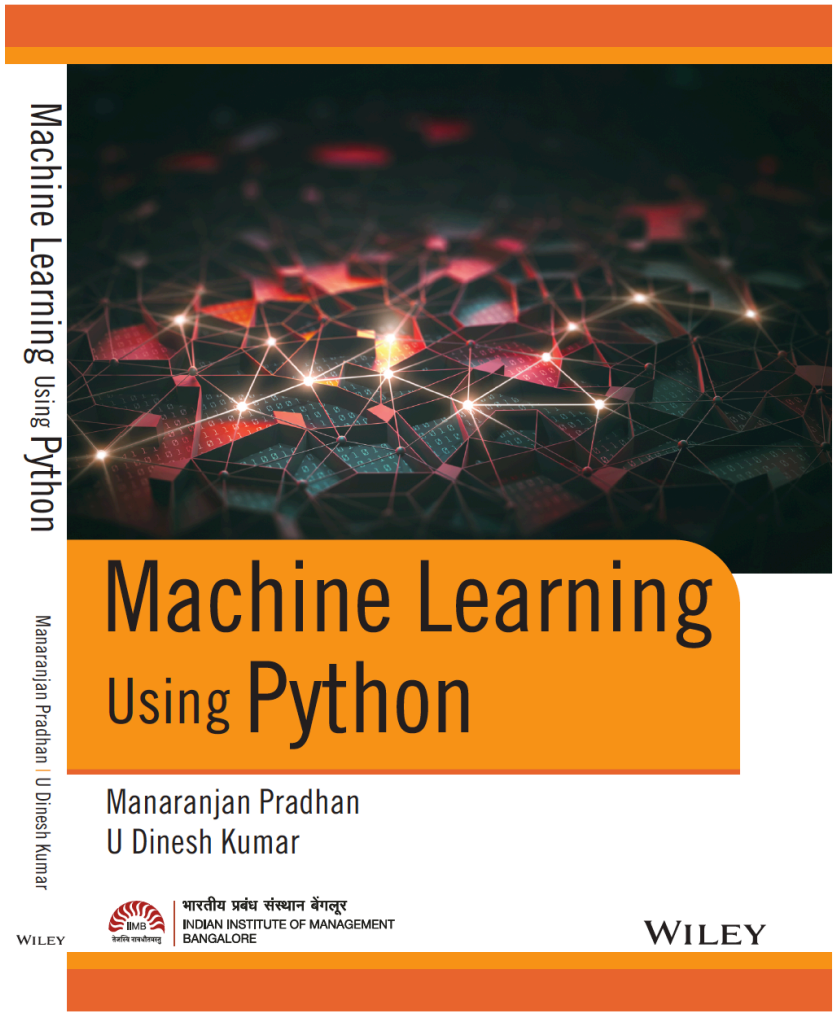


Chapter 08: Forecasting

Learning Objectives

- Importance of time-series data, forecasting techniques and their impact on business
- Learning various components of time-series data such as trend, seasonality, cyclical component and random component
- Learning Auto-Regressive(AR), Moving Average(MA) and Auto-Regressive Integrated Moving Average(ARIMA) Models
- Building forecasting models using *statsmodel* API in python

Introduction to Forecasting

Forecasting is one of the most important and frequently addressed problems in analytics. Inaccurate forecasting can have significant impact on both top line and bottom line of an organization. For example, non-availability of product in the market can result in customer dissatisfaction, whereas, too much inventory can erode the organization's profit. Thus, it becomes necessary to forecast the demand for a product and service as accurately as possible.

Components of Time-Series Data

- Time-series data is a data on a response variable, Y_t, such as demand for a spare parts of a capital equipment or a product or a service or market share of a brand observed at different time points t.
- The variable Y_t is a random variable.
- If the time series data contains observations of just a single variable (such as demand of a product at time t), then it is termed as univariate time series
- A time-series data can be broken into four components.

Trend Component(T_t)

Trend is the consistent long-term upward or downward movement of the data.

Seasonal Component(S_t)

Seasonal component (or seasonality index) is the repetitive upward or downward movement (or fluctuations) from the trend that occurs within a calendar year such as seasons, quarters, months, days of the week, etc.

Cyclical Component(C_t)

- Cyclical component is fluctuation around the trend line that happens due to macro-economic changes such as recession, unemployment, etc.
- Cyclical fluctuations have repetitive patterns with a time between repetitions of more than a year.

Irregular Component(I_t)

Irregular component is the white noise or random uncorrelated changes that follow a normal distribution with mean value of 0 and constant variance.

Loading and Visualizing Time-Series Dataset

```
import pandas as pd

wsb_df = pd.read_csv('wsb.csv')
wsb_df.head(10)
```

	Month	Sale Quantity	Promotion Expenses	Competition Promotion
0	1	3002666	105	1
1	2	4401553	145	0
2	3	3205279	118	1
3	4	4245349	130	0
4	5	3001940	98	1
5	6	4377766	156	0
6	7	2798343	98	1
7	8	4303668	144	0
8	9	2958185	112	1
9	10	3623386	120	0

Time-Series Plot

To visualize *Sale Quantity* over *Month using plot() method* from Matplotlib use the following code:

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline
```

```
plt.figure(figsize=(10,4))
plt.xlabel("Months")
plt.ylabel("Quantity")
plt.plot(wsb_df['Sale Quantity']);
```

Continued….

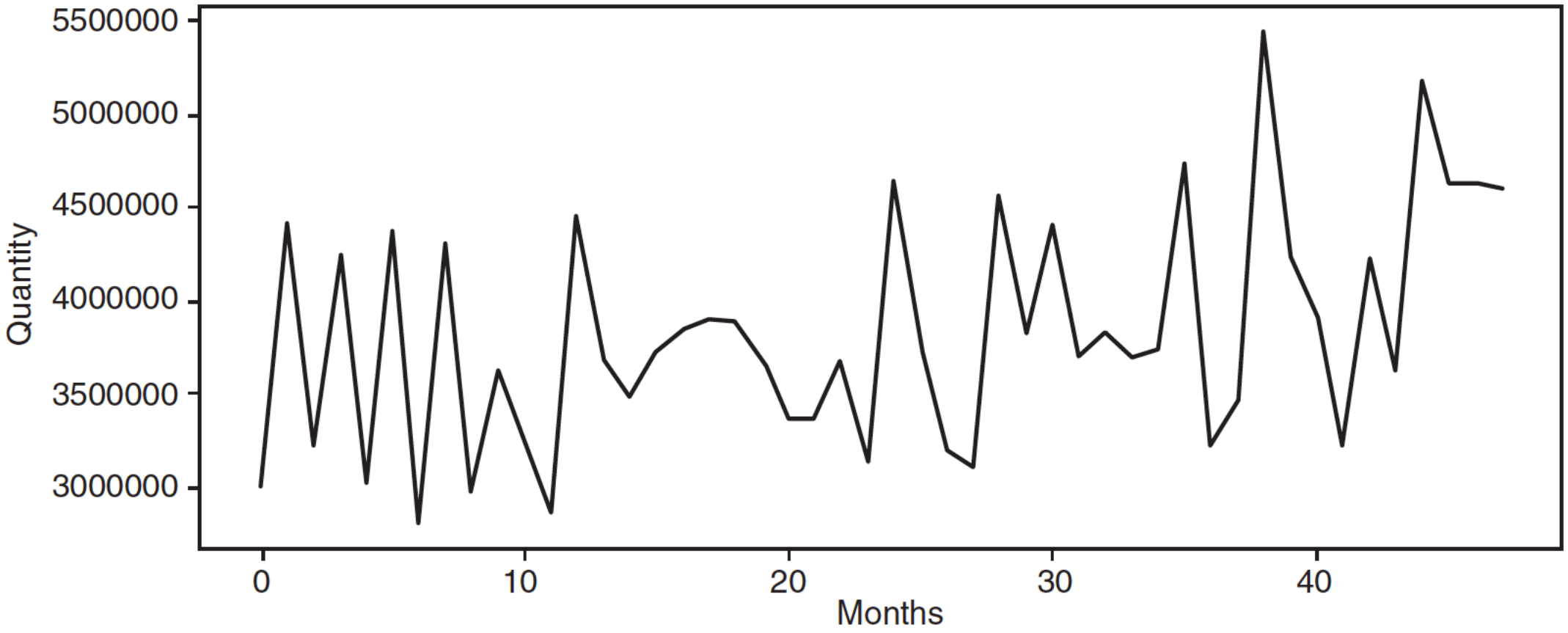


FIGURE 8.1 Sales quantity over 48 months.

Summary of the Dataset

```
wsb_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48 entries, 0 to 47
Data columns (total 4 columns):
Month                    48 non-null int64
Sale Quantity            48 non-null int64
Promotion Expenses       48 non-null int64
Competition Promotion    48 non-null int64
dtypes: int64(4)
memory usage: 1.6 KB
```

Moving Average

Moving average is the simplest forecasting technique which forecasts the future value of a time-series data using average (or weighted average) of the past N observations. Forecasted value for time t+1 using the simple moving average is given by

$$F_{t+1} = \frac{1}{N} \sum_{k=t+1-N}^{t} Y_k$$

Forecasting using Moving Average

```
wsb_df['mavg_12'] = wsb_df['Sale Quantity'].rolling(window = 12).
mean().shift(1)
```

To display values up to 2 decimal points, we can use *pd.set_option* and floating format %.2f.

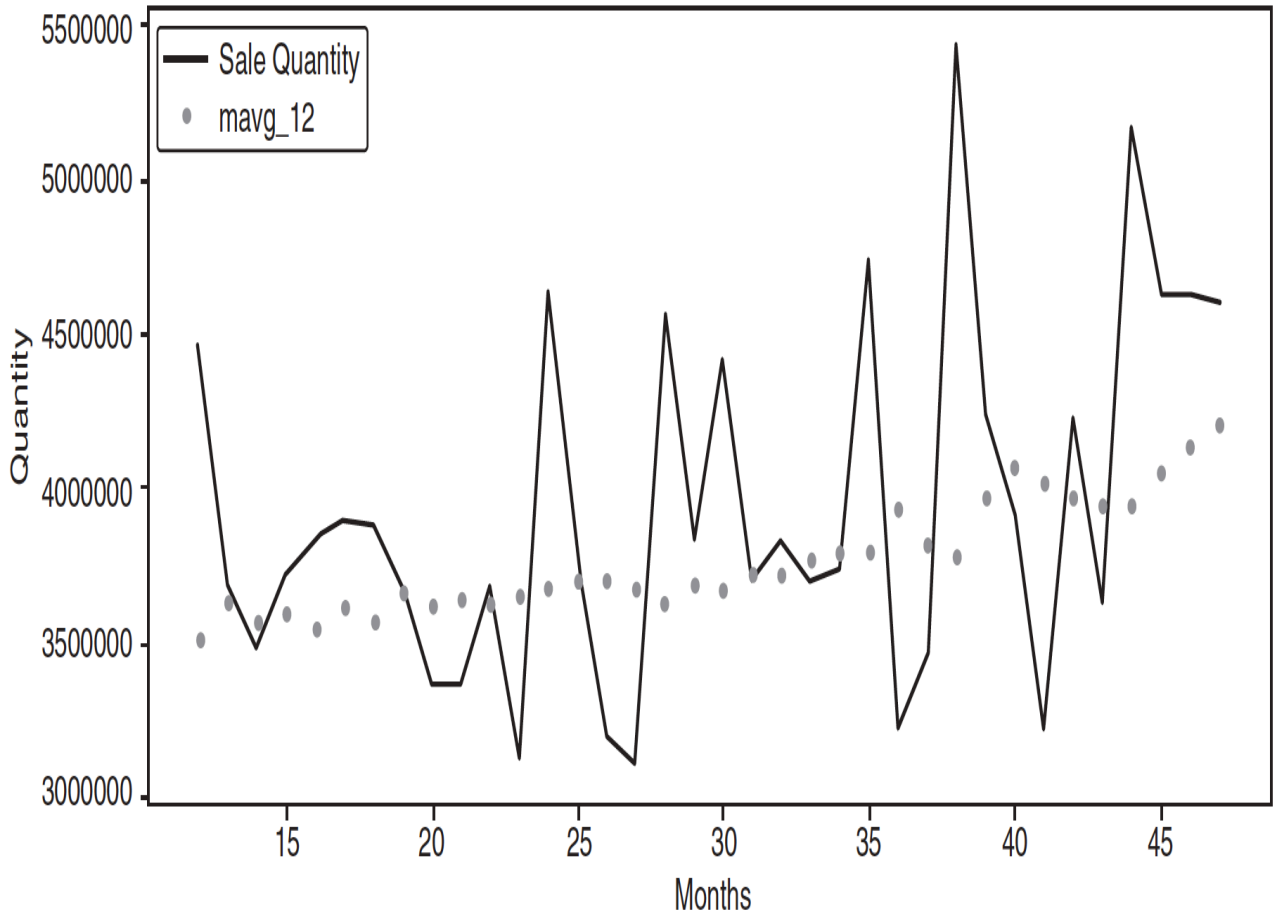
```
pd.set_option('display.float_format', lambda x: '%.2f' % x)
wsb_df[['Sale Quantity', 'mavg_12']][36:]
```

Forecasted Values

	Sale Quantity	mavg_12
36	3216483	3928410.33
37	3453239	3810280.00
38	5431651	3783643.33
39	4241851	3970688.42
40	3909887	4066369.08
41	3216438	4012412.75
42	4222005	3962369.58
43	3621034	3946629.42
44	5162201	3940489.50
45	4627177	4052117.17
46	4623945	4130274.75
47	4599368	4204882.33

Actual vs Predicted

```
plt.figure(figsize=(10,4))
plt.xlabel("Months")
plt.ylabel("Quantity")
plt.plot(wsb_df['Sale Quantity'][12:]);
plt.plot(wsb_df['mavg_12'][12:], '.');
plt.legend();
```



Forecast Accuracy

Root mean square error (RMSE) and mean absolute percentage error (MAPE) are the two most popular accuracy measures of forecasting.

$$\text{MAPE} = \frac{1}{n}\sum_{t=1}^{n}\frac{|Y_t - F_t|}{Y_t} \times 100\%$$

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(Y_t - F_t)^2}$$

Mean Absolute Percentage Error

```
import numpy as np

def get_mape(actual, predicted):
    y_true, y_pred = np.array(actual), np.array(predicted)
    return np.round( np.mean(np.abs((actual - predicted) / actual))
    * 100, 2 )
```

```
get_mape( wsb_df['Sale Quantity'][36:].values,
          wsb_df['mavg_12'][36:].values)
```

The MAPE in this case is 14.04. So, forecasting using moving average gives us a MAPE of 14.04%.

Root Mean Square Error

```
from sklearn.metrics import mean_squared_error

np.sqrt(mean_squared_error(wsb_df['Sale Quantity'][36:].values,
                           wsb_df['mavg_12'][36:].values))
```

The RMSE in this case is 734725.83. So, the RMSE of the moving average model indicates that the prediction by the models has a standard deviation of 734725.83.

Exponential Smoothing

- Exponential Smoothing(also known as Single Exponential Smoothing) assigns differential weights to past observations

$$F_{t+1} = \alpha Y_t + (1-\alpha) F_t$$

- Parameter α in Eq. is called the smoothing constant and its value lies between 0 and 1.

Forecasting using SES

```
wsb_df['ewm'] = wsb_df['Sale Quantity'].ewm( alpha = 0.2 ).mean()
```

Set the floating point formatting up to 2 decimal points.

```
pd.options.display.float_format = '{:.2f}'.format
```

Use the following code to display the records from the 37th month.

```
wsb_df[36:]
```

Continued

	Month	Sale Quantity	Promotion Expenses	Competition Promotion	mavg_12	Ewm
36	37	3216483	121	1	3928410.33	3828234.64
37	38	3453239	128	0	3810280.00	3753219.93
38	39	5431651	170	0	3783643.33	4088961.93
39	40	4241851	160	0	3970688.42	4119543.81
40	41	3909887	151	1	4066369.08	4077607.99
41	42	3216438	120	1	4012412.75	3905359.34
42	43	4222005	152	0	3962369.58	3968692.78
43	44	3621034	125	0	3946629.42	3899157.24
44	45	5162201	170	0	3940489.50	4151776.99
45	46	4627177	160	0	4052117.17	4246860.31
46	47	4623945	168	0	4130274.75	4322279.35
47	48	4599368	166	0	4204882.33	4377698.31

Continued

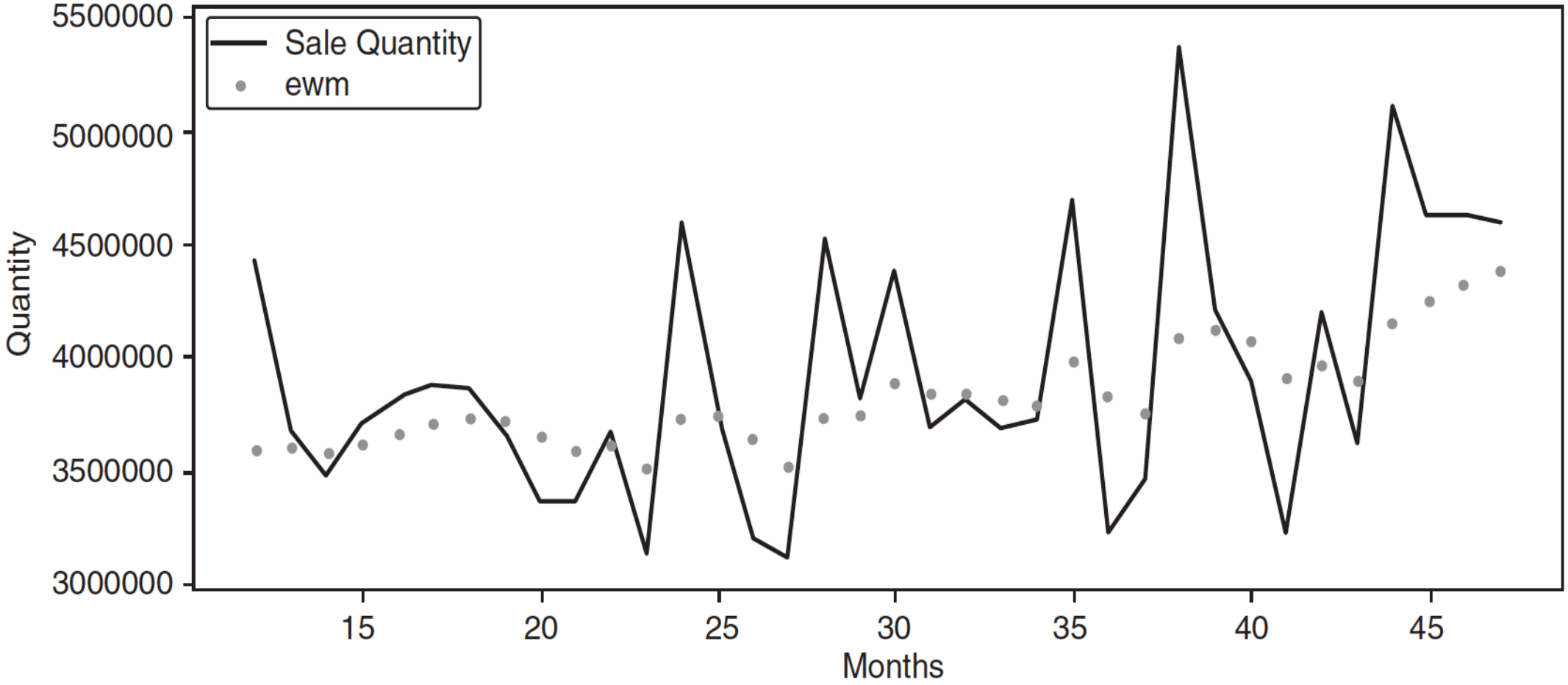
```
get_mape(wsb_df[['Sale Quantity']][36:].values,
         wsb_df[['ewm']][36:].values)
```

```
The MAPE of the model is 11.15.
```

So, forecasting using exponential smoothing has about 11.15% error (MAPE) from the actual values. It is an improvement compared to the simple moving average model. Let us plot the output to view the difference between the forecasted and actual sales quantity using exponential moving average.

```
plt.figure( figsize=(10,4))
plt.xlabel( "Months" )
plt.ylabel( "Quantity" )
plt.plot( wsb_df['Sale Quantity'][12:] );
plt.plot( wsb_df['ewm'][12:], '.' );
plt.legend();
```

Forecasted vs Actual using SES



Decomposition of Time-series

- The time-series data can be modelled as addition or product of trend, seasonality, cyclical, and irregular components.
- The additive time-series model is given by

$$Y_t = T_t + S_t + C_t + I_t$$

- The multiplicative time-series model is given by

$$Y_t = T_t \times S_t \times C_t \times I_t$$

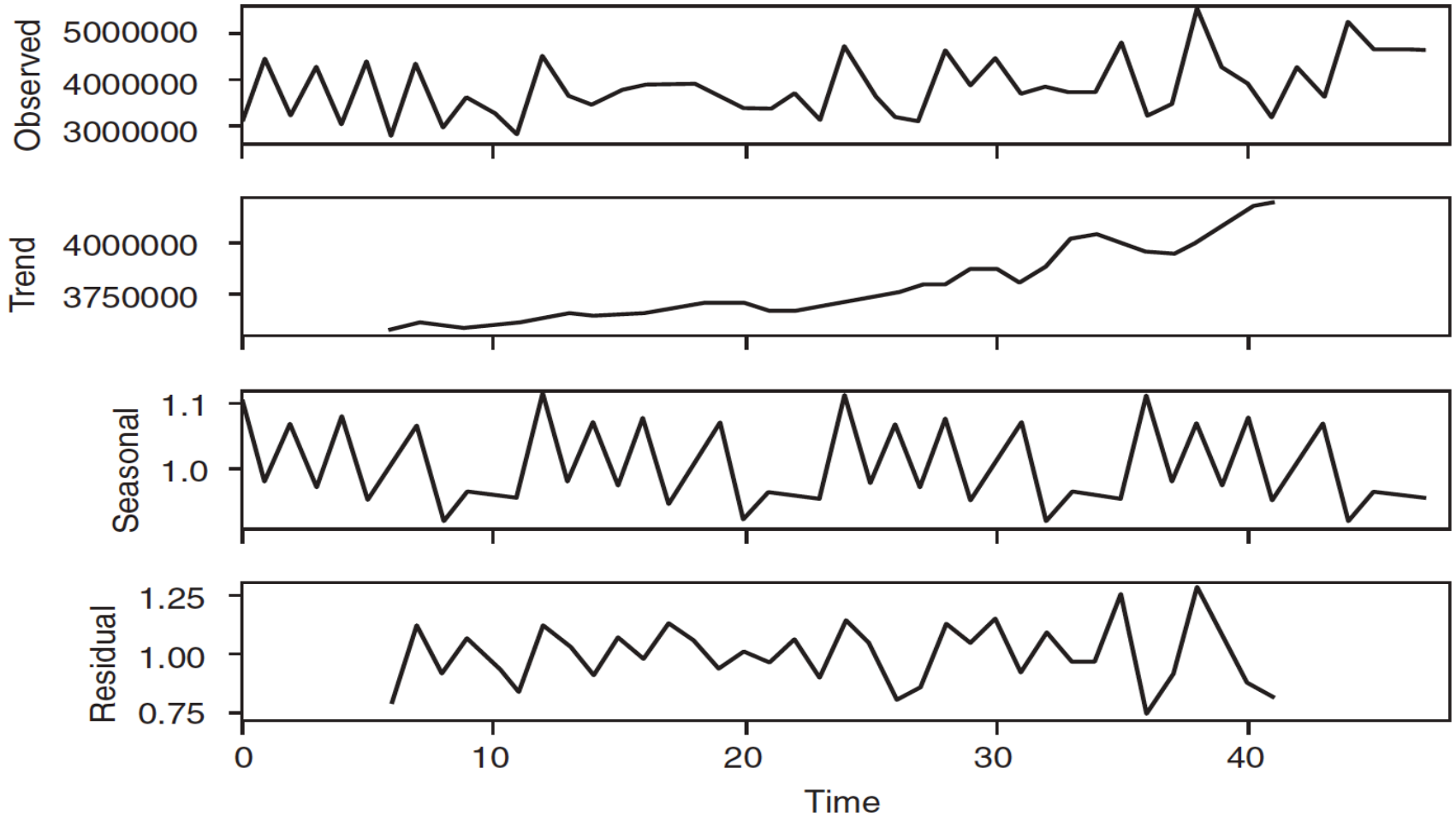
Continued

```
from statsmodels.tsa.seasonal import seasonal_decompose

ts_decompse = seasonal_decompose(np.array(wsb_df['Sale Quantity']),
                                 model='multiplicative',
                                 freq = 12)

## Plotting the deocomposed time series components
ts_plot = ts_decompose.plot()
```

To capture the seasonal and trend components after time-series decomposition we use the following code. The information can be read from two variables *seasonal* and *trend* in *ts_decompose*.

```
wsb_df['seasonal'] = ts_decompse.seasonal
wsb_df['trend'] = ts_decompse.trend
```

Plot of Decomposed Time-Series



Auto-Regressive (AR) Models

Auto-regression is regression of a variable on itself measured at different time points. Auto-regressive model with lag 1, AR(1), is given by

$$Y_{t+1} = \beta\, Y_t + \varepsilon_{t+1}$$

which can be re-written as

$$Y_{t+1} - \mu = \beta \times (Y_t - \mu) + \varepsilon_{t+1}$$

ACF

Auto-correlation of lag k is the correlation between Y_t measured at different k values (e.g., Y_t and Y_{t-1} or Y_t and Y_{t-k}). A plot of auto-correlation for different values of k is called an auto-correlation function (ACF) or correlogram.

statsmodels.graphics.tsaplots.plot_acf plots the autocorrelation plot.

PACF

Partial auto-correlation of lag k is the correlation between Y_t and Y_{t-k} when the influence of all intermediate values (Y_{t-1}, Y_{t-2}, …, Y_{t-k+1}) is removed (partial out) from both Y_t and Y_{t-k}. A plot of partial auto-correlation for different values of k is called partial auto-correlation function (PACF).

statsmodels.graphics.tsaplots.plot_pacf plots the partial auto-correlation plot.

Monthly Demand for Aircraft Spare Parts

We first read the dataset from *viman.csv* onto a DataFrame and print the first 5 records.

```
vimana_df = pd.read_csv('vimana.csv')

vimana_df.head(5)
```

	Month	Demand
0	1	457
1	2	439
2	3	404
3	4	392
4	5	403

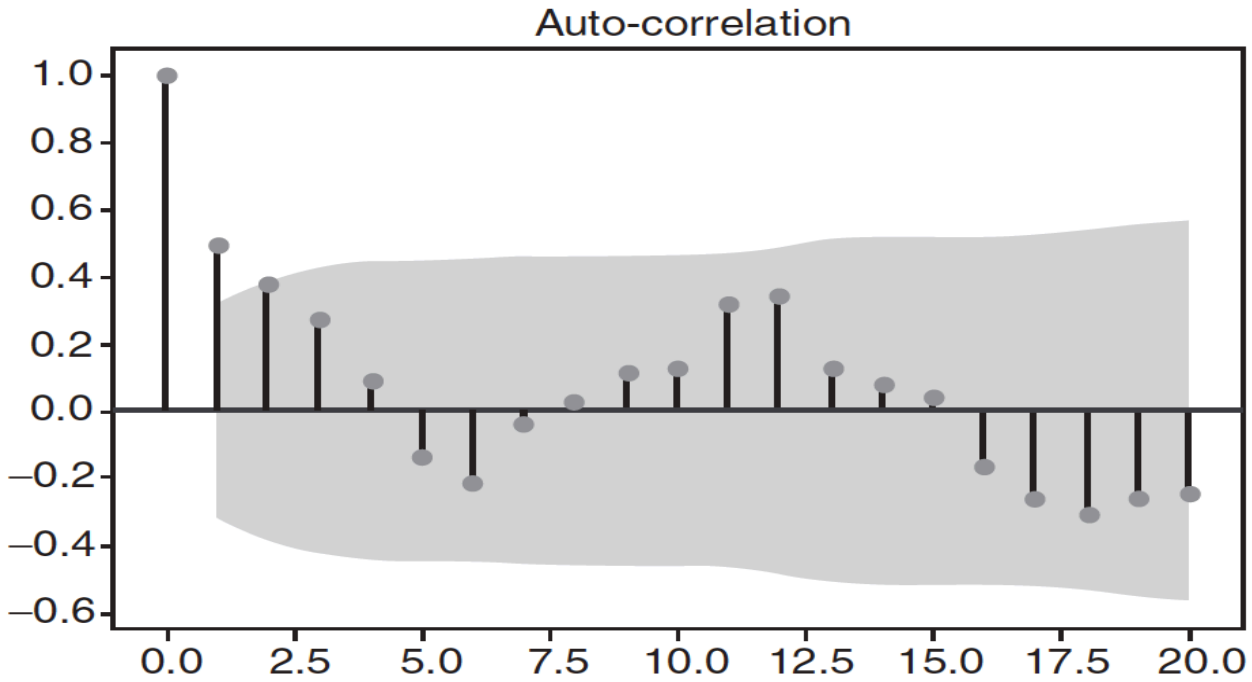
Continued….

```
vimana_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 37 entries, 0 to 36
Data columns (total 2 columns):
Month      37 non-null int64
demand     37 non-null int64
dtypes: int64(2)
memory usage: 672.0 bytes
```

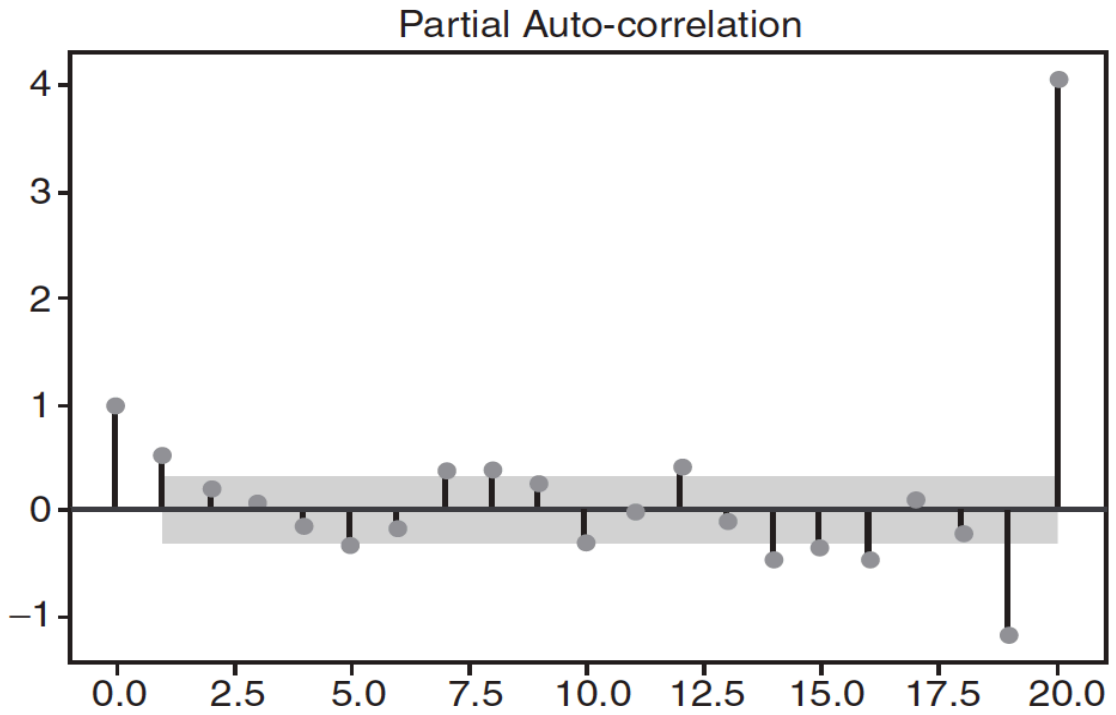
ACF Plot

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

# Show autocorrelation upto lag 20
acf_plot = plot_acf( vimana_df.demand,
                     lags=20)
```



PACF Plot

```
pacf_plot = plot_pacf( vimana_df.demand,
                       lags=20 )
```



In the above case, based on ACF and PACF plots, AR with lag 1, AR(1), can be used.

Building AR Model

The *statsmodels.tsa.arima_model.ARIMA* can be used to build AR model. It takes the following two parameters:

1. endog: list of values – It is the endogenous variable of the time series.
2. order: The (p, d, q) – ARIMA model parameters. Order of AR is given by the value *p*, the order of integration is *d*, and the order of MA is given by *q*.

```
from statsmodels.tsa.arima_model import ARIMA
```

```
arima = ARIMA(vimana_df.demand[0:30].astype(np.float64).as_matrix(),
              order = (1,0,0))
ar_model = arima.fit()
```

Model Summary

```
ar_model.summary2()
```

Model:	ARMA	BIC:	375.7336
Dependent Variable:	y	Log-Likelihood:	–182.77
Date:	2019-03-01 13:40	Scale:	1.0000
No. Observations:	30	Method:	css-mle
Df Model:	2	Sample:	0
Df Residuals:	28		0
Converged:	1.0000	S.D. of innovations:	106.593
No. Iterations:	14.0000	HQIC:	372.875
AIC:	371.5300		

Continued

	Coef.	Std.Err.	*t*	$P > \|t\|$	[0.025	0.975]
const	513.4433	35.9147	14.2962	0.0000	443.0519	583.8348
ar.L1 .y	0.4726	0.1576	2.9995	0.0056	0.1638	0.7814

	Real	Imaginary	Modulus	Frequency
AR.1	2.1161	0.0000	2.1161	0.0000

Forecast and Measure Accuracy

```
forecast_31_37 = ar_model.predict(30, 36)
```

```
forecast_31_37
```

```
array([480.15343682, 497.71129378, 506.00873185, 509.92990963,
       511.78296777, 512.65868028, 513.07252181 ])
```

```
get_mape( vimana_df.demand[30:],
          forecast_31_37 )
```

The MAPE of the AR model with lag 1 is 19.12.

Building MA model

```
arima = ARIMA(vimana_df.demand[0:30].astype(np.float64).as_matrix(),
              order = (0,0,1))
ma_model = arima.fit()
ma_model.summary2()
```

Model:	ARMA	BIC:	378.7982
Dependent Variable:	y	Log-Likelihood:	–184.30
Date:	2019-03-01 13:40	Scale:	1.0000

Continued….

No. Observations:	30	Method:	css-mle
Df Model:	2	Sample:	0
Df Residuals:	28		0
Converged:	1.0000	S.D. of innovations:	112.453
No. Iterations:	15.0000	HQIC:	375.939
AIC:	374.5946		

	Coef.	Std.Err.	*t*	$P > \|t\|$	[0.025	0.975]
const	516.5440	26.8307	19.2520	0.0000	463.9569	569.1312
ma.L1.y	0.3173	0.1421	2.2327	0.0337	0.0388	0.5958

	Real	Imaginary	Modulus	Frequency
MA.1	−3.1518	0.0000	3.1518	0.5000

Continued….

```
forecast_31_37 = ma_model.predict(30, 36)
get_mape(vimana_df.demand[30:],
         forecast_31_37 )
```

```
17.8
```

The MAPE of the MA model with lag 1 is 17.8.

ARMA Model

- Auto-regressive moving average (ARMA) is a combination auto-regressive and moving average process. ARMA(p, q) process combines AR(p) and MA(q) processes.
- The values of p and q in an ARMA process can be identified using the following thumb rules:

1. Auto-correlation values are significant for first q values (first q lags) and cuts off to zero.
2. Partial auto-correlation values are significant for first p values and cuts off to zero.

Building ARMA Model

```
arima = ARIMA( vimana_df.demand[0:30].astype(np.float64).as_
               matrix(), order = (1,0,1))
arma_model = arima.fit()
arma_model.summary2()
```

Model:	ARMA	BIC:	377.2964
Dependent Variable:	y	Log-Likelihood:	–181.85
Date:	2019-03-01 13:41	Scale:	1.0000
No. Observations:	30	Method:	css-mle
Df Model:	3	Sample:	0
Df Residuals:	27		0
Converged:	1.0000	S.D. of innovations:	103.223
No. Iterations:	21.0000	HQIC:	373.485
AIC:	371.6916		

Continued….

	Coef.	Std.Err.	*t*	$P>\|t\|$	[0.025	0.975]
const	508.3995	45.3279	11.2160	0.0000	419.5585	597.2405
ar.L1.y	0.7421	0.1681	4.4158	0.0001	0.4127	1.0715
MA.L1.y	–0.3394	0.2070	–1.6401	0.1126	–0.7451	0.0662

	Real	Imaginary	Modulus	Frequency
AR.1	1.3475	0.0000	1.3475	0.0000
MA.1	2.9461	0.0000	2.9461	0.0000

```
forecast_31_37 = arma_model.predict(30, 36)
get_mape(vimana_df.demand[30:],
         forecast_31_37 )
```

```
20.27
```

ARIMA Model

- The drawback of ARMA models is that ARMA can only be used when the time-series data is stationary. ARIMA models are used when the time-series data is non-stationary.

- ARIMA has the following three components and is represented as ARIMA (p, d, q):
  1. AR component with p lags AR(p).
  2. Integration component (d).
  3. MA with q lags, MA(q).

What is Stationary Data

Time-series data should satisfy the following conditions to be stationary:

1. The mean values of *Yt* at different values of *t* are constant.
2. The variances of *Yt* at different time periods are constant (Homoscedasticity).
3. The covariance of Y_t and Y_{t-k} for different lags depend only on *k* and not on time *t*.

ARIMA Model for Daily Demand of a Product

```
store_df = pd.read_excel('store.xls')
```

```
store_df.head(5)
```

	Date	Demand
0	2014-10-01	15
1	2014-10-02	7
2	2014-10-03	8
3	2014-10-04	10
4	2014-10-05	13

Continued

```
store_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 115 entries, 0 to 114
Data columns (total 2 columns):
Date      115 non-null datetime64[ns]
demand    115 non-null int64
dtypes: datetime64[ns](1), int64(1)
memory usage: 1.9 KB
```

Creating DataTime Index

```
store_df.set_index(pd.to_datetime(store_df.Date),inplace=True)
store_df.drop('Date', axis = 1, inplace = True)
store_df[-5:]
```

Now we print the last 5 records to make sure that index is created correctly and is sorted by date.

Date	Demand
2015-01-19	18
2015-01-20	22
2015-01-21	22
2015-01-22	21
2015-01-23	17

Demand Against Time

```
plt.figure( figsize=(10,4))
plt.xlabel( "Date" )
plt.ylabel( "Demand" )
plt.plot( store_df.demand );
```

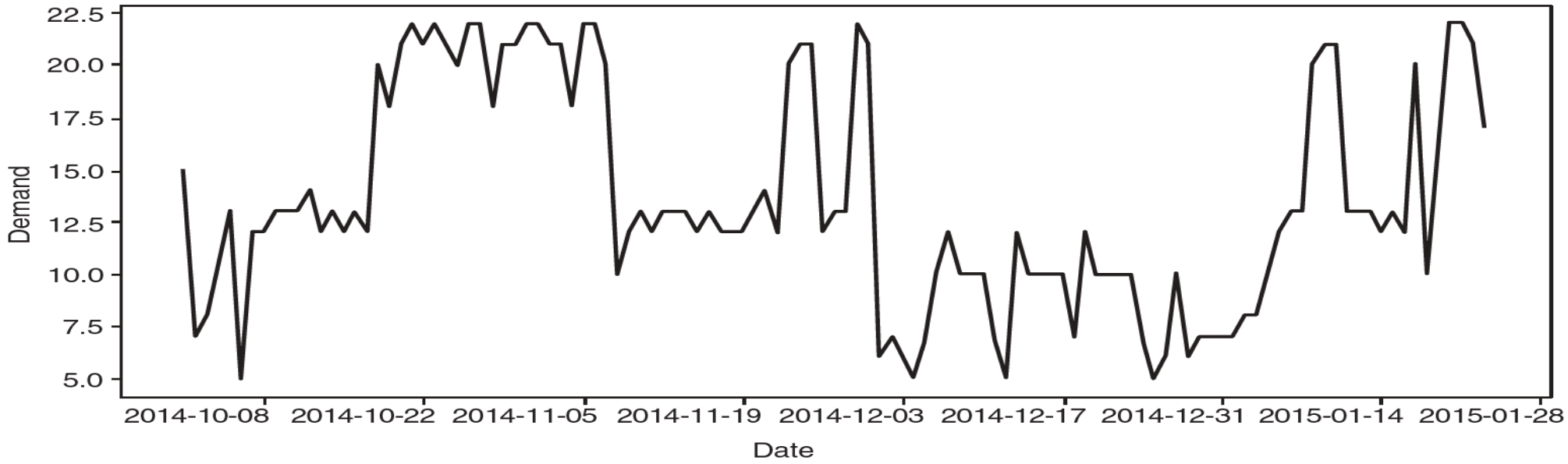
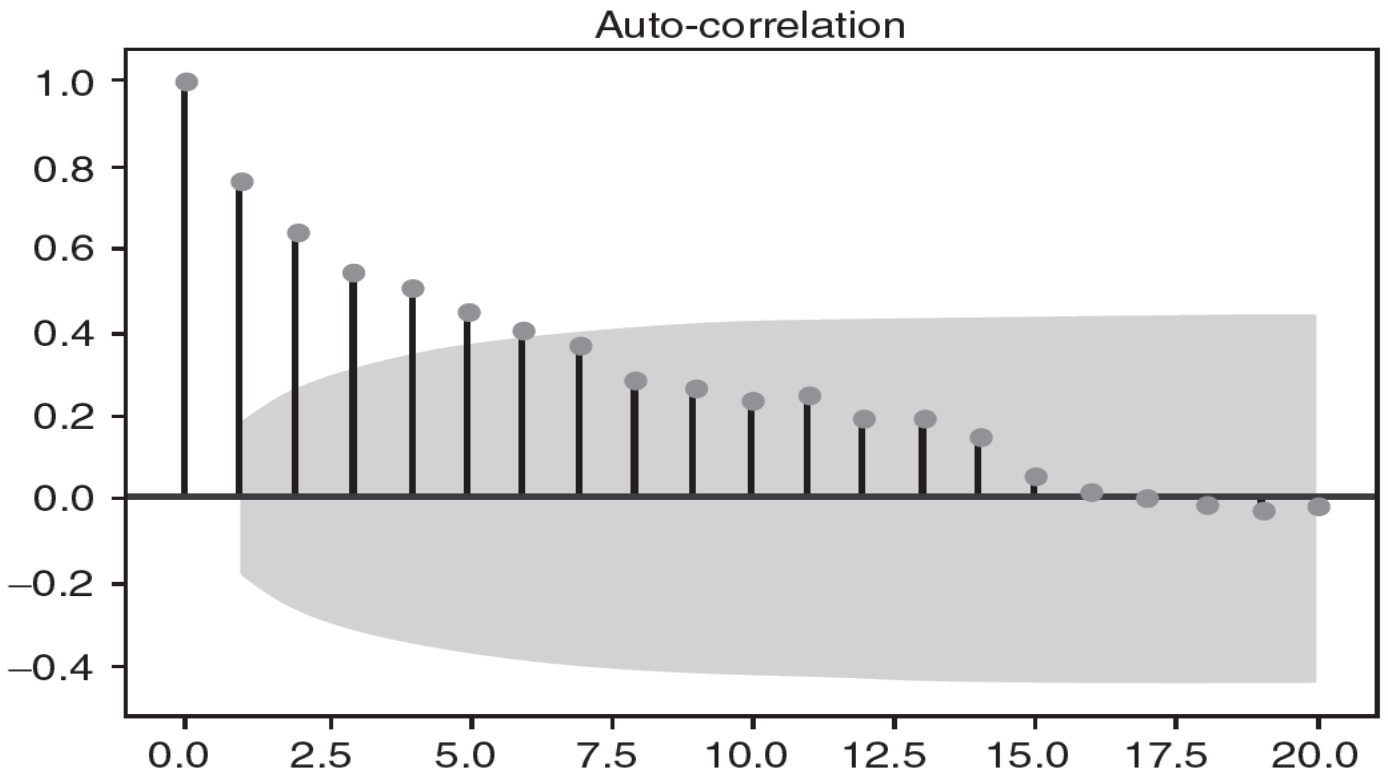


FIGURE 8.7 Daily store demand.

ACF Plot

```
acf_plot = plot_acf( store_df.demand,
                     lags=20 )
```



Dickey-Fuller Test

- Dickey Fuller Test can be used to check if a time-series is stationary.
- This test checks whether the β in the equation given below is less than 1 or equal to 1.

$$Y_{t+1} = \mu + \beta Y_t + \varepsilon_{t+1}$$

- It is a hypothesis test where null and alternate hypotheses are

H_0 : β = 1 (the time-series is non-stationary)

H_1 : β < 1 (the time-series is stationary)

Applying Dickey-Fuller Test

```
from statsmodels.tsa.stattools import adfuller
```

```
def adfuller_test(ts):
    adfuller_result = adfuller(ts, autolag=None)
    adfuller_out = pd.Series(adfuller_result[0:4],
                             index=['Test Statistic',
                                    'p-value',
                                    'Lags Used',
                                    'Number of Observations Used'])
    print(adfuller_out)
    adfuller_text(stone_df.demand)
```

```
Test Statistic                  -1.65
p-value                          0.46
Lags Used                       13.00
Number of Observations Used    101.00
dtype: float64
```

The *p*-value (>0.05) indicates that we cannot reject the null hypothesis and hence, the series is not stationary.

Differencing

Differencing the original time series is an usual approach for converting the non-stationary process into a stationary process (called *difference stationarity*). For example, the first difference (d = 1) is the difference between consecutive values of the time series (Y_t). That is, the first difference is given by

$$\Delta Y_t = Y_t - Y_{t-1}$$

Continued….

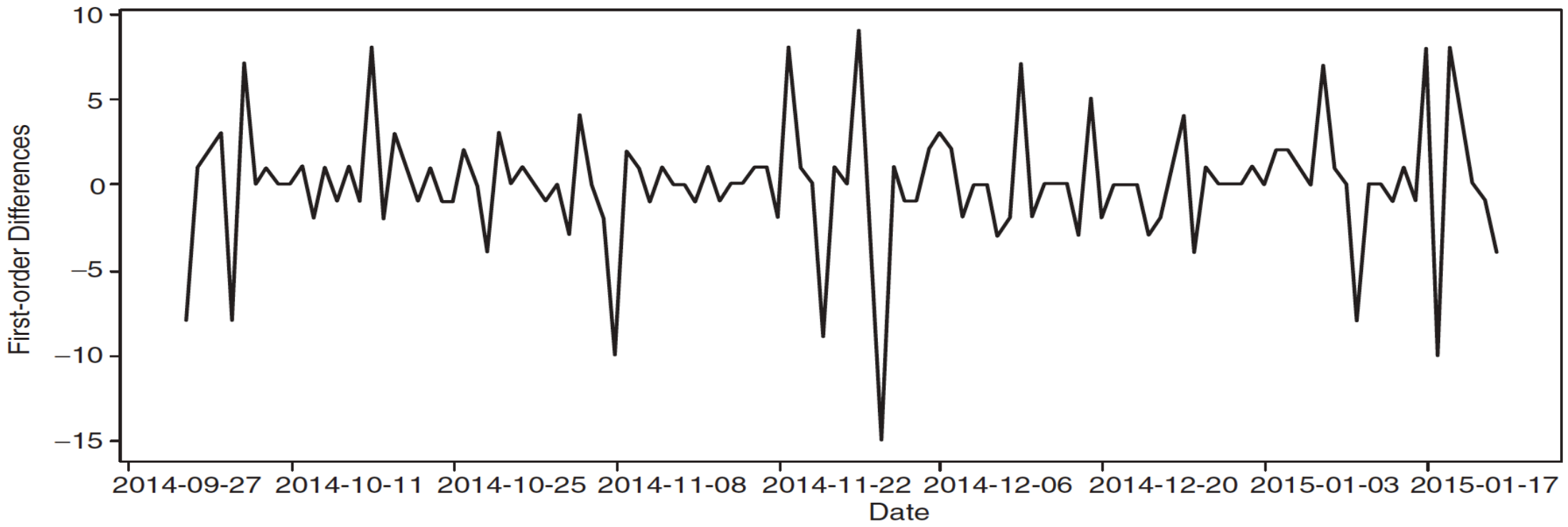
```
store_df['demand_diff'] = store_df.demand - store_df.demand.shift(1)
```

```
store_df.head(5)
```

Date	demand	demand_diff
2014-10-01	15	nan
2014-10-02	7	–8.00
2014-10-03	8	1.00
2014-10-04	10	2.00
2014-10-05	13	3.00

Plotting Differenced Values

```
plt.figure(figsize=(10,4))
plt.xlabel("Date")
plt.ylabel("First Order Differences")
plt.plot( store_diff_df.demand_diff);
```



Building ARIMA Model

Creating training and test sets

```
store_train = store_df[0:100]
store_test = store_df[100:]
```

Building the model with training set

```
arima = ARIMA(store_train.demand.astype(np.float64).as_matrix(),
              order = (1,1,1))
arima_model = arima.fit()
arima_model.summary2()
```

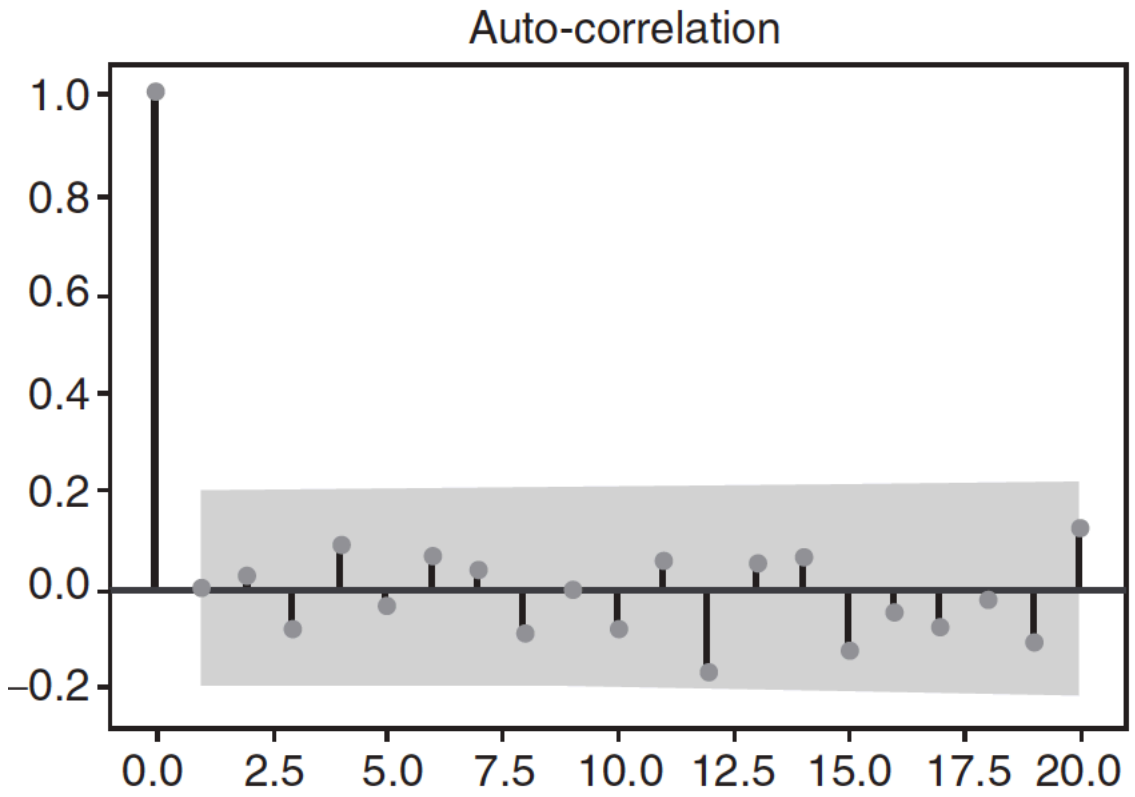
Continued….

Model:	ARIMA	BIC:	532.1510
Dependent Variable:	D.y	Log-Likelihood:	–256.89
Date:	2019-03-01 13:42	Scale:	1.0000
No. Observations:	99	Method:	css-mle
Df Model:	3	Sample:	1
Df Residuals:	96		0
Converged:	1.0000	S.D. of innovations:	3.237
No. Iterations:	10.0000	HQIC:	525.971
AIC:	521.7706		

	Coef.	Std.Err.	*t*	*P* > |*t*|	[0.025	0.975]
const	0.0357	0.1599	0.2232	0.8238	–0.2776	0.3490
ar.L1.D.y	0.4058	0.2294	1.7695	0.0800	–0.0437	0.8554
ma.L1.D.y	–0.7155	0.1790	–3.9972	0.0001	–1.0663	–0.3647

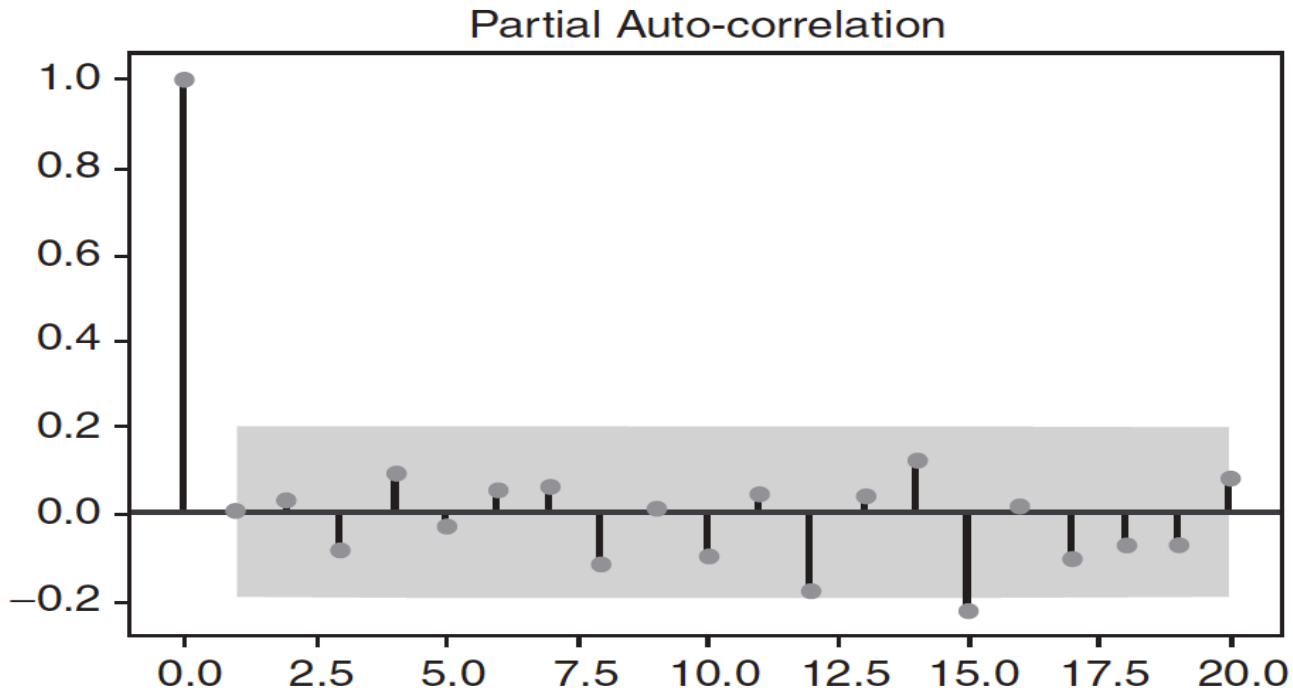
ACF Plot for Residuals

```
acf_plot = plot_acf(arima_model.resid,
                    lags = 20)
```



PACF Plot for Residuals

```
pacf_plot = plot_pacf(arima_model.resid,
                      lags = 20)
```



The ACF and PACF plots don’t show any auto-correlation of residuals.

Forecast and Measure Accuracy

```
store_predict, stderr, ci = arima_model.forecast(steps = 15)
```

```
store_predict
```

```
array([ 17.32364962,   16.2586981,    15.84770837,   15.70211912,
        15.66423863,   15.67007012,   15.69364144,   15.7244122,
        15.75810475,   15.79298307,   15.8283426,    15.86389744,
        15.89953153,   15.93519779,   15.9708771 ])
```

```
get_mape(store_df.demand[100:],
         store_predict)
```

```
24.17
```

The ARIMA model with first-order differencing gives forecast accuracy of 24.17%.

Thank You!