data/acgl.us.txt',sep=',',index_col='Date',p
```

In [139…
```
stock_data.head()
```

Out[139]:

Date	Open	High	Low	Close	Volume	OpenInt
2005-02-25	13.583	13.693	13.430	13.693	156240	0
2005-02-28	13.697	13.827	13.540	13.827	370509	0
2005-03-01	13.780	13.913	13.720	13.760	224484	0
2005-03-02	13.717	13.823	13.667	13.810	286431	0
2005-03-03	13.783	13.783	13.587	13.630	193824	0

In [139…

In [140…
```
stock_data.describe()
```

Out[140]:

	Open	High	Low	Close	Volume	OpenInt
count	3201.000000	3201.000000	3201.000000	3201.000000	3.201000e+03	3201.0
mean	41.800246	42.127681	41.465992	41.826403	7.830475e+05	0.0
std	23.728163	23.847994	23.593876	23.730770	6.482674e+05	0.0
min	13.170000	13.270000	11.010000	13.180000	5.236800e+04	0.0
25%	22.653000	22.907000	22.370000	22.643000	3.591570e+05	0.0
50%	33.100000	33.430000	32.800000	33.150000	5.823510e+05	0.0
75%	57.550000	57.890000	57.240000	57.550000	1.033461e+06	0.0
max	102.450000	102.600000	101.840000	102.380000	1.388090e+07	0.0

In [141…
```
stock_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3201 entries, 2005-02-25 to 2017-11-10
Data columns (total 6 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   Open     3201 non-null   float64
 1   High     3201 non-null   float64
 2   Low      3201 non-null   float64
 3   Close    3201 non-null   float64
 4   Volume   3201 non-null   int64  
 5   OpenInt  3201 non-null   int64  
dtypes: float64(4), int64(2)
memory usage: 175.1 KB
```

In [142…
```
stock_data.isna().sum()
```

Out[142]:
```
Open       0
High       0
Low        0
Close      0
Volume     0
OpenInt    0
dtype: int64
```

In [143…
```
df_close=stock_data['Close']
```

In [144…
```
df_close=pd.DataFrame(df_close)
```

In [145…
```
df_close.head()
```

Out[145]:

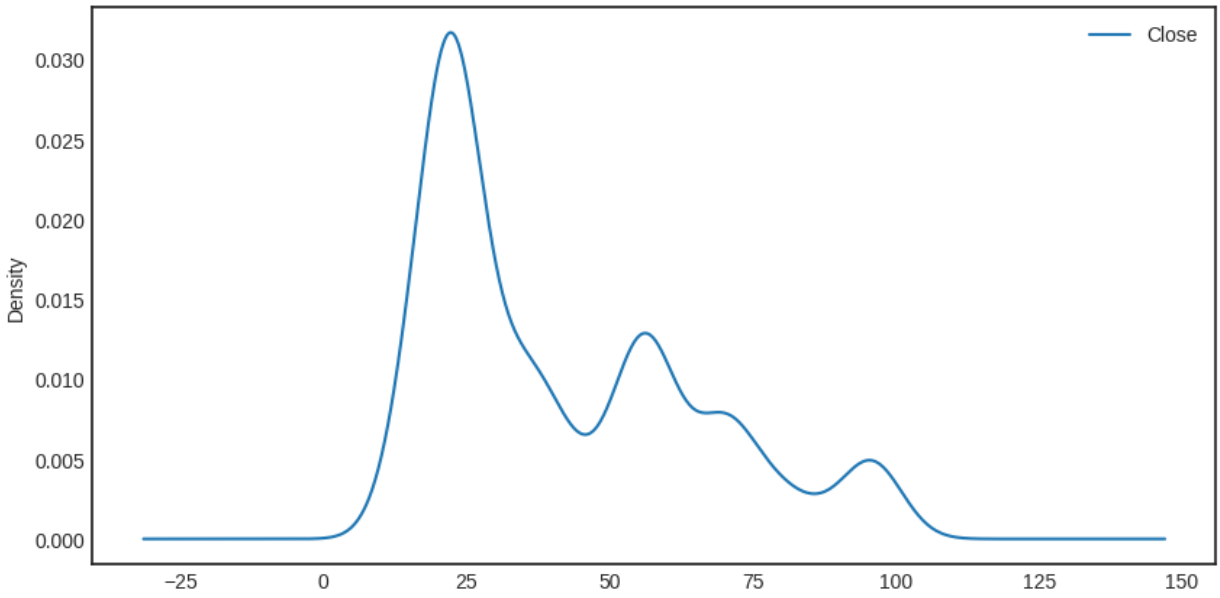
	Close
Date	
2005-02-25	13.693
2005-02-28	13.827
2005-03-01	13.760
2005-03-02	13.810
2005-03-03	13.630

In [146…
```
#Visualize the per day closing price of the stock.

#plot close price
plt.figure(figsize=(8,5))
plt.grid(True)
plt.xlabel('Date')
plt.ylabel('Close Prices')
plt.plot(df_close,c='b')
plt.title('ARCH CAPITAL GROUP closing price',c='k')
plt.show()
```

In [147…

```
#We can also visualize the data in our series through a probability distribution too.
#Distribution of the dataset
df_close.plot(kind='kde')
plt.show()
```



Also, a given time series is thought to consist of three systematic components including level, trend, seasonality, and one non-systematic component called noise.

These components are defined as follows:

Level: The average value in the series.

Trend: The increasing or decreasing value in the series.

Seasonality: The repeating short-term cycle in the series.

Noise: The random variation in the series.

First, we need to check if a series is stationary or not because time series analysis only works with stationary data.

ADF (Augmented Dickey-Fuller) Test

The Dickey-Fuller test is one of the most popular statistical tests. It can be used to determine the presence of unit root in the series, and hence help us understand if the series is stationary or not. The null and alternate hypothesis of this test is:

Null Hypothesis: The series has a unit root (value of a =1)

Alternate Hypothesis: The series has no unit root.

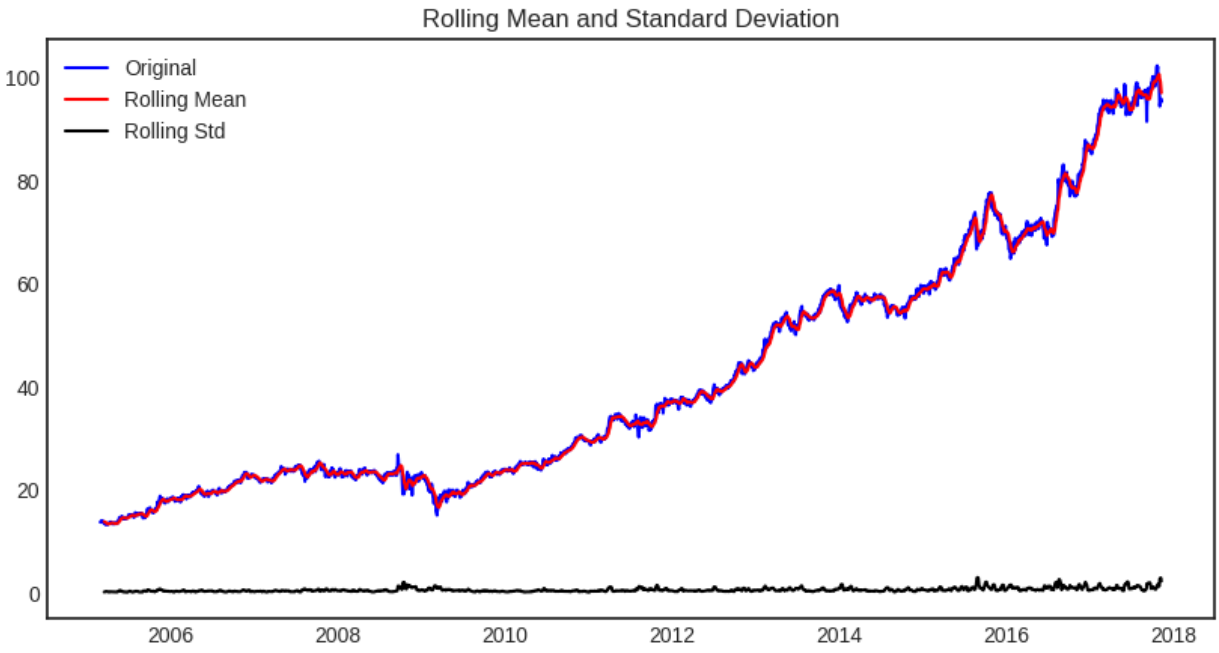
If we fail to reject the null hypothesis, we can say that the series is non-stationary. This means that the series can be linear or difference stationary.

If both mean and standard deviation are flat lines(constant mean and constant variance), the series becomes stationary.

**So let's check for stationarity:

In [148…
```
#Test for staionarity
def test_stationarity(timeseries):
    #Determing rolling statistics
    rolmean = timeseries.rolling(12).mean()
    rolstd = timeseries.rolling(12).std()
    #Plot rolling statistics:
    plt.plot(timeseries, color='blue',label='Original')
    plt.plot(rolmean, color='red', label='Rolling Mean')
    plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean and Standard Deviation')
    plt.show(block=False)
    print("Results of dickey fuller test")
    adft = adfuller(timeseries,autolag='AIC')
    # output for dft will give us without defining what the values are.
    #hence we manually write what values does it explains using a for loop
    output = pd.Series(adft[0:4],index=['Test Statistics','p-value','No. of lags used'
    for key,values in adft[4].items():
        output['critical value (%s)'%key] =  values
    print(output)
```

In [149…
```
test_stationarity(df_close)
#Results of dickey fuller test
```




```
Results of dickey fuller test
Test Statistics                  1.374899
p-value                          0.996997
No. of lags used                 5.000000
Number of observations used    3195.000000
critical value (1%)              -3.432398
critical value (5%)              -2.862445
critical value (10%)             -2.567252
dtype: float64
```

Through the above graph, we can see the increasing mean and standard deviation and hence our series is not stationary.

We see that the p-value is greater than 0.05 so we cannot reject the Null hypothesis. Also, the test statistics is greater than the critical values. so the data is non-stationary.

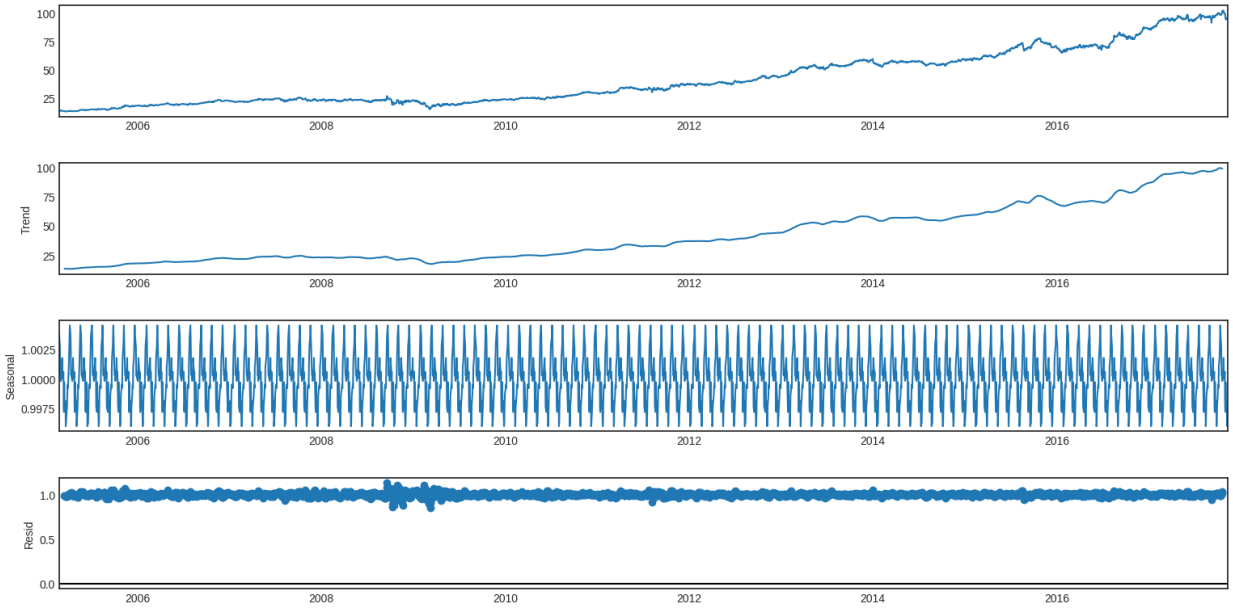
In order to perform a time series analysis, we may need to separate seasonality and trend from our series. The resultant series will become stationary through this process.

So let us separate Trend and Seasonality from the time series.

In [150…

```
#To separate the trend and the seasonality from a time series,
# we can decompose the series using the following code.
result = seasonal_decompose(df_close, model='multiplicative',period=30)
fig = plt.figure()
fig = result.plot()
fig.set_size_inches(16, 8)
plt.show()
```

```
<Figure size 1000x500 with 0 Axes>
```

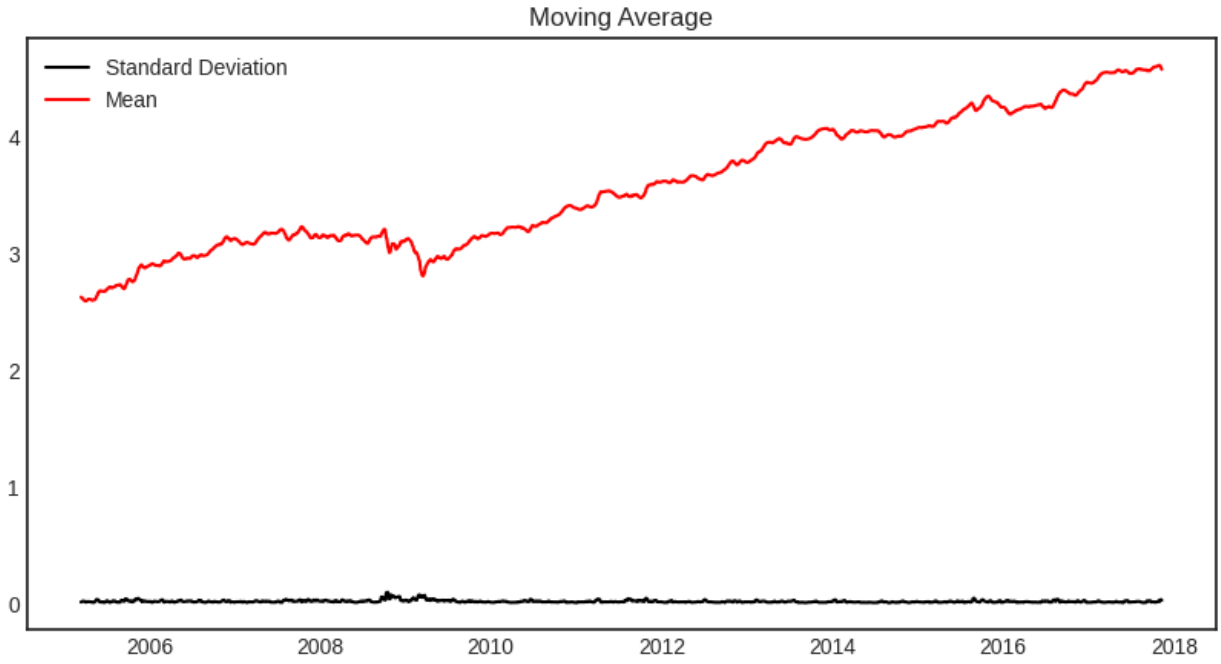


We start by taking a log of the series to reduce the magnitude of the values and reduce the rising trend in the series. Then after getting the log of the series, we find the rolling average of the series. A rolling average is calculated by taking input for the past 12 months and giving a mean consumption value at every point further ahead in series.

In [151…

```
#if not stationary then eliminate trend
#Eliminate trend
from pylab import rcParams
rcParams['figure.figsize'] = 10, 5
df_log = np.log(df_close)
moving_avg = df_log.rolling(12).mean()
std_dev = df_log.rolling(12).std()
plt.legend(loc='best')
plt.title('Moving Average')
plt.plot(std_dev, color ="black", label = "Standard Deviation")
plt.plot(moving_avg, color="red", label = "Mean")
plt.legend()
plt.show()
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that a
rtists whose label start with an underscore are ignored when legend() is called with
no argument.
```



Now we are going to create an ARIMA model and will train it with the closing price of the stock on the train data. So let us split the data into training and test set and visualize it.

In [152…
```
#split data into train and training set
train_data, test_data = df_log[3:int(len(df_log)*0.9)], df_log[int(len(df_log)*0.9):]
plt.figure(figsize=(10,5))
plt.grid(True)
plt.xlabel('Dates')
plt.ylabel('Closing Prices')
plt.plot(df_log, 'green', label='Train data')
plt.plot(test_data, 'blue', label='Test data')
plt.legend()
```

Out[152]:
```
<matplotlib.legend.Legend at 0x7fa34b3ea710>
```

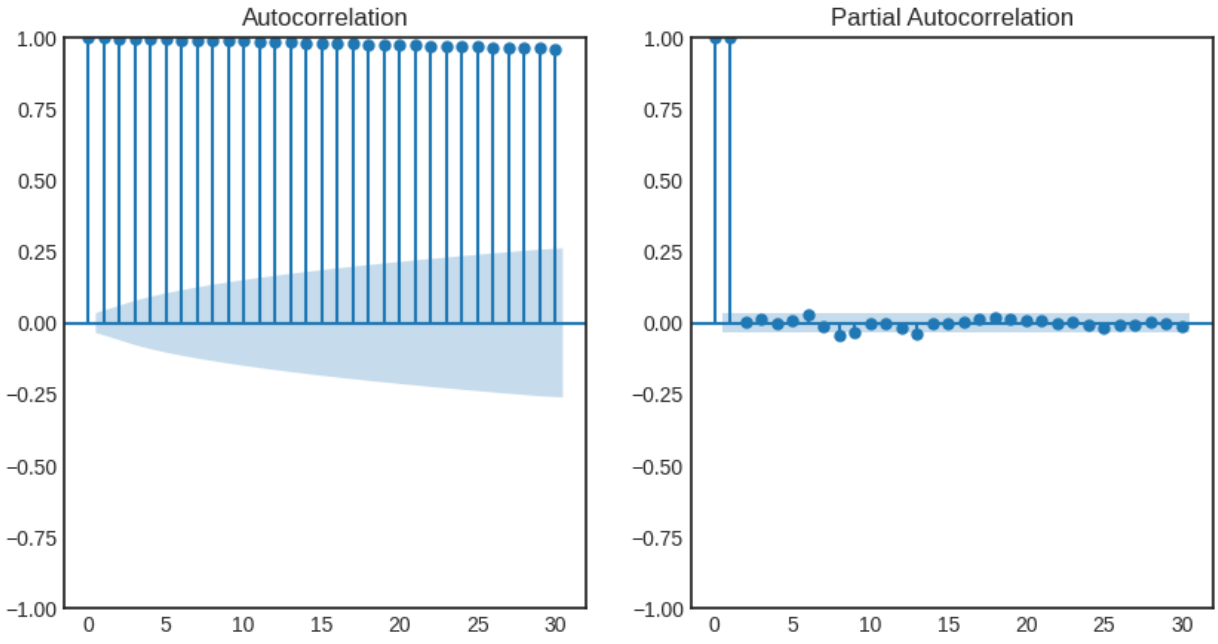
Its time to choose parameters p,q,d for ARIMA model. Last time we chose the value of p,d, and q by observing the plots of ACF and PACF but now we are going to use Auto ARIMA to get the best parameters without even plotting ACF and PACF graphs.

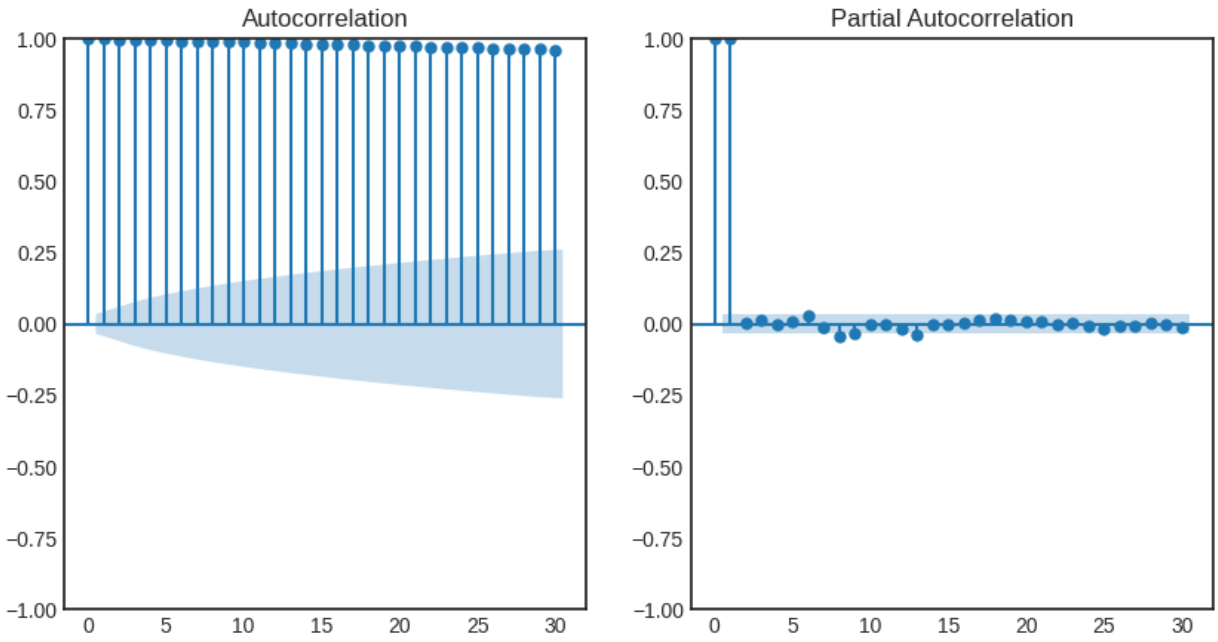
Auto ARIMA: Automatically discover the optimal order for an ARIMA model. The auto_arima function seeks to identify the most optimal parameters for an ARIMA model, and returns a fitted ARIMA model. This function is based on the commonly-used R function, forecast::auto.arima.

The auro_arima function works by conducting differencing tests (i.e., Kwiatkowski–Phillips–Schmidt–Shin, Augmented Dickey-Fuller or Phillips–Perron) to determine the order of differencing, d, and then fitting models within ranges of defined start_p, max_p, start_q, max_q ranges. If the seasonal optional is enabled, auto_arima also seeks to identify the optimal P and Q hyper- parameters after conducting the Canova-Hansen to determine the optimal order of seasonal differencing, D.

In [153…
```
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
plot_acf(df_close,lags=30,ax=ax[0])
plot_pacf(df_close,lags=30,ax=ax[1])
```

Out[153]:





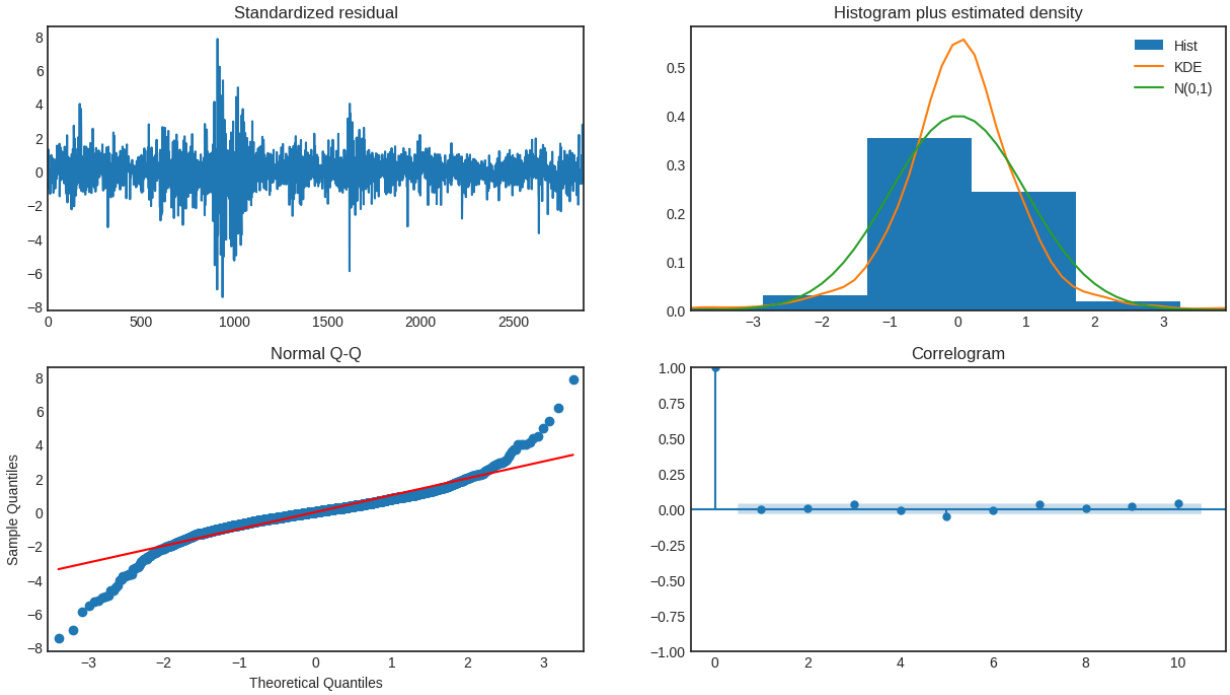
In [154…

```
model_autoARIMA = auto_arima(train_data, start_p=0, start_q=0,
                      test='adf',       # use adftest to find optimal 'd'
                      max_p=3, max_q=3, # maximum p and q
                      m=1,              # frequency of series
                      d=None,           # let model determine 'd'
                      seasonal=False,   # No Seasonality
                      start_P=0,
                      D=0,
                      trace=True,
                      error_action='ignore',
                      suppress_warnings=True,
                      stepwise=True)
print(model_autoARIMA.summary())
model_autoARIMA.plot_diagnostics(figsize=(15,8))
plt.show()
```

```
Performing stepwise search to minimize aic
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=-16491.508, Time=0.40 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=-16525.992, Time=0.21 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=-16527.964, Time=1.99 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=-16488.323, Time=0.28 sec
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=-16527.157, Time=4.19 sec
 ARIMA(0,1,2)(0,0,0)[0] intercept   : AIC=-16527.120, Time=1.35 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : AIC=-16528.810, Time=1.90 sec
 ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=inf, Time=3.97 sec
 ARIMA(1,1,3)(0,0,0)[0] intercept   : AIC=-16526.020, Time=3.67 sec
 ARIMA(0,1,3)(0,0,0)[0] intercept   : AIC=-16524.974, Time=5.98 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=-16525.435, Time=1.33 sec
 ARIMA(2,1,3)(0,0,0)[0] intercept   : AIC=-16516.417, Time=0.93 sec
 ARIMA(1,1,2)(0,0,0)[0]             : AIC=-16527.597, Time=0.87 sec

Best model:  ARIMA(1,1,2)(0,0,0)[0] intercept
Total fit time: 27.131 seconds
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 2877
Model:               SARIMAX(1, 1, 2)   Log Likelihood                8269.405
Date:                Mon, 01 May 2023   AIC                         -16528.810
Time:                        12:25:59   BIC                         -16498.989
Sample:                             0   HQIC                        -16518.061
                                - 2877
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept   2.467e-05   7.12e-06      3.464      0.001    1.07e-05    3.86e-05
ar.L1          0.9538      0.009    104.140      0.000       0.936       0.972
ma.L1         -1.0708      0.015    -73.566      0.000      -1.099      -1.042
ma.L2          0.0877      0.012      7.504      0.000       0.065       0.111
sigma2         0.0002   2.32e-06     80.805      0.000       0.000       0.000
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):              7207.33
Prob(Q):                              0.97   Prob(JB):                         0.00
Heteroskedasticity (H):               0.30   Skew:                            -0.39
Prob(H) (two-sided):                  0.00   Kurtosis:                        10.72
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```



Performing stepwise search to minimize aic

So how to interpret the plot diagnostics?

Top left: The residual errors seem to fluctuate around a mean of zero and have a uniform variance.

Top Right: The density plot suggest normal distribution with mean zero.

Bottom left: All the dots should fall perfectly in line with the red line. Any significant deviations would imply the distribution is skewed.

Bottom Right: The Correlogram, aka, ACF plot shows the residual errors are not autocorrelated. Any autocorrelation would imply that there is some pattern in the residual errors which are not explained in the model. So you will need to look for more X's (predictors) to the model.

Overall, it seems to be a good fit. Let's start forecasting the stock prices.

Next, create an ARIMA model with provided optimal parameters p, d and q.

So the Auto ARIMA model provided the value of p,d, and q as 1, 1 and 2 respectively.

In [157…

```
#Modeling
# Build Model
model = ARIMA(train_data, order=(1,1,2))
fitted = model.fit()
print(fitted.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                  Close   No. Observations:                 2877
Model:                 ARIMA(1, 1, 2)   Log Likelihood                8267.798
Date:                Mon, 01 May 2023   AIC                         -16527.597
Time:                        12:27:08   BIC                         -16503.740
Sample:                             0   HQIC                        -16518.997
                                 - 2877
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.8973      0.036     24.798      0.000       0.826       0.968
ma.L1         -1.0130      0.039    -25.731      0.000      -1.090      -0.936
ma.L2          0.0826      0.014      5.842      0.000       0.055       0.110
sigma2         0.0002   2.29e-06     81.316      0.000       0.000       0.000
===================================================================================
Ljung-Box (L1) (Q):                   0.07   Jarque-Bera (JB):              7150.64
Prob(Q):                              0.80   Prob(JB):                         0.00
Heteroskedasticity (H):               0.30   Skew:                            -0.23
Prob(H) (two-sided):                  0.00   Kurtosis:                        10.71
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

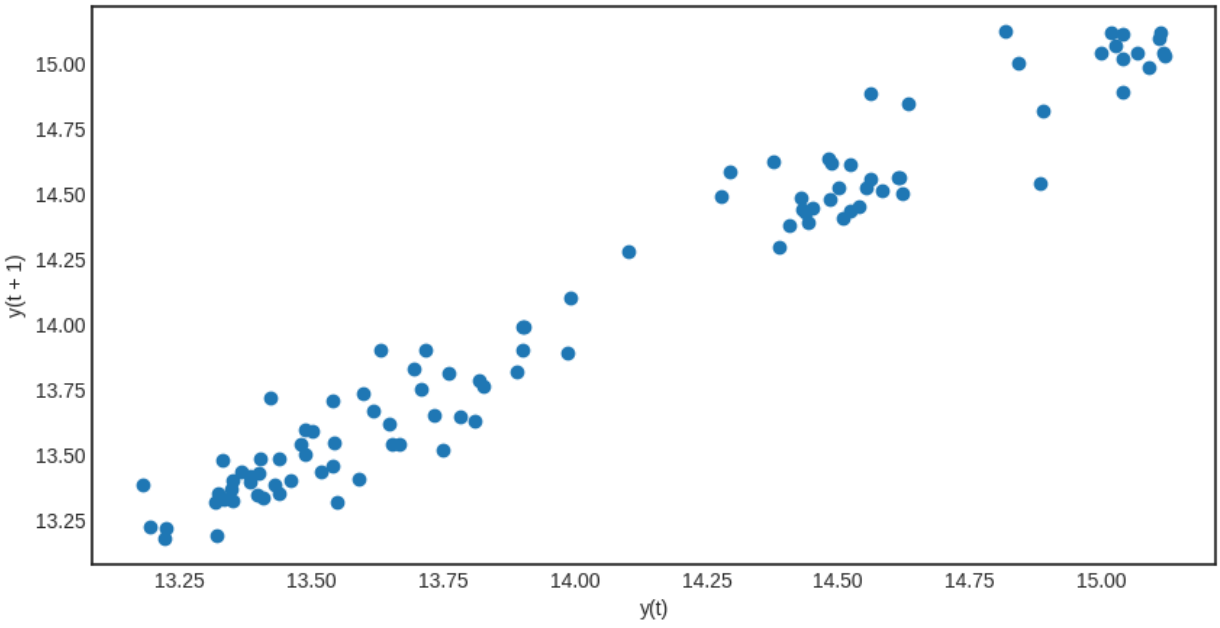
Now let's start forecast the stock prices on the test dataset keeping 95% confidence level.

In [158…

```
from pandas.plotting import lag_plot
lag_plot(df_close.iloc[:100])
```

Out[158]:

```
<Axes: xlabel='y(t)', ylabel='y(t + 1)'>
```



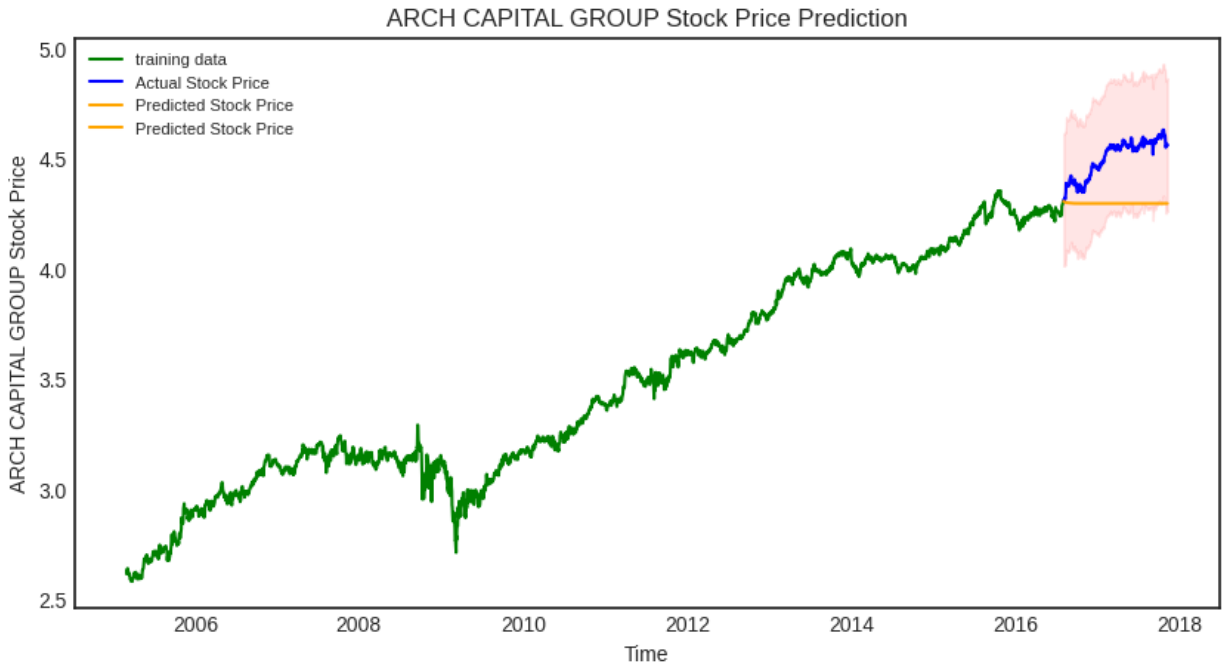
This shows evidence of a strong autocorrelation, as yt values increase, nearby (lagged) values also increase

In [219…
```
# Forecast
fc=fitted.forecast(321, alpha=0.05) # 95% conf
fc=pd.DataFrame(fc)
fc['index']=test_data.index
fc.set_index('index',inplace=True)
```

Plot the results

In [218…
```
# Make as pandas series
fc_series = fc
lower_series = test_data-0.3
upper_series = test_data+0.3
```

In [233…
```
# Plot
plt.figure(figsize=(10,5), dpi=100)
plt.plot(train_data,color='g', label='training data')
plt.plot(test_data, color = 'blue', label='Actual Stock Price')
plt.plot(fc_series, color = 'orange',label='Predicted Stock Price')
plt.fill_between(lower_series.index, lower_series['Close'], upper_series['Close'],colo
plt.title('ARCH CAPITAL GROUP Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('ARCH CAPITAL GROUP Stock Price')
plt.legend(loc='upper left', fontsize=8)
plt.show()
```



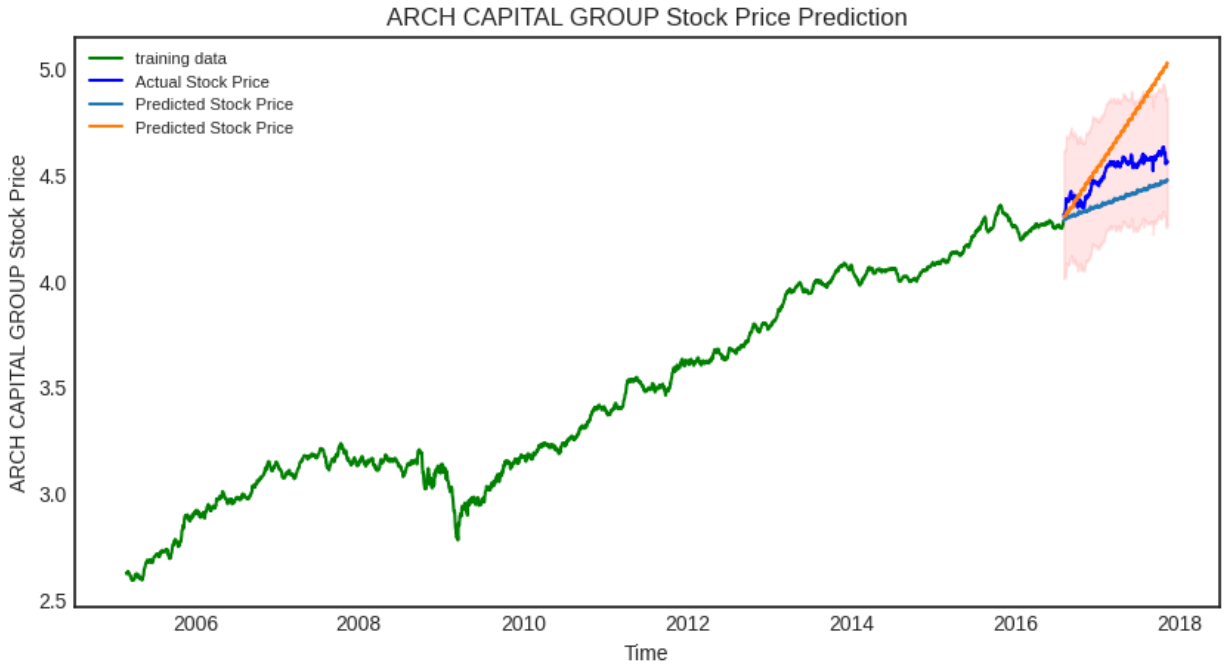
In [166…
```
from statsmodels.tsa.holtwinters import ExponentialSmoothing
```

In [272…
```
fit1=ExponentialSmoothing(train_data,seasonal_periods=36,trend='add',
                          seasonal='add').fit(smoothing_level=0.2,smoothing_trend=0.00
f1=fit1.forecast(321)
f1=pd.DataFrame(f1)
f1['index']=test_data.index
f1.set_index('index',inplace=True)
```

In [276…
```
#stepwise_fit=auto_arima(stock_data["Close"],start_p=1,start_q=1,max_p=3,
#                         max_q=3,m=12,seasonal=True,start_P=0,d=None,D=1,trace=True,
#                        error_action='ignore',suppress_warnings=True,
#                         stepwise=True)
#stepwise_fit.summary()
```

In [277…
```
mod=ARIMA(train_data,order=(0,1,1),seasonal_order=(2,1,0,12)).fit()
f2=mod.forecast(321)
f1['f2']=np.array(f2)
```

In [278…
```
plt.figure(figsize=(10,5), dpi=100)
plt.plot(fit1.fittedvalues,color='g', label='training data')
plt.plot(test_data, color = 'blue', label='Actual Stock Price')
plt.plot(f1,label='Predicted Stock Price')
plt.fill_between(lower_series.index, lower_series['Close'], upper_series['Close'],colo
plt.title('ARCH CAPITAL GROUP Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('ARCH CAPITAL GROUP Stock Price')
plt.legend(loc='upper left', fontsize=8)
plt.show()
```



As you can see our model did quite handsomely. Let us also check the commonly used accuracy metrics to judge forecast results:

In [311…
```
# report performance
mse = mean_squared_error(test_data, f1[0])
print('MSE: ',(mse))
mae = mean_absolute_error(test_data, f1[0])
print('MAE: ',(mae))
rmse = math.sqrt(mean_squared_error(test_data, f1[0]))
print('RMSE: ',(rmse))
mape = np.mean(np.abs((test_data['Close']-f1[0])/(test_data['Close'])))
print('MAPE: ',(mape)*100,'%')
```

```
MSE:  0.016781861572311093
MAE:  0.12194526552725807
RMSE:  0.12954482456783478
MAPE:  2.6934073438289263 %
```

Around 2.69% MAPE implies the model is about 97.3% accurate in predicting the next 15 observations.