Sarit Maitra

1.4K Followers About

STOCHASTIC TIME-SERIES MODELING

How to Model & Predict Future Time-Steps Using ARIMA Statistical Model

Time-Series modeling using Natural Gas data

Sarit Maitra Sep 18, 2020 · 7 min read ★

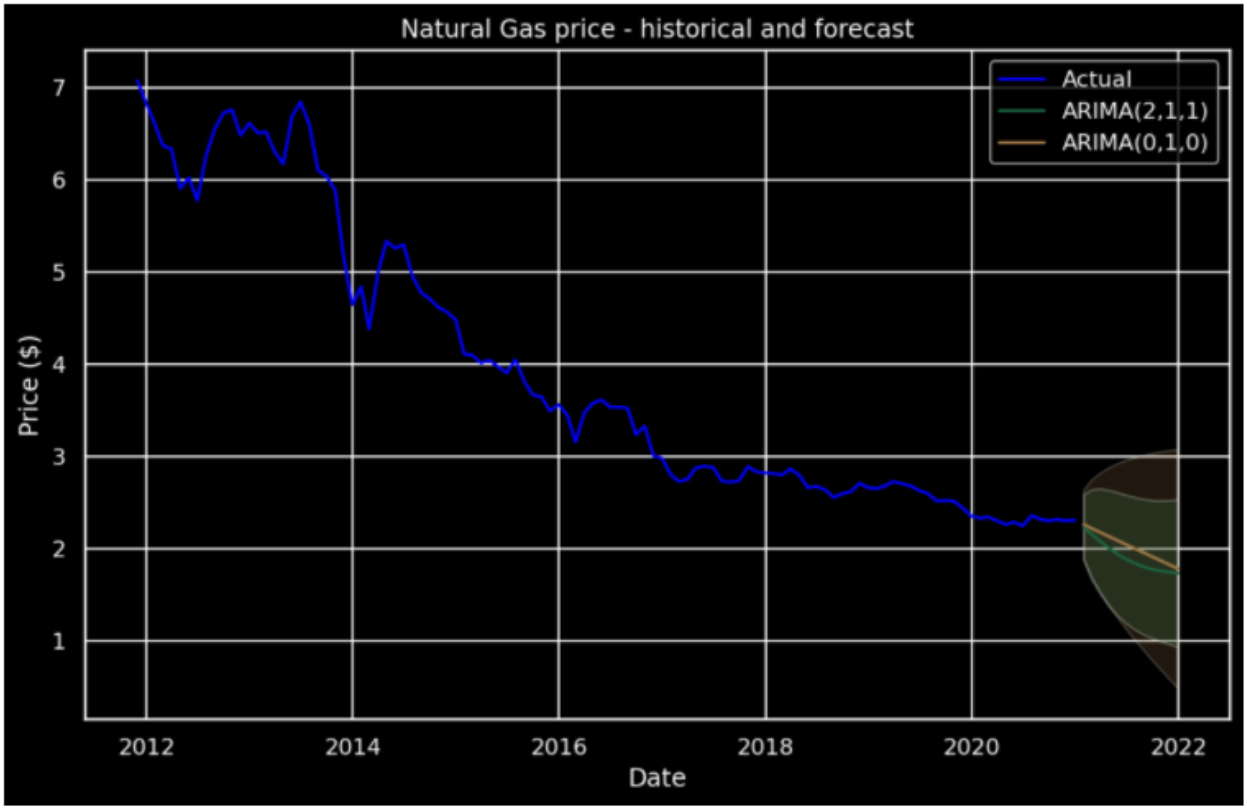


Image by author

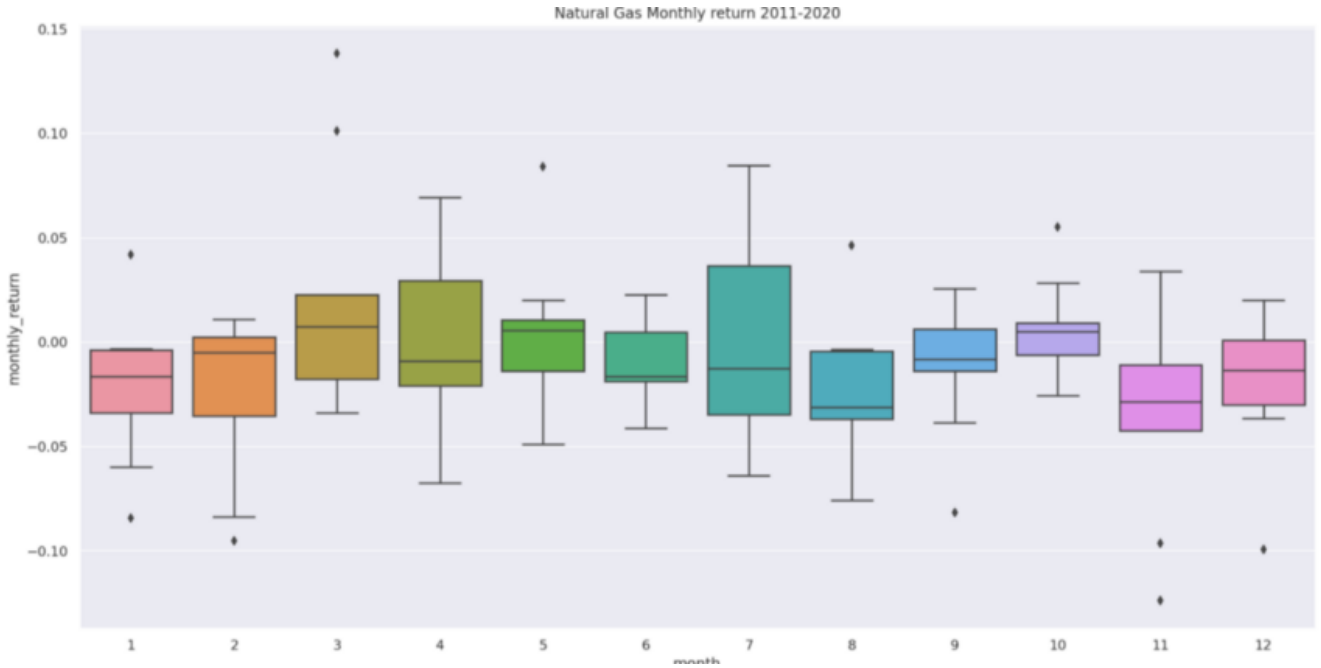
case here is to forecast future time steps using the univariate data. The time series is stochastic/ random walk price series. Here, we will discuss basic time series analysis and concepts of stationary or non-stationary time series, and how we can model financial data displaying such behavior.

We will introduce and implement advanced mathematical approaches Autoregressive (AR), Moving Average (MA), Differentiation (D), AutoCorrelation Function (ACF), and Partial Autocorrelation Function (PACF) for dealing with non-stationary time series datasets. We also will introduce seasonality concept in time-series. Let us load the check the data we have and resample to monthly frequency for the ease of computation.

```
print("....Data Loading...."); print();
print('\033[4mHenry Hub Natural Gas Price\033[0m');
data = web.DataReader('NNJ24.NYM', data_source = 'yahoo', start =
'2000-01-01')
data = data.sort_index(ascending=True)
print(data.head())
df = data.resample('M').last()
print(df.tail())
```

```
....Data Loading....

Henry Hub Natural Gas Price
             High    Low   Open  Close  Volume  Adj Close
Date
2011-11-29  7.089  7.089  7.089  7.089     0.0      7.089
2011-11-30  7.074  7.074  7.074  7.074     0.0      7.074
2011-12-01  7.059  7.059  7.059  7.059     0.0      7.059
2011-12-02  7.060  7.060  7.060  7.060     0.0      7.060
2011-12-05  7.060  7.060  7.060  7.060     0.0      7.060
             High    Low   Open  Close  Volume  Adj Close
Date
2020-08-31  2.317  2.317  2.317  2.317     0.0      2.317
2020-09-30  2.298  2.298  2.298  2.298     0.0      2.298
2020-10-31  2.314  2.314  2.314  2.314    16.0      2.314
2020-11-30  2.300  2.300  2.300  2.300     0.0      2.300
2020-12-31  2.306  2.306  2.306  2.306     0.0      2.306
```

```
df['monthly_return'] = df['price'].pct_change()
df['month'] = df.index.month
sns.set(rc={'figure.figsize':(18, 9)})
sns.boxplot(data=df, x='month', y='monthly_return')
plt.title("Natural Gas Monthly return 2011-2020")
plt.suptitle("")
plt.show()
```



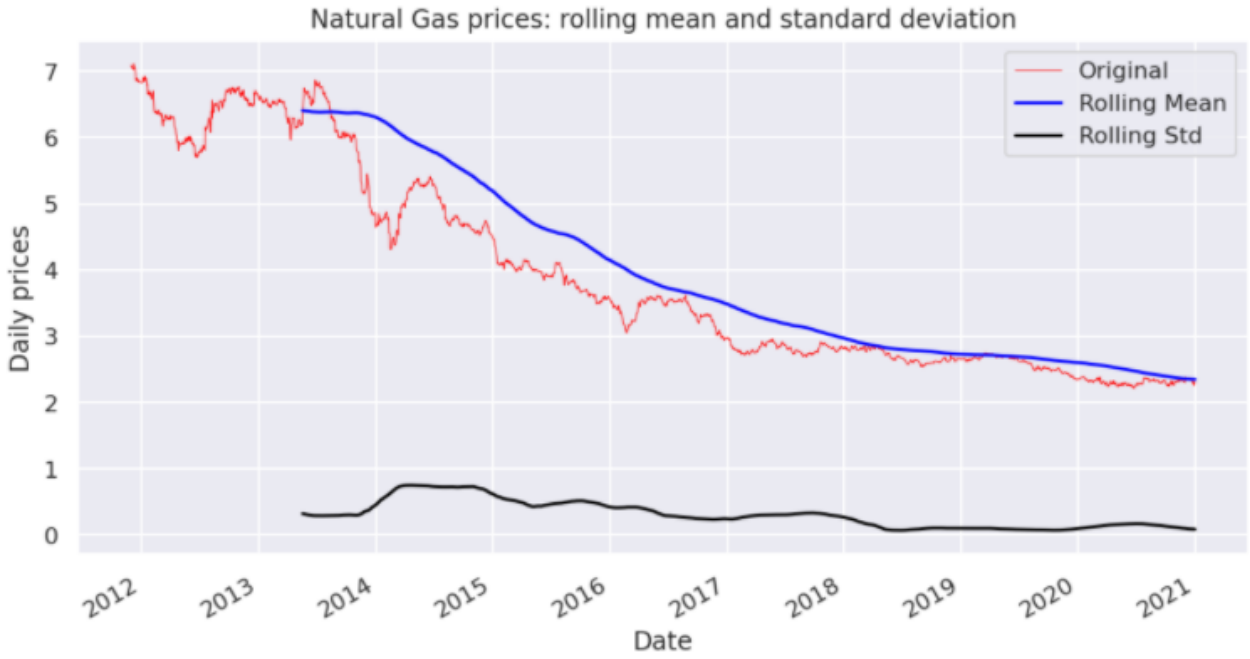
We can see from above plot that, July (7 month) seems to be the highest return, unlike August, where we see a drop in the return.

We are dealing with time-series and thus need to study the stationary (mean, variance remain constant over time).

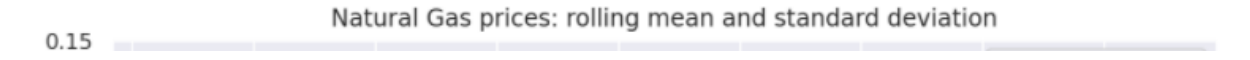
Rolling statistics:

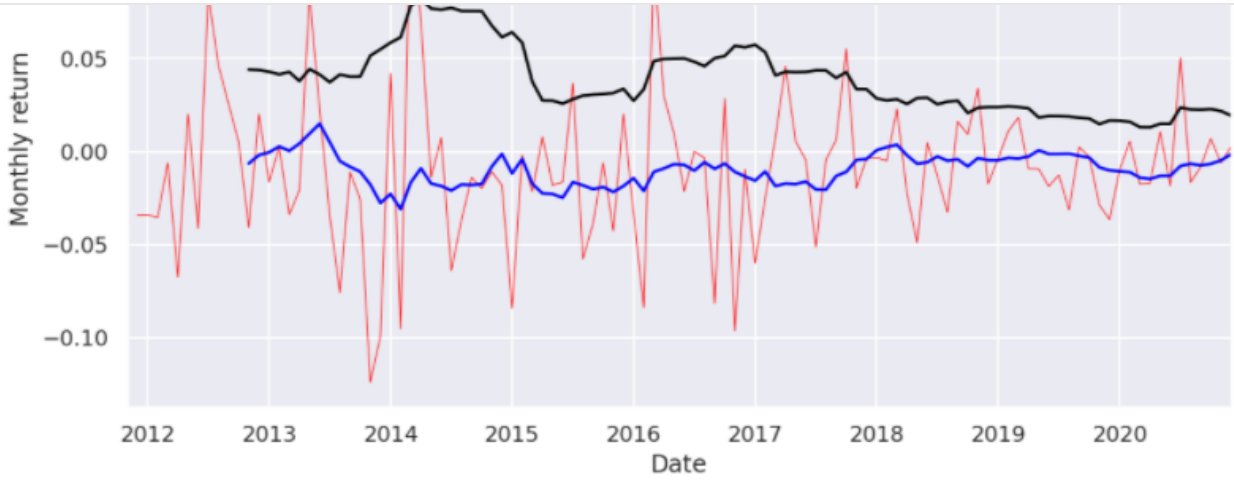
```
def plot_rolling_statistics_ts(ts, titletext,ytext, window_size=12):
    ts.plot(color='red', label='Original', lw=0.5)
    ts.rolling(window_size).mean().plot(color='blue',label='Rolling
Mean')
    ts.rolling(window_size).std().plot(color='black', label='Rolling
Std')
    plt.legend(loc='best')
    plt.ylabel(ytext)
```

```
plot_rolling_statistics_ts(df['monthly_return'],
                           'Natural Gas prices: rolling mean and
standard deviation',
                           'Monthly return')
plt.figure(figsize=(10,5))
plot_rolling_statistics_ts(data['Open'],
                           'Natural Gas prices: rolling mean and
standard deviation',
                           'Daily prices', 365)
```



Series with daily prices is not stationary because we can see the rolling mean and variance are not constant. Non-linear pattern can be observed in the 12-month moving average and that the rolling standard deviation which are on decreasing trend. Here, the general assumption of stationary time-series is violated and we will need to make this time series stationary. However, non-stationary for a time series can generally be attributed to two factors: trend and seasonality. We need to remove the trend and seasonality by modeling and removing them from the initial data. Once we find a model predicting future values for the data without seasonality and trend, we can apply back the seasonality and trend values to get the actual forecasted data.





Here, we can see that, the trend has disappeared.

We will use multiplicative model. Multiplicative model assumes that as the data increase, so does the seasonal pattern. Most time series plots exhibit such a pattern. In this model, the trend and seasonal components are multiplied and then added to the error component.

Decomposition

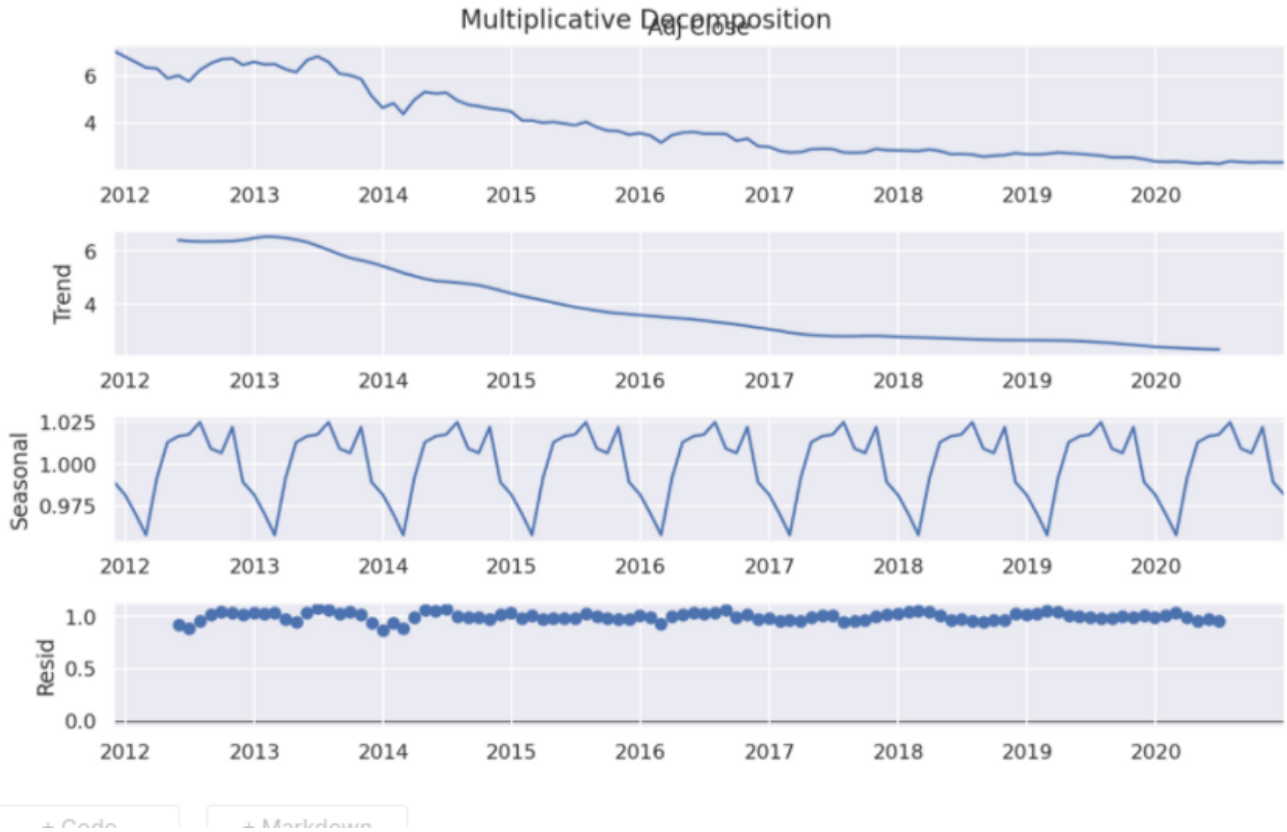
Decomposition will be performed by breaking down the series into multiple components. Objective is to have deeper understanding of the series. It provides insight in terms of modeling complexity and which approaches to follow in order to accurately capture each of the components.

The components are:

1. level which is the mean value in the series.
2. trend is associated with the slope (increasing/decreasing) of the series.
3. seasonality which is the deviations from the mean caused by repeating short-term cycles and
4. noise is the random variation in the series

want to work with the multiplicative model, we can apply transformations e.g. log transformation to make the trend/seasonality linear.

```
decomp = seasonal_decompose(df['Adj Close'], model='multiplicative')
rcParams['figure.figsize'] = 10, 6
decomp.plot().suptitle('Multiplicative Decomposition', fontsize=14);
```



It looks like the variance in the residuals is slightly higher in the 1st half of the data set. In case of additive model, the residuals display an increasing pattern over time.

Stationarity test:

To confirm our observation, let us use statistical test:

- The Augmented Dickey-Fuller (ADF) test
- The Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test

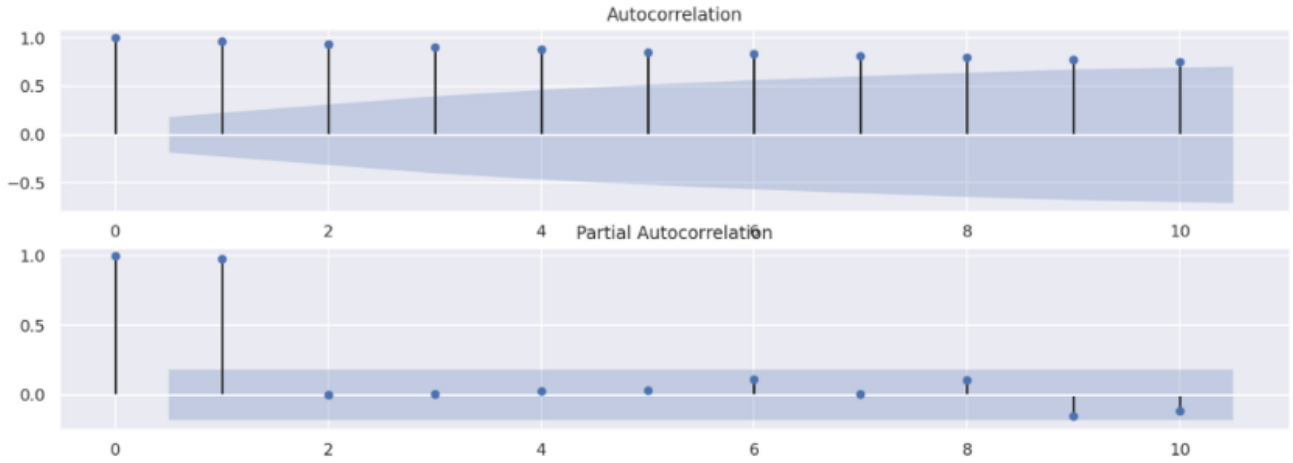
```
def test_stationarity(timeseries):
    print('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries[1:], autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic',
'pvalue',
                                             '#Lags Used', 'Number of
Observations Used'])
    print (dfoutput)

print(test_stationarity(df['Adj Close'])); print()
print(test_stationarity(df['monthly_return']))
```

```
Results of Dickey-Fuller Test:
Test Statistic                  -1.678498
pvalue                           0.442224
#Lags Used                       0.000000
Number of Observations Used    108.000000
dtype: float64
None

Results of Dickey-Fuller Test:
Test Statistic                -1.045826e+01
pvalue                         1.380415e-18
#Lags Used                     0.000000e+00
Number of Observations Used    1.080000e+02
dtype: float64
None
```

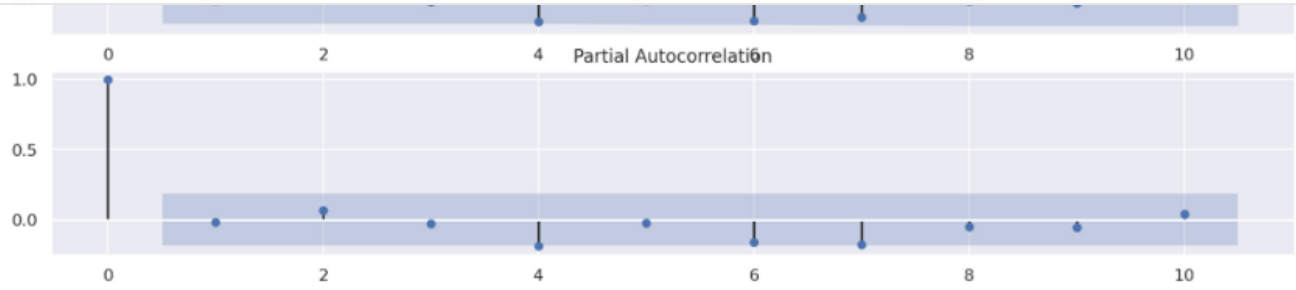
```
def kpss_test(x, h0_type='c'):
    indices = ['Test Statistic', 'p-value', '# of Lags']
    kpss_test = kpss(x, regression=h0_type)
    results = pd.Series(kpss_test[0:3], index=indices)
    for key, value in kpss_test[3].items():
        results[f'Critical Value ({key})'] = value
        return results

print(kpss_test(df['Adj Close'])); print()
print(kpss_test(df.monthly_return).dropna())
```

```
# of Lags                  13.000000
Critical Value (10%)        0.347000
dtype: float64

# of Lags                  13.000
Critical Value (10%)        0.347
dtype: float64
```

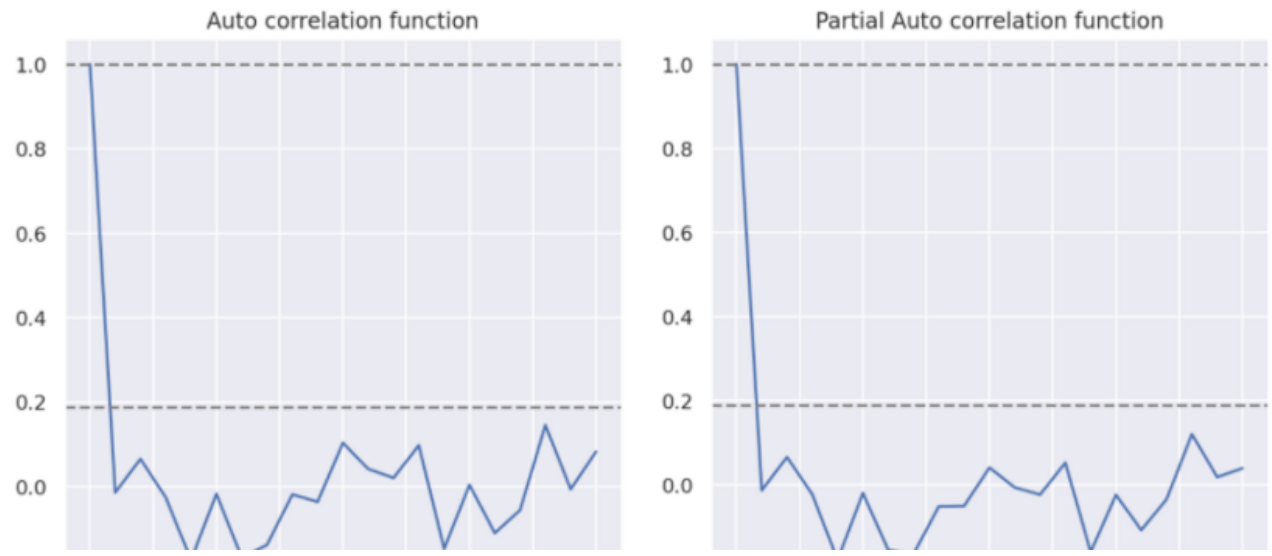
```
plt.figure(figsize=(15,5))
plt.subplot(211)
plot_acf(df['Adj Close'], ax=plt.gca(),lags=10)
plt.subplot(212)
plot_pacf(df['Adj Close'], ax=plt.gca(),lags=10)
plt.show()
```




```
plt.figure(figsize=(15,5))
plt.subplot(211)
plot_acf(df['monthly_return'].dropna(),
         ax=plt.gca(),lags=10)
plt.subplot(212)
plot_pacf(df['monthly_return'].dropna(),
          ax=plt.gca(),lags=10)
plt.show()
```




```
ret = df.monthly_return.dropna()

lag_acf = acf(ret, nlags=20)
lag_pacf = pacf(ret, nlags=20, method = 'ols')

# plot acf
plt.subplot(121); plt.plot(lag_acf);
plt.axhline(y=1, linestyle='--', color = 'gray')
plt.axhline(y=-1.96/np.sqrt(len(ret)), linestyle='--', color =
'gray')
plt.axhline(y=1.96/np.sqrt(len(ret)), linestyle='--', color = 'gray')
plt.title('Auto correlation function')

# plot pacf
plt.subplot(122); plt.plot(lag_pacf);
plt.axhline(y=1, linestyle='--', color = 'gray')
plt.axhline(y=-1.96/np.sqrt(len(ret)), linestyle='--', color =
'gray')
plt.axhline(y=1.96/np.sqrt(len(ret)), linestyle='--', color = 'gray')
plt.title('Partial Auto correlation function'); plt.tight_layout();
```



Finally in order to forecast we shall use the Auto- Regression Integrated Moving Averages (ARIMA) model. This model has three parameters: (1) Autoregressive (AR) term (p) — lags of dependent variables, (2) Moving average (MA) term (q) — lags for errors in prediction and (3) Differentiation (d) — This is the d number of occasions where we apply differentiation between values.

ARIMA model

```
arima = ARIMA(df['Adj Close'], order=(2, 1, 2)).fit(disp=0)
arima.summary()
```

ARIMA Model Results

Dep. Variable:	D.Adj Close	**No. Observations:**	109
Model:	ARIMA(2, 1, 2)	**Log Likelihood**	29.190
Method:	css-mle	**S.D. of innovations**	0.181
Date:	Sat, 02 Jan 2021	**AIC**	-46.380
Time:	20:17:45	**BIC**	-30.232
Sample:	12-31-2011	**HQIC**	-39.832
	- 12-31-2020		

	coef	std err	z	P>|z|	[0.025	0.975]
const	-0.0403	0.012	-3.296	0.001	-0.064	-0.016
ar.L1.D.Adj Close	1.8641	0.033	56.847	0.000	1.800	1.928
ar.L2.D.Adj Close	-0.9562	0.034	-28.152	0.000	-1.023	-0.890
ma.L1.D.Adj Close	-1.9356	0.078	-24.928	0.000	-2.088	-1.783
ma.L2.D.Adj Close	1.0000	0.080	12.529	0.000	0.844	1.156

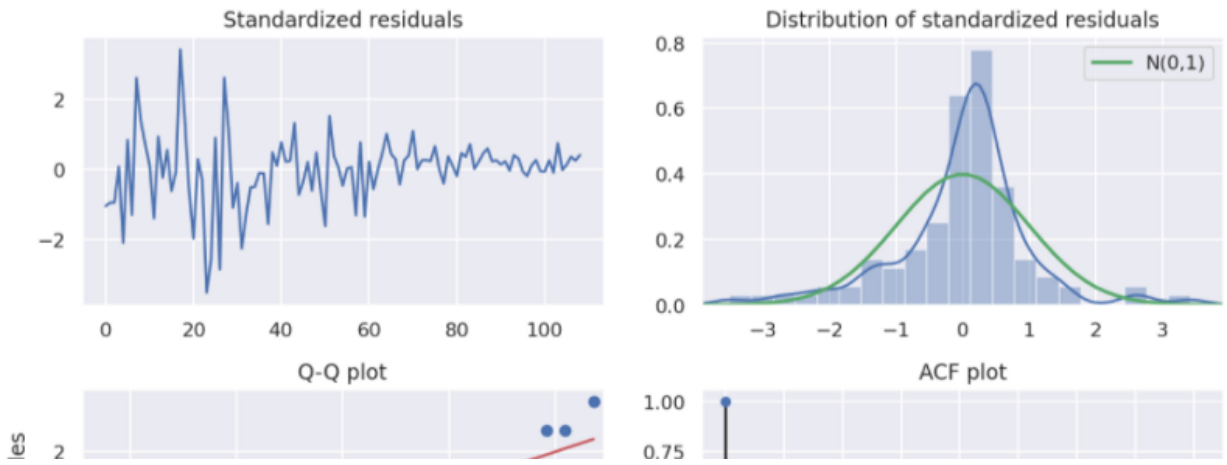
Roots

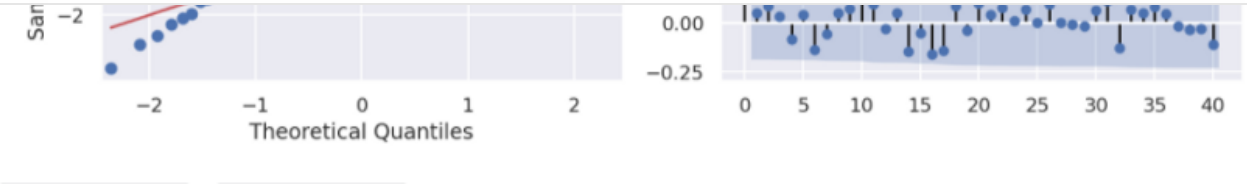
	Real	Imaginary	Modulus	Frequency
AR.1	0.9748	-0.3092j	1.0227	-0.0489

MA.2	0.9678	+0.2517j	1.0000	0.0405

The AR & MA (p & q) orders taken based on acf/pacf plots as shown above.

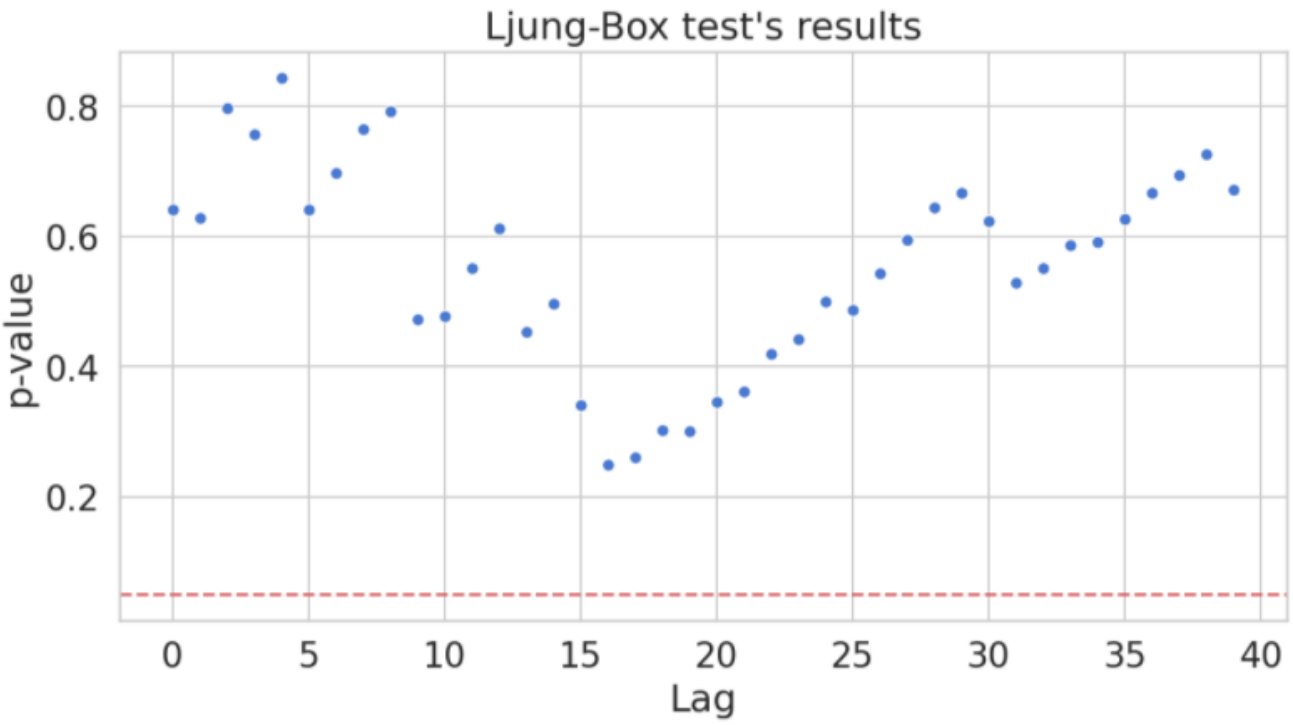
```
def arima_diagnostics(resids, n_lags=40):
  fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2);
  r = resids;  resids = (r - np.nanmean(r)) / np.nanstd(r);
  resids_nonmissing = resids[~(np.isnan(resids))];

  sns.lineplot(x=np.arange(len(resids)), y=resids, ax=ax1);
  ax1.set_title('Standardized residuals');
  x_lim = (-1.96 * 2, 1.96 * 2);  r_range = np.linspace(x_lim[0],
x_lim[1]);  norm_pdf = scs.norm.pdf(r_range);
  sns.distplot(resids_nonmissing, hist=True, kde=True,   norm_hist
=True, ax=ax2);
  ax2.plot(r_range, norm_pdf, 'g', lw=2, label='N(0,1)');
  ax2.set_title('Distribution of standardized residuals');
  ax2.set_xlim(x_lim); ax2.legend();

  # Q-Q plot
  qq = sm.qqplot(resids_nonmissing, line='s', ax=ax3);
  ax3.set_title('Q-Q plot');

  # ACF plot
  plot_acf(resids, ax=ax4, lags=n_lags, alpha=0.05);
  ax4.set_title('ACF plot');
  return fig

arima_diagnostics(arima.resid, 40); plt.tight_layout();
```





Looking at the diagnostics plots, the residuals look like normal distribution. The average of the residuals is close to 0 (-0.05), and ACF plot says that the residuals are not correlated.Ljung-Box test

```
ljung_box_results = acorr_ljungbox(arima.resid)
fig, ax = plt.subplots(1, figsize=[10, 5])
sns.scatterplot(x=range(len(ljung_box_results[1])),
y=ljung_box_results[1], ax=ax)
ax.axhline(0.05, ls='--', c='r')
ax.set(title="Ljung-Box test's results", xlabel='Lag', ylabel='p-
value')
plt.show()
```



Ljung-Box test satisfies the goodness-of-fit by showing no significant autocorrelation for any of the selected lags.

```
forecast = int(12)
arima_pred, std, ci = (arima.forecast(steps=forecast))

arima_pred = DataFrame(arima_pred)
d = DataFrame(df['Adj Close'].tail(len(arima_pred)));
d.reset_index(inplace = True)
d = d.append(DataFrame({'Date': pd.date_range(start =
d.Date.iloc[-1],
                                              periods = (len(d)+1),
freq = 'm', closed = 'right')}))
d = d.tail(forecast); d.set_index('Date', inplace = True)
arima_pred.index = d.index
arima_pred.rename(columns = {0: 'arima_fcast'}, inplace=True)

# 95% prediction interval
ci = DataFrame(ci)
ci.rename(columns = {0: 'lower95', 1:'upper95'}, inplace=True)
ci.index = arima_pred.index

ARIMA = concat([arima_pred, ci], axis=1)
ARIMA
```

Date	arima_fcast	lower95	upper95
2021-01-31	2.231399	1.875835	2.586963
2021-02-28	2.154214	1.669010	2.639419
2021-03-31	2.077957	1.508365	2.647550
2021-04-30	2.005900	1.378365	2.633435
2021-05-31	1.940785	1.272916	2.608653
2021-06-30	1.884594	1.188233	2.580956
2021-07-31	1.838404	1.121149	2.555660
2021-08-31	1.802321	1.068563	2.536079
2021-09-30	1.775518	1.027176	2.523860
2021-10-31	1.756348	0.993318	2.519378
2021-11-30	1.742537	0.962899	2.522174
2021-12-31	1.731414	0.931537	2.531291

Auto-ARIMA

To re-validate the manual selection of ARIMA parameters, let us run through auto-arima.

```
model = pm.auto_arima(df['Adj Close'], error_action='ignore',
suppress_warnings=True,
                      seasonal=False)
model.summary()
```

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	110
Model:	SARIMAX(0, 1, 0)	**Log Likelihood**	25.931
Date:	Sat, 02 Jan 2021	**AIC**	-47.862
Time:	20:23:34	**BIC**	-42.480
Sample:	0	**HQIC**	-45.679
	- 110		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	-0.0437	0.018	-2.394	0.017	-0.080	-0.008
sigma2	0.0364	0.003	11.411	0.000	0.030	0.043

Ljung-Box (Q):	42.54	**Jarque-Bera (JB):**	35.07
Prob(Q):	0.36	**Prob(JB):**	0.00
Heteroskedasticity (H):	0.04	**Skew:**	-0.01
Prob(H) (two-sided):	0.00	**Kurtosis:**	5.78

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

Prediction (auto-arima)

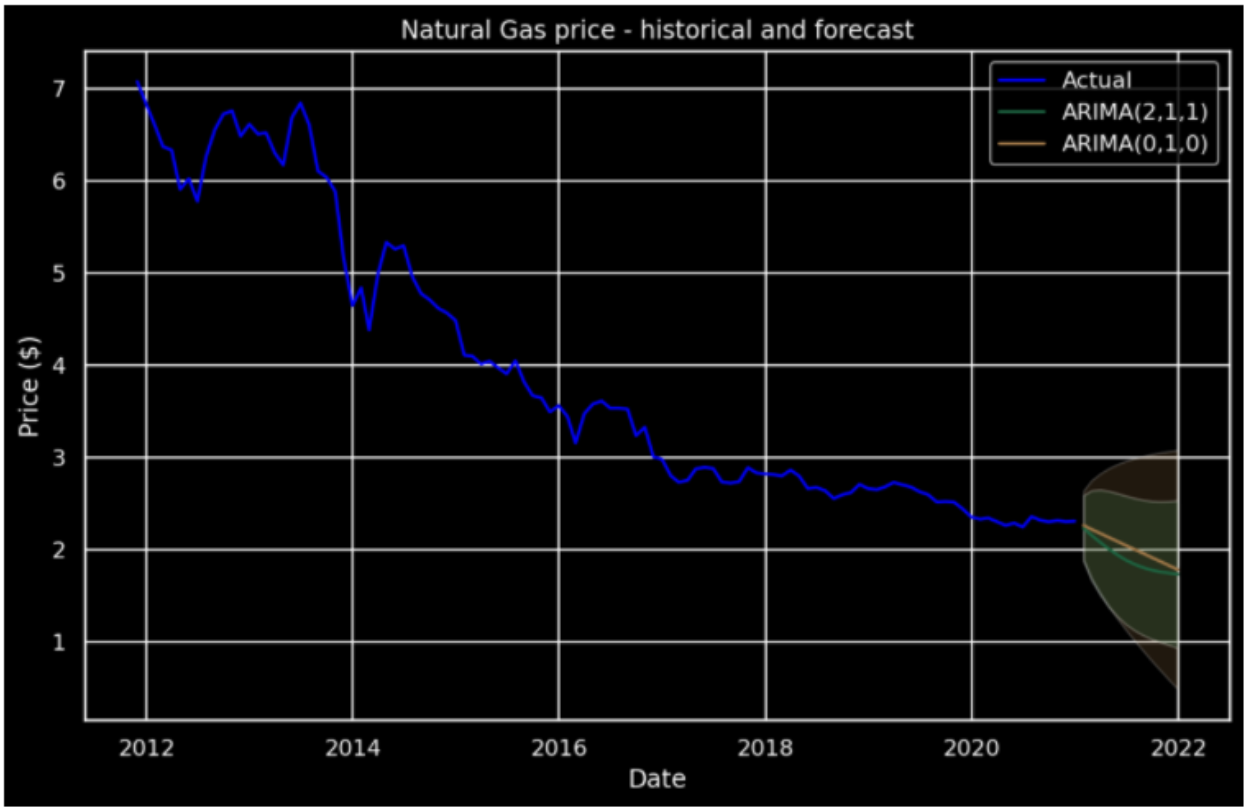
```
auto_arima_pred = [DataFrame(auto_arima_pred[0],
                             columns=['prediction']),
                   DataFrame(auto_arima_pred[1],
                             columns=['ci_lower', 'ci_upper'])]
auto_arima_pred =
concat(auto_arima_pred,axis=1).set_index(ARIMA.index)
auto_arima_pred
```

Date	prediction	ci_lower	ci_upper
2021-01-31	2.262257	1.888412	2.636102
2021-02-28	2.218514	1.689817	2.747211
2021-03-31	2.174771	1.527251	2.822290
2021-04-30	2.131027	1.383337	2.878718
2021-05-31	2.087284	1.251341	2.923228
2021-06-30	2.043541	1.127811	2.959272
2021-07-31	1.999798	1.010696	2.988900
2021-08-31	1.956055	0.898661	3.013449
2021-09-30	1.912312	0.790776	3.033848
2021-10-31	1.868569	0.686366	3.050771
2021-11-30	1.824826	0.584921	3.064730
2021-12-31	1.781082	0.486044	3.076121

Visualization

```
plt.set_cmap('cubehelix'); sns.set_palette('cubehelix')
COLORS = [plt.cm.cubehelix(x) for x in [0.1, 0.3, 0.5, 0.7]]; fig, ax
= plt.subplots(1)
ax = sns.lineplot(data=df['Adj Close'], color=COLORS[0],
label='Actual')
ax.plot(ARIMA.arima_fcast, c=COLORS[1], label='ARIMA(2,1,1)')
ax.fill_between(ARIMA.index, ARIMA.lower95, ARIMA.upper95, alpha=0.3,
facecolor=COLORS[1])
ax.plot(auto_arima_pred.prediction, c=COLORS[2],
```

```
                alpha=0.2, facecolor=COLORS[2])
ax.set(title="Natural Gas price - historical and forecast",
xlabel='Date', ylabel='Price ($)')
ax.legend(loc='best')
```



Conclusion

The approach applied here to forecast the future trends of price movements based on its past behavior using stochastic time-series modeling . ARIMA model is designated by ARIMA (p, d, q), where ‘p’ is the auto regressive process order, ‘d’ corresponds to stationary data order and ‘q’ denotes the moving average process order. Validity of the developed ARIMA models can be testified via statistical parameters such as RMSE, MAPE, AIC, AICc and BIC which was not shown here. ARIMA with GARCH volatility model can be tried too to capture the volatility in the series.

Connect me here.

Timeseries Forecasting | Arima | Predictive Modeling | Stochastic Modelling