B.TECH. (2020-24)
Artificial Intelligence

TIME SERIES ANALYSIS FOR AI

[CSE471]

Submitted To
Dr Sneha Sharma

Submitted By
HITESH
A023119820027
7AI 1

DEPARTMENT OF ARTIFICIAL INTELLIGENCE
AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY
AMITY UNIVERSITY UTTAR PRADESH
NOIDA (U.P)

INDEX

Exp. No.	Name of Experiment	Date of Allotment of experiment	Date of Evaluation	Remarks	Signature of Faculty
1	Exploratory data analysis of time series data.	20/07/2023	27/07/2023		
2	Handling missing values in data using: Forward fill method (Last observation carried forward), Backward fill method (Next observation carried backwards), Linear interpolation, Spline interpolation, Seasonal decomposition, and interpolation.	27/07/2023	03/08/2023		
3	Decompose Time-Series to See Components (Trend, Seasonality, Noise, etc), Dicky-Fuller Test for Stationarity, Remove Trend (Logged Transformation, Power Transformation, Applying Moving Window Functions, Applying Moving Window Function on Log Transformed Time-Series, Applying Moving Window Function on Power Transformed Time-Series, Applying Linear Regression to Remove Trend)	03/08/2023	17/08/2023		
4	To Remove Seasonality (Differencing Over Log Transformed Time-Series, Differencing Over Power Transformed Time-Series, Differencing Over Time-Series with Rolling Mean taken over 12 Months, Differencing Over Power Transformed & Mean Rolled Time-Series, Differencing Over Linear Regression Transformed Time-Series),), Dicky-Fuller Test for Stationarity	17/08/2023	24/08/2023		
5	Implementation of Auto regression (AR) model and using Auto correlation function (ACF) to find the order of AR model.	24/08/2023	14/09/2023		
6	Implementing Autoregressive integrated moving average (ARIMA) model, also implement Auto-ARIMA model.	14/09/2023	28/09/2023		
7	Implementing Random Forest Regressor Model for time series Forecasting	28/09/2023	05/10/2023		
8	Implementing 1D CNN for time series Forecasting	05/10/2023	12/10/2023		
9	Implementation of multivariate forecasting using Vector AutoRegressive (VAR) model	12/10/2023	19/10/2023		
10	Time series forecasting using Recurrent neural network (RNN) and LSTM (Long short-term memory)	19/10/2023	26/10/2023		

Experiment 1

AIM

Exploratory data analysis of time series data.

SOFTWARE USED

Google Colaboratory

CODE AND OUTPUT

Hitesh_Lab1_July_20_2023.ipynb

File Edit View Insert Runtime Tools Help Last edited on August 27

Comment Share

+ Code + Text

Connect

```
[ ]  1 import numpy as np # linear algebra
     2 import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
     3
```

```
[ ]  1 #read the csv file using pandas
     2 data = pd.read_csv('/content/month.csv')
```

```
[ ]  1 # preview the data using head or tail command
```

```
[ ]  1 # check data info
     2 data.info
```

```
<bound method DataFrame.info of         Period       Revenue  Sales_quantity  Average_cost  \
0   01.01.2015  1.601007e+07        12729.0   1257.763541
1   01.02.2015  1.580759e+07        11636.0   1358.507000
2   01.03.2015  2.204715e+07        15922.0   1384.697024
3   01.04.2015  1.881458e+07        15227.0   1235.606705
4   01.05.2015  1.402148e+07         8620.0   1626.621765
..         ...           ...            ...           ...
91  01.08.2022           NaN            NaN           NaN
92  01.09.2022           NaN            NaN           NaN
93  01.10.2022           NaN            NaN           NaN
94  01.11.2022           NaN            NaN           NaN
95  01.12.2022           NaN            NaN           NaN

    The_average_annual_payroll_of_the_region
0                                30024676.0
1                                30024676.0
2                                30024676.0
3                                30024676.0
4                                30024676.0
..                                      ...
91                                      NaN
92                                      NaN
93                                      NaN
94                                      NaN
95                                      NaN

[96 rows x 5 columns]>
```

```
[ ]  1 data.describe()
```

	Revenue	Sales_quantity	Average_cost	The_average_annual_payroll_of_the_region
count	6.400000e+01	64.000000	64.000000	6.400000e+01
mean	3.236045e+07	19197.375000	1695.061159	2.869083e+07
std	1.164150e+07	6591.287257	296.844793	1.057191e+06
min	1.402148e+07	8314.000000	1110.576805	2.740647e+07
25%	2.242655e+07	15065.500000	1499.142841	2.782857e+07
50%	3.209088e+07	18368.000000	1654.399798	2.819785e+07
75%	3.992999e+07	22856.250000	1916.401096	2.987852e+07
max	5.875647e+07	38069.000000	2559.328184	3.002468e+07

```
[ ]  1 # check the number of unique values in the period column
     2
     3 data.Period.unique()
```

```
array(['01.01.2015', '01.02.2015', '01.03.2015', '01.04.2015',
       '01.05.2015', '01.06.2015', '01.07.2015', '01.08.2015',
       '01.09.2015', '01.10.2015', '01.11.2015', '01.12.2015',
       '01.01.2016', '01.02.2016', '01.03.2016', '01.04.2016',
       '01.05.2016', '01.06.2016', '01.07.2016', '01.08.2016',
       '01.09.2016', '01.10.2016', '01.11.2016', '01.12.2016',
       '01.01.2017', '01.02.2017', '01.03.2017', '01.04.2017',
       '01.05.2017', '01.06.2017', '01.07.2017', '01.08.2017',
       '01.09.2017', '01.10.2017', '01.11.2017', '01.12.2017',
       '01.01.2018', '01.02.2018', '01.03.2018', '01.04.2018',
       '01.05.2018', '01.06.2018', '01.07.2018', '01.08.2018',
       '01.09.2018', '01.10.2018', '01.11.2018', '01.12.2018',
       '01.01.2019', '01.02.2019', '01.03.2019', '01.04.2019',
       '01.05.2019', '01.06.2019', '01.07.2019', '01.08.2019',
       '01.09.2019', '01.10.2019', '01.11.2019', '01.12.2019',
       '01.01.2020', '01.02.2020', '01.03.2020', '01.04.2020',
       '01.05.2020', '01.06.2020', '01.07.2020', '01.08.2020',
       '01.09.2020', '01.10.2020', '01.11.2020', '01.12.2020',
       '01.01.2021', '01.02.2021', '01.03.2021', '01.04.2021',
       '01.05.2021', '01.06.2021', '01.07.2021', '01.08.2021',
       '01.09.2021', '01.10.2021', '01.11.2021', '01.12.2021',
       '01.01.2022', '01.02.2022', '01.03.2022', '01.04.2022',
       '01.05.2022', '01.06.2022', '01.07.2022', '01.08.2022',
       '01.09.2022', '01.10.2022', '01.11.2022', '01.12.2022'],
      dtype=object)
```

+ Code + Text

```python
# if there is any null value then calculate the sum of it

data.isnull().sum().sum()
```

128

```python
# preview the data to see the presence of null values in that dataset

data.isnull().values.any()
```

True

```python
# check where are the null values present in the datasets

data.isnull().values
```

```
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
```

```python
# remove the rows that has those null values

data.dropna()
```

	Period	Revenue	Sales_quantity	Average_cost	The_average_annual_payroll_of_the_region
0	01.01.2015	1.601007e+07	12729.0	1257.763541	30024676.0
1	01.02.2015	1.580759e+07	11636.0	1358.507000	30024676.0
2	01.03.2015	2.204715e+07	15922.0	1384.697024	30024676.0
3	01.04.2015	1.881458e+07	15227.0	1235.606705	30024676.0
4	01.05.2015	1.402148e+07	8620.0	1626.621765	30024676.0
...	...	...	...	...	...
59	01.12.2019	5.875647e+07	38069.0	1543.420464	29878525.0
60	01.01.2020	5.628830e+07	27184.0	2070.640850	29044998.0
61	01.02.2020	4.022524e+07	23509.0	1711.057181	29044998.0
62	01.03.2020	5.002217e+07	32569.0	1535.882748	29044998.0
63	01.04.2020	5.232069e+07	26615.0	1965.834790	29044998.0

64 rows × 5 columns

We have been given revenue values for continuous 64 months, we have to predict for 32 months

```python
#convert dates in a DataFrame to Pandas datetime or Period objects with a specific frequency
data['Period']=pd.to_datetime(data['Period'],format="%d.%m.%Y")
data['Period']=data['Period'].dt.to_period('M')
```

```python
# get data infor
data.info
```

```
<bound method DataFrame.info of      Period       Revenue  Sales_quantity  Average_cost  \
0   2015-01  1.601007e+07         12729.0   1257.763541
1   2015-02  1.580759e+07         11636.0   1358.507000
2   2015-03  2.204715e+07         15922.0   1384.697024
3   2015-04  1.881458e+07         15227.0   1235.606705
4   2015-05  1.402148e+07          8620.0   1626.621765
..      ...           ...             ...           ...
91  2022-08           NaN             NaN           NaN
92  2022-09           NaN             NaN           NaN
93  2022-10           NaN             NaN           NaN
94  2022-11           NaN             NaN           NaN
95  2022-12           NaN             NaN           NaN

    The_average_annual_payroll_of_the_region
0                                 30024676.0
1                                 30024676.0
2                                 30024676.0
3                                 30024676.0
4                                 30024676.0
..                                       ...
91                                       NaN
92                                       NaN
93                                       NaN
94                                       NaN
95                                       NaN

[96 rows x 5 columns]>
```

```python
ts=data.loc[0:63,['Period','Revenue','Sales_quantity']]
ts
```

	Period	Revenue	Sales_quantity
0	2015-01	1.601007e+07	12729.0
1	2015-02	1.580759e+07	11636.0
2	2015-03	2.204715e+07	15922.0
3	2015-04	1.881458e+07	15227.0
4	2015-05	1.402148e+07	8620.0
...	...	...	...
59	2019-12	5.875647e+07	38069.0
60	2020-01	5.628830e+07	27184.0
61	2020-02	4.022524e+07	23509.0
62	2020-03	5.002217e+07	32569.0
63	2020-04	5.232069e+07	26615.0

64 rows × 3 columns

```
[ ] 1 #extract the month and year from dates in a DataFrame
    2 ts['Month']=ts['Period'].dt.month
    3 ts['Year']=ts['Period'].dt.year
    4 ts.head()
```

	Period	Revenue	Sales_quantity	Month	Year
0	2015-01	1.601007e+07	12729.0	1	2015
1	2015-02	1.580759e+07	11636.0	2	2015
2	2015-03	2.204715e+07	15922.0	3	2015
3	2015-04	1.881458e+07	15227.0	4	2015
4	2015-05	1.402148e+07	8620.0	5	2015

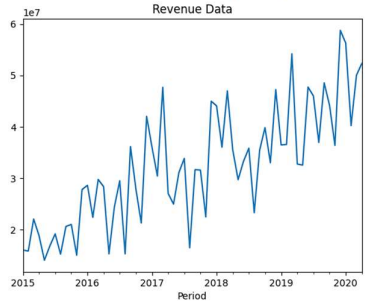
```
[ ] 1 ts.set_index('Period',inplace=True)
    2 ts.head()
```

Period	Revenue	Sales_quantity	Month	Year
2015-01	1.601007e+07	12729.0	1	2015
2015-02	1.580759e+07	11636.0	2	2015
2015-03	2.204715e+07	15922.0	3	2015
2015-04	1.881458e+07	15227.0	4	2015
2015-05	1.402148e+07	8620.0	5	2015

```
[ ] 1 import matplotlib.pyplot as plt
    2 import seaborn as sns
    3 %matplotlib inline
```

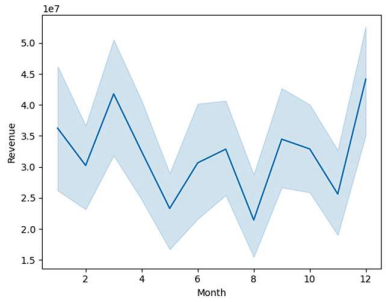
```
[ ] 1 plt.title('Revenue Data')
    2 ts['Revenue'].plot()
```

```
<Axes: title={'center': 'Revenue Data'}, xlabel='Period'>
```



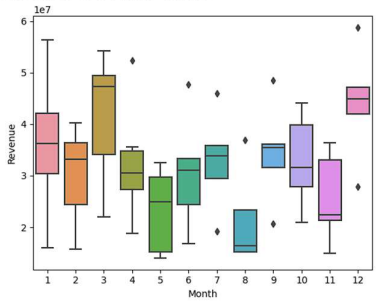

```
1 sns.lineplot(x='Month',y='Revenue',data=ts)
```

```
<Axes: xlabel='Month', ylabel='Revenue'>
```




```
[ ] 1 sns.boxplot(x='Month',y='Revenue',data=ts)
```

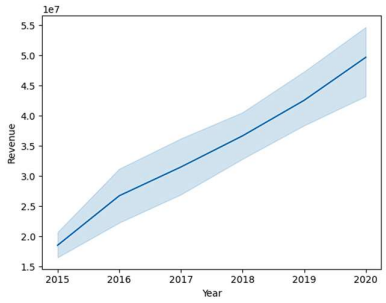
```
<Axes: xlabel='Month', ylabel='Revenue'>
```



It points toward seasonal pattern

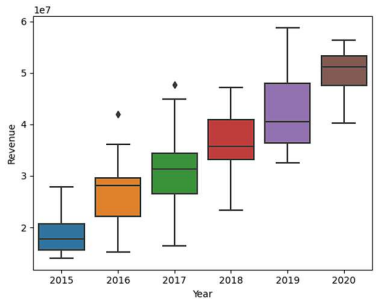
```
[ ] 1 sns.lineplot(x='Year',y='Revenue',data=ts)
```

```
<Axes: xlabel='Year', ylabel='Revenue'>
```




```
[ ]  1 sns.boxplot(x='Year',y='Revenue',data=ts)
```

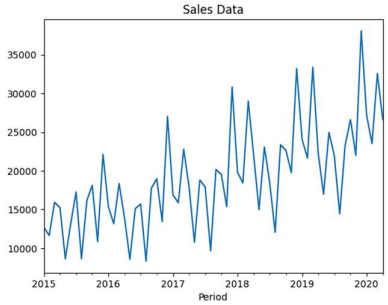
```
<Axes: xlabel='Year', ylabel='Revenue'>
```



This shows there is linearly increasing trend

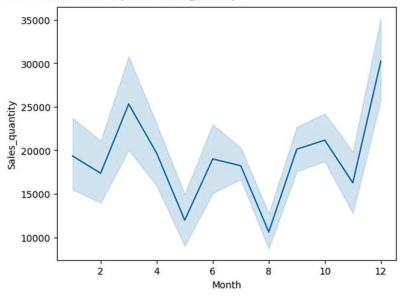
```
[ ]  1 plt.title('Sales Data')
     2 ts['Sales_quantity'].plot()
```

```
<Axes: title={'center': 'Sales Data'}, xlabel='Period'>
```



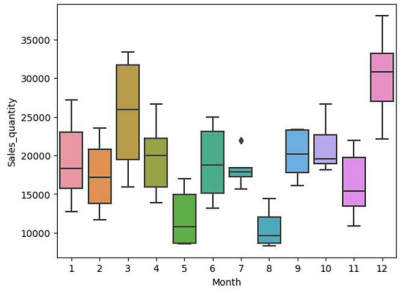

```
[ ]  1 sns.lineplot(x='Month',y='Sales_quantity',data=ts)
```

```
[ ]
<Axes: xlabel='Month', ylabel='Sales_quantity'>
```




```
1 sns.boxplot(x='Month',y='Sales_quantity',data=ts)
```

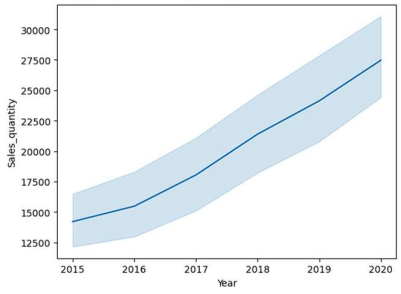
```
<Axes: xlabel='Month', ylabel='Sales_quantity'>
```



It points toward seasonal pattern

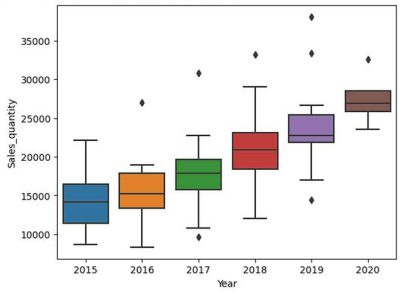
```
[ ]  1 sns.lineplot(x='Year',y='Sales_quantity',data=ts)
```

```
<Axes: xlabel='Year', ylabel='Sales_quantity'>
```




```
[ ]  1 sns.boxplot(x='Year',y='Sales_quantity',data=ts)
```

```
1 sns.boxplot(x='Year',y='Sales_quantity',data=ts)
<Axes: xlabel='Year', ylabel='Sales_quantity'>
```



RESULT

The implementation was successful.

Experiment 2

AIM

Handling missing values in data using: Forward fill method (Last observation carried forward), Backward fill method (Next observation carried backwards), Linear interpolation, Spline interpolation, Seasonal decomposition and interpolation.

SOFTWARE USED

Google Colaboratory

CODE AND OUTPUT

Hitesh_Lab2_July_27_2023.ipynb

File Edit View Insert Runtime Tools Help Last edited on August 27

Comment Share Connect

+ Code + Text

About the Dataset

The dataset contains number of air passengers of each month from the year 1949 to 1960.

```
[ ] 1 import pandas, numpy, matplotlib,seaborn as sns
    2
    3
    4 sns.set(style="whitegrid", color_codes=True, font_scale=1.3)
    5
    6 import warnings
    7 warnings.filterwarnings('ignore')
```

First let´s load the dataset and get a general overview.

```
[ ] 1 original = pandas.read_csv("/content/AirPassengers (1).csv", index_col=['Month'], parse_dates=['Month'])
    2
    3 # read data csv file using pandas and make Month column as index
    4 #original = original.set_index('Month')
    5 # preview data using head command
    6 original.head()
```

Month	#Passengers
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

```
[ ] 1 original.rename(columns = {'#Passengers' : 'Passengers'}, inplace = True)
```

```
1 # preview data using head command
2 original.head()
```

Month	Passengers
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

```
[ ] 1 # check data shape
    2 original.shape
```

```
(144, 1)
```

```
[ ] 1 # use describe command to describe your data
    2 original.describe()
```

	Passengers
count	144.000000
mean	280.298611
std	119.966317
min	104.000000
25%	180.000000
50%	265.500000
75%	360.500000
max	622.000000

The dataset isn´t too complex. We have 2 columns, one for the `month` and the other column is our target.

In total we have 144 rows. Not very much, but let´s see if we can do something with it.

```
[ ]  1 no_missing_vals = original['Passengers'].isnull().sum()
     2 print(f'No of missing observations: {no_missing_vals}')
```

```
No of missing observations: 0
```

if no missing values are in the dataset. We should be happy but for this case we need some missing values to apply the appropiate techniques.
So let´s randomly create some NaN s in our dataset.

```
[ ]  1 import numpy as np
     2 data = original.copy()
     3 # crete a data copy using .copy() command,
     4
     5
     6 # then include some nan values in passengers columns at rows 10:11, 25:29, 40:45, 70:79, 120:125
     7
     8 data.iloc[10:11, data.columns.get_loc('Passengers')] = np.nan
     9 data.iloc[25:29, data.columns.get_loc('Passengers')] = np.nan
    10 data.iloc[40:45, data.columns.get_loc('Passengers')] = np.nan
    11 data.iloc[70:79, data.columns.get_loc('Passengers')] = np.nan
    12 data.iloc[120:125, data.columns.get_loc('Passengers')] = np.nan
```

And check again for missing values.

```
[ ]  1 no_missing_vals = data.isnull().sum()
     2 # check the sum of null values
     3 print(f'No of missing observations: {no_missing_vals}')
     4 print(f'% of missing observations: {no_missing_vals / len(data)}')
```

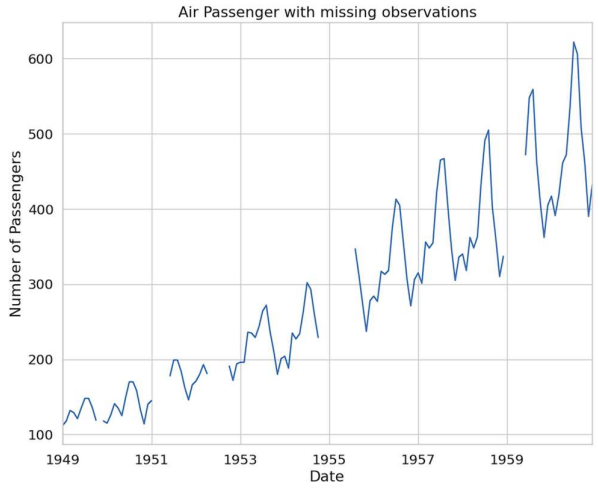
```
No of missing observations: Passengers    24
dtype: int64
% of missing observations: Passengers    0.166667
dtype: float64
```

Now 24 observations are missing in our time series. Roughly 17% of our data are affected.

Lets plot our time series data to see the missing values.

```
[ ]  1 ax = data.plot(legend=None, figsize=(10,8))
     2 ax.set_title('Air Passenger with missing observations')
     3 ax.set_ylabel('Number of Passengers')
     4 ax.set_xlabel('Date')
```

```
[ ] Text(0.5, 0, 'Date')
```



Missing value handling

I will show 5 techniques for missing value handling.

- Forward fill method (Last observation carried forward)
- Backward fill method (Next observation carried backwards)
- Linear interpolation
- Spline interpolation
- Seasonal decomposition and interpolation

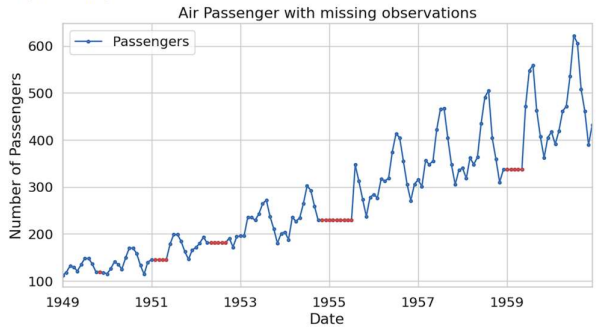
```
[ ]  1 ffm = data.copy()
     2 bfm = data.copy()
     3 li = data.copy()
     4 si = data.copy()
     5 sdi = data.copy()
```

Forward fill method

As the name implies, here the missing data points are filled with the value of the last known data point.

```
[ ]  1 ffm_imputed = ffm.fillna(method='ffill') # fill the null values using fillna and specify the method as 'ffill' for forward fill
     2 ax = ffm_imputed.plot(linestyle="-", marker=".", figsize=[10, 5])
     3 ffm_imputed[ffm.isnull()].plot(ax=ax, legend=None, marker=".", color="r")
     4 ax.set_title('Air Passenger with missing observations')
     5 ax.set_ylabel('Number of Passengers')
     6 ax.set_xlabel('Date')
```

```
Text(0.5, 0, 'Date')
```

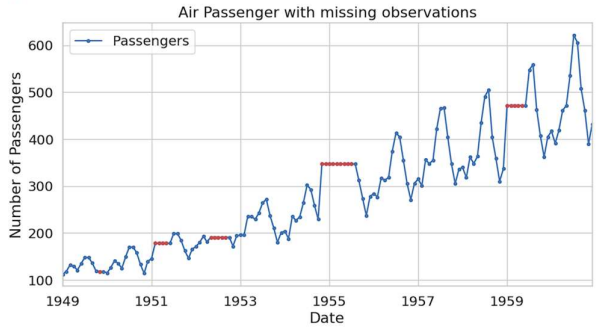


Backward fill method

Same like forward fill method, but this time the missing values will be filled with the next known observation.

```
bfm_imputed = bfm.fillna(method = 'bfill') # fill the null values using fillna and specify the method for backward fill
# plot same as we did in forward fill
ax = bfm_imputed.plot(linestyle="-", marker=".", figsize=[10, 5])
bfm_imputed[bfm.isnull()].plot(ax=ax, legend=None, marker=".", color="r")
ax.set_title('Air Passenger with missing observations')
ax.set_ylabel('Number of Passengers')
ax.set_xlabel('Date')
```

```
Text(0.5, 0, 'Date')
```



As we see in the above plots both methods can distort important structures of the time series (e.g. the shape of the seasonal component) depending on how large the gaps of missing data are. If these gaps are small the distortions are small, if the gaps are large then the distortions are also large.

The distortions to the time series created by data imputation can negatively impact the fitting of a model. If the proportion of missing data is small then the impact of these distortions on the fitting of a model should be less and may be tolerable depending on the use case.

One minor advantage of forward filling over backward filling is that is does not leak any data from the future to the imputed observations.

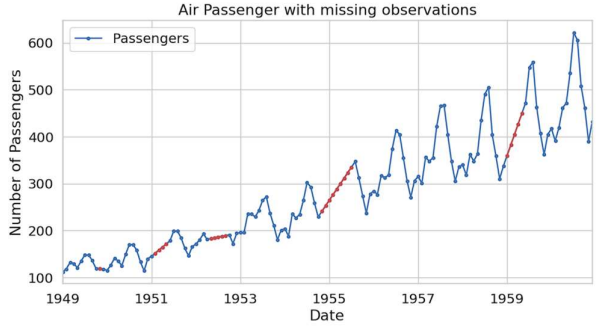
If a dataset has missing data with small gap sizes then the forward filling method would be a sufficient method to impute missing data without distorting the time series too heavily and ensuring no data leakage is occurring. Since we have a big gape size arounnd 1953, 1955 and 1959 we have to search for a better alternative.

Linear interpolation

Linear interpolation in time series refers to a method of estimating missing values or filling in gaps in a time series data by assuming a linear relationship between the available data points. It assumes that the values between the known data points change linearly over time.

```
li_imputed = li.interpolate(method = 'linear') # use the interpolation method using interpolate command and specify the method as linear
# plot the interpolated graph

ax = li_imputed.plot(linestyle="-", marker=".", figsize=[10, 5])
li_imputed[li.isnull()].plot(ax=ax, legend=None, marker=".", color="r")
ax.set_title('Air Passenger with missing observations')
ax.set_ylabel('Number of Passengers')
ax.set_xlabel('Date')
```

```
Text(0.5, 0, 'Date')
```



In the above plot we can see that linear interpolation is reasonable for small gaps but also captures the trend for larger gaps. However, the seasonality is lost.

Spline interpolation

Spline interpolation is used to estimate missing values or fill gaps in a time series data by fitting a smooth curve or spline to the available data points. It ais to capture the underlying pattern of the data more accurately by using a piecewise polynomial function.

Spline interpolation offers several advantages over linear interpolation in time series:

- It can capture more complex and non-linear patterns in the data, as cubic splines can represent a wider range of curves.
- Spline interpolation produces a smoother curve without sharp edges, resulting in a more visually appealing representation.
- It reduces the risk of overfitting compared to higher-degree polynomial interpolation.

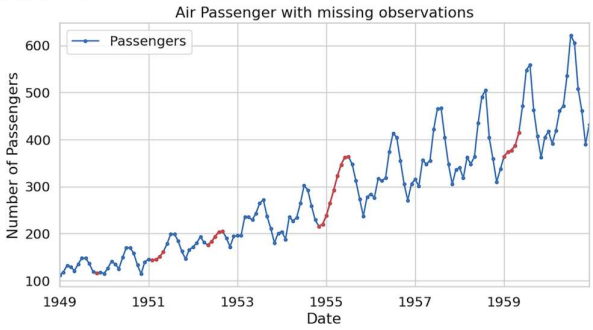
However, spline interpolation can be computationally more intensive and may require more data points to achieve accurate estimates. Additionally, the choice of spline interpolation method, such as cubic or higher-degree splines, may depend on the characteristics of the data and the desired smoothness.

Overall, spline interpolation is a flexible and effective technique for estimating missing values or filling gaps in time series data, providing a more accurate representation of the underlying patterns and dynamics.

```
[ ]  1
```

```
[ ]  1 si_imputed = si.interpolate(method='spline', order=3)
     2 ax = si_imputed.plot(linestyle="-", marker=".", figsize=[10, 5])
     3 si_imputed[si.isnull()].plot(ax=ax, legend=None, marker=".", color="r")
     4 ax.set_title('Air Passenger with missing observations')
     5 ax.set_ylabel('Number of Passengers')
     6 ax.set_xlabel('Date')
```

```
[ ]
Text(0.5, 0, 'Date')
```



Seasonal decomposition and interpolation

This method involves estimating the seasonal component of a time series. This is then subtracted from the original time series to provide a de-seasoned time series. Any of the prior interpolation methods can then be used on the de-seasoned time series and the seasonal component can be added back to the de-seasoned time series.

There are many different methods to decompose a time series into seasonal and other components. We shall use STL for this notebook. n advantage of STL is that it is able to estimate a seasonal component which can change over time (e.g., if the amplitude or frequency of the seasonal component changes).

Seasonal-Trend decomposition using LOESS (STL) is a robust method of time series decomposition often used in economic and environmental analyses. The STL method uses locally fitted regression models to decompose a time series into trend, seasonal, and remainder components.

```
[ ]  1 from statsmodels.tsa.seasonal import STL
```

STL can't handle missing data. A linear interpolation is typically used to fill missing data before decomposing the time series using STL.

```
[ ]  1 type(sdi)
```

```
pandas.core.frame.DataFrame
```

```
[ ]  1 # in this place "use linear interpolation here" use the commands for linear interpolation.
     2 res = STL(sdi.interpolate(method="linear"), seasonal=31).fit()
```

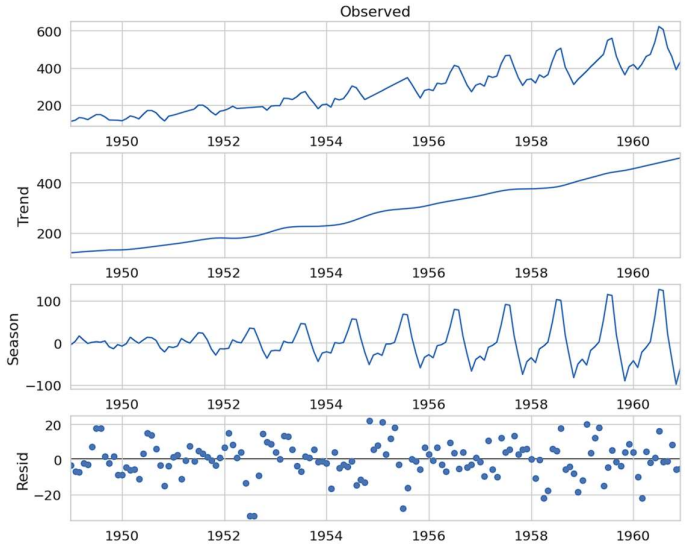
The seasonal parameter determines how much data is used to infer the seasonality at any given point. If the seasonal component is thought to be fixed throughout time then a large seasonal parameter can be set so that more data is used to determine the seasonal component. Likewise if the seasonal component is thought to change (e.g., the frequency) quickly over time the seasonal parameter can be reduced so that only recent data contributes to determining the seasonal component.

The large missing gap is linearly interpolated. This means that the algorithm sees a region of data with no seasonal component and could distort the estimation of the seasonal component. A large seasonal parameter is set to overcome this. This works because a larger portion of the data is now used to estimate the seasonal component meaning that the local distortion from the interpolation has less effect.

We now plot the decomposition to inspect that it is reasonable.

```
[ ]  1 import matplotlib.pyplot as plt
```

```
[ ]  1 plt.rc("figure", figsize=(10, 8))
     2 plt.rc("font", size=5)
     3 res.plot();
```



We now:

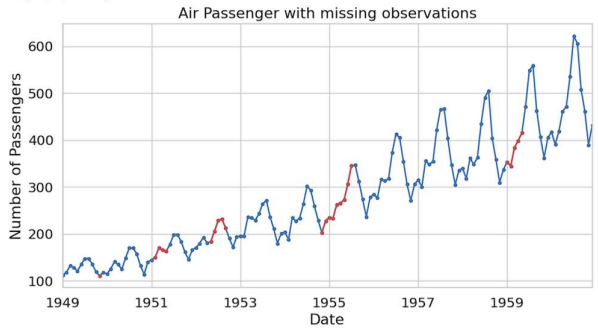
1. extract the seasonal component
2. de-seasonalise the original time series
3. perform linear interpolation on the de-seasonalised data
4. Add the seasonal component back to the imputed de-seasonalised data

```
[ ]  1 seasonal_component = res.seasonal
     2 seasonal_component.head()
```

```
Month
1949-01-01    -4.732242
1949-02-01     3.324701
1949-03-01    16.248299
1949-04-01     6.673724
1949-05-01    -1.617146
Name: season, dtype: float64
```

```
[ ]  1 sdi_deseasonalised = sdi['Passengers'] - seasonal_component
     2 sdi_deseasonalised_imputed = sdi_deseasonalised.interpolate(method="linear")
     3 sdi_imputed = sdi_deseasonalised_imputed + seasonal_component
     4 sdi_imputed = sdi_imputed.to_frame().rename(columns={0: 'Passengers'})
     5 ax = sdi_imputed.plot(linestyle="-", marker=".", figsize=[10, 5], legend=None)
     6 ax = sdi_imputed[sdi.isnull()].plot(ax=ax, legend=None, marker=".", color="r")
     7 ax.set_title('Air Passenger with missing observations')
     8 ax.set_ylabel('Number of Passengers')
     9 ax.set_xlabel('Date')
```

```
[ ]  Text(0.5, 0, 'Date')
```

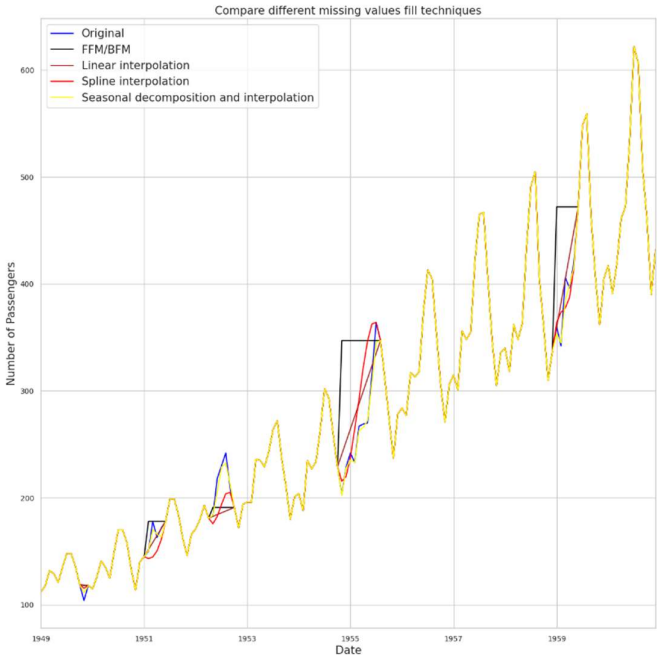


Simpler seasonal decompositions methods exist in Statsmodels such as seasonal_decompose. However, the simplicity of the seasonal_decompose method has drawbacks which may or may not be relevant for your use case (e.g., naive seasonal decomposition methods may not capture seasonality which changes over time, may not return values at the start and end of a time series etc.). STL is a more advanced method that is commonly used for data imputation.

Wrap up

Now let`s wrap up the missing value section and plot the result of each technique in on plot to choose the best fit.

```
[ ]  1 fig, ax = plt.subplots(figsize=(15,15))
     2 original.plot(ax=ax, color='blue', label='Original')
     3 bfm_imputed.plot(ax=ax, color='black', label='FFM/BFM')
     4 li_imputed.plot(ax=ax, color='brown', label='Linear interpolation')
     5 si_imputed.plot(ax=ax, color='red', label='Spline interpolation')
     6 sdi_imputed.plot(ax=ax, color='yellow', label='Seasonal decomposition and interpolation')
     7 plt.xlabel('Date', fontsize=15)
     8 plt.ylabel('Number of Passengers', fontsize=15)
     9 plt.title('Compare different missing values fill techniques', fontsize=15)
    10 plt.xticks(fontsize=10)
    11 plt.yticks(fontsize=10)
    12 plt.legend(fontsize=15)
    13 ax.legend(['Original', 'FFM/BFM', 'Linear interpolation', 'Spline interpolation', 'Seasonal decomposition and interpolation'], fontsize=15);
    14 plt.show()
```



RESULT

The implementation was successful.

Experiment 3

AIM

Decompose Time-Series to See Components (Trend, Seasonality, Noise, etc), Dicky-Fuller Test for Stationarity, Remove Trend (Logged Transformation, Power Transformation, Applying Moving Window Functions, Applying Moving Window Function on Log Transformed Time-Series, Applying Moving Window Function on Power Transformed Time-Series, Applying Linear Regression to Remove Trend).

SOFTWARE USED

Google Colaboratory

CODE AND OUTPUT

Hitesh_Lab3_August_03_2023.ipynb

File Edit View Insert Runtime Tools Help Last edited on August 27

Comment Share Connect

+ Code + Text

TSA LAB 3: Seasonal Decomposition, Stationairy Test, ADF Test - Air Passengers Dataset

Load Time Series Dataset

We'll now explore trend and seasonality removal with examples. We'll be using famous air passenger datasets available on-line for our purpose because it has both trend and seasonality. It has information about US airline passengers from 1949 to 1960 recorded each month. Please download the dataset to follow along.

*Air Passengers Dataset

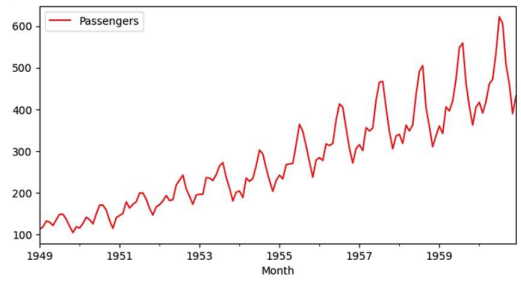
```
[ ] 1 # import pandas, numpy, matplotlib
    2 import pandas as pd
    3 import numpy as np
    4 import matplotlib.pyplot as plt
    5
```

```
[ ] 1 air_passengers= pd.read_csv("/content/AirPassengers (1).csv", index_col = 0, parse_dates= True)# read the data using pandas include (index_col=0, parse_dates=True)
    2
    3 air_passengers.rename(columns = {"#Passengers":"Passengers"}, inplace=True)
    4 air_passengers.head()# preview data using head command
    5
```

Month	Passengers
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

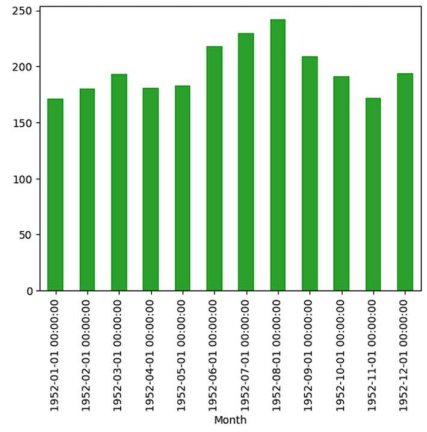
```
1 air_passengers.plot(figsize=(8,4), color="tab:red")
```

```
<Axes: xlabel='Month'>
```




```
[ ] 1 air_passengers["1952"].plot(kind="bar", color="tab:green", legend=False);
```

```
air_passengers["1952"].plot(kind="bar", color="tab:green", legend=False);
```



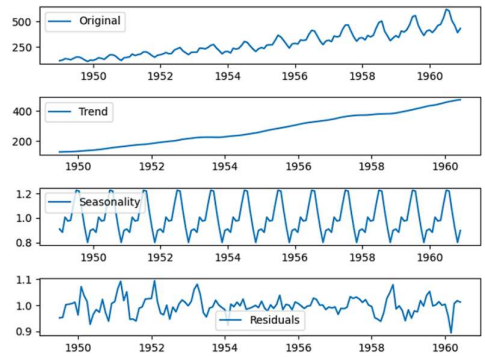
By looking at the above plots we can see that our time-series is multiplicative time-series and has both trend as well as seasonality. We can see the trend as passengers are constantly increasing over time. We can see seasonality with the same variations repeating for 1 year where value peaks somewhere are around August.

Decompose Time-Series to See Components (Trend, Seasonality, Noise, etc)

We can decompose time-series to see various components of time-series. Python module named statmodels provides us with easy to use utility which we can use to get an individual component of time-series and then visualize it.

```
#import seasonal decomposing using statsmodel and seasonal decompose
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.tsa.arima_model import ARIMA
```

```
decompose_result = seasonal_decompose(air_passengers, model='multiplicative', filt=None, period=None, two_sided=True, extrapolate_trend=0)

 # perform seasonal decomposing using multiplicative model type

trend = decompose_result.trend # get trend
seasonal = decompose_result.seasonal #get seasonal
residual = decompose_result.resid #get residual

#plot the decompose result
# decompose_result.plot("Multiplicative Decomposition")

plt.subplot(411)
plt.plot(air_passengers, label='Original')
plt.legend(loc='best')

plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')

plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')

plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```



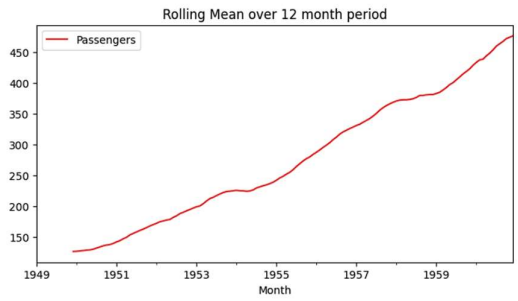
We can notice trend and seasonality components separately as well as residual components. There is a loss of residual in the beginning which is settling later.

Checking Whether Time-Series is Stationary or Not

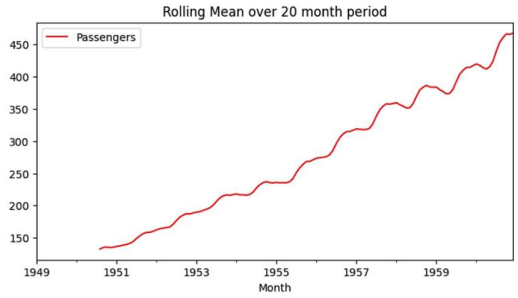
As we declared above time-series is stationary whose mean, variance and auto-covariance are independent of time. We can check mean, variance and auto-covariance using moving window functions available with pandas. We'll also use a dicky-fuller test available with statsmodels to check the stationarity of time-series. If time-series is not stationary then we need to make it stationary.

Below we have taken an average over moving window of 12 samples. We noticed from the above plots that there is the seasonality of 12 months in time-series. We can try different window sizes for testing purposes.

```
[ ]  1 air_passengers.rolling(window = 12).mean().plot(figsize=(8,4), color="tab:red", title="Rolling Mean over 12 month period");
```



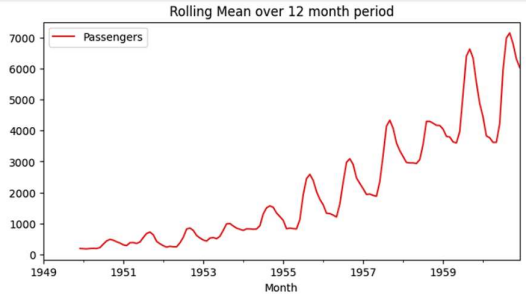

```
[ ]  1 #same as above apply the rolling mean for 20 month period
     2 air_passengers.rolling(window = 20).mean().plot(figsize=(8,4), color="tab:red", title="Rolling Mean over 20 month period");
```



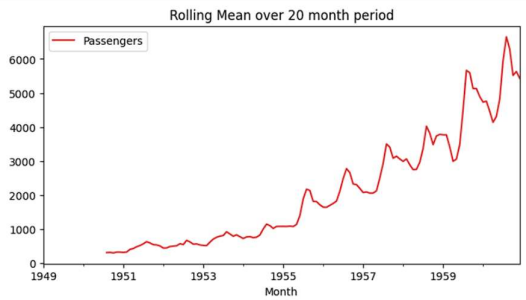
We can clearly see that time-series has a visible upward trend.

Below we have taken variance over the moving window of 12 samples. We noticed from the above plots that there is the seasonality of 12 months in time-series

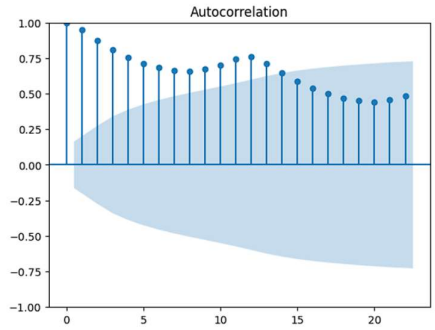
```
[ ]  1 # same as above instead of rolling mean now calculate the variance for 12 months
     2 air_passengers.rolling(window = 12).var().plot(figsize=(8,4), color="tab:red", title="Rolling Mean over 12 month period");
```




```
     1 # same as above instead of rolling mean now calculate the variance for 20 months
     2 air_passengers.rolling(window = 20).var().plot(figsize=(8,4), color="tab:red", title="Rolling Mean over 20 month period");
```



```
[ ]  1 #using from statsmodels.graphics.tsaplots import the autocorrelation function (plot_acf )
     2 from statsmodels.graphics.tsaplots import plot_acf
     3
     4 plot_acf(air_passengers);
```



We can notice from the above chart that after 13 lags, the line gets inside confidence interval (light blue area). This can be due to seasonality of 12-13 months in our data.

```
[ ]  1 from statsmodels.tsa.stattools import adfuller
     2
     3 dftest = adfuller(air_passengers['Passengers'], autolag = 'AIC')
     4
     5 print("1. ADF : ",dftest[0])
     6 print("2. P-Value : ", dftest[1])
     7 print("3. Num Of Lags : ", dftest[2])
     8 print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
     9 print("5. Critical Values :")
    10 for key, val in dftest[4].items():
    11     print("\t",key, ": ", val)
```

```
1. ADF :  0.8153688792060498
2. P-Value :  0.991880243437641
3. Num Of Lags :  13
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 130
5. Critical Values :
	1% :  -3.4816817173418295
	5% :  -2.8840418343195267
	10% :  -2.578770059171598
```

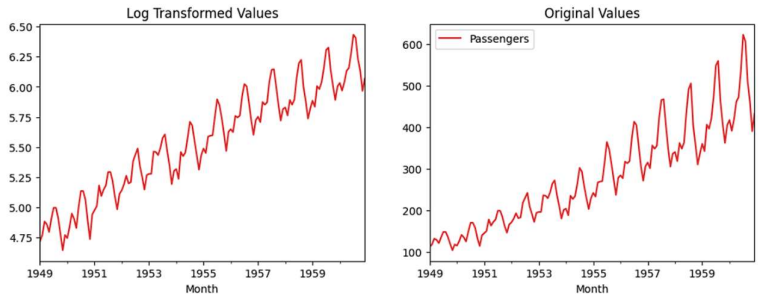
We can interpret above results based on p-values of result.

p-value > 0.05 - This implies that time-series is non-stationary. p-value <=0.05 - This implies that time-series is stationary. We can see from the above results that p-value is greater than 0.05 hence our time-series is not stationary. It still has time-dependent components present which we need to remove.

Logged Transformation

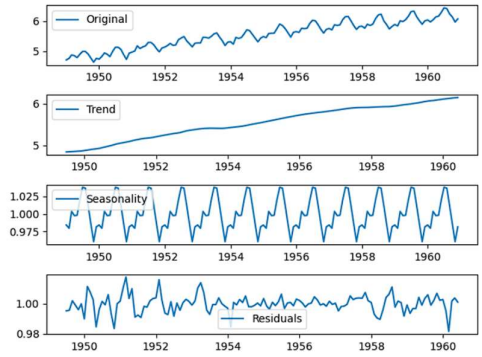
To apply log transformation, we need to take a log of each individual value of time-series data.

```
 1 logged_passengers = air_passengers["Passengers"].apply(lambda x : np.log(x))
 2
 3 ax1 = plt.subplot(121)
 4 logged_passengers.plot(figsize=(12,4) ,color="tab:red", title="Log Transformed Values", ax=ax1);
 5 ax2 = plt.subplot(122)
 6 air_passengers.plot(color="tab:red", title="Original Values", ax=ax2);
```



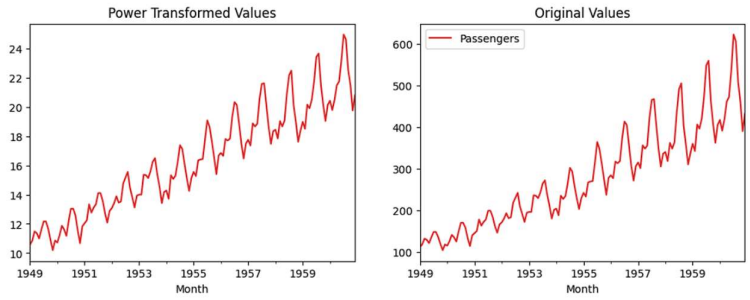
From the above first chart, we can see that we have reduced the variance of time-series data. We can look at y-values of original time-series data and log-transformed time-series data to conclude that the variance of time-series is reduced.

```
[ ]  1 # perform seasonal decompose on logged transformed
     2 decompose_result = seasonal_decompose(logged_passengers, model='multiplicative', filt=None, period=None, two_sided=True, extrapolate_trend=0)
     3
     4 # perform seasonal decomposing using multiplicative model type
     5
     6 trend = decompose_result.trend # get trend
     7 seasonal = decompose_result.seasonal #get seasonal
     8 residual = decompose_result.resid #get residual
     9
    10 #plot the decompose result
    11 # decompose_result.plot("Multiplicative Decomposition")
    12
    13 plt.subplot(411)
    14 plt.plot(logged_passengers, label='Original')
    15 plt.legend(loc='best')
    16
    17 plt.subplot(412)
    18 plt.plot(trend, label='Trend')
    19 plt.legend(loc='best')
    20
    21 plt.subplot(413)
    22 plt.plot(seasonal,label='Seasonality')
    23 plt.legend(loc='best')
    24
    25 plt.subplot(414)
    26 plt.plot(residual, label='Residuals')
    27 plt.legend(loc='best')
    28 plt.tight_layout()
```



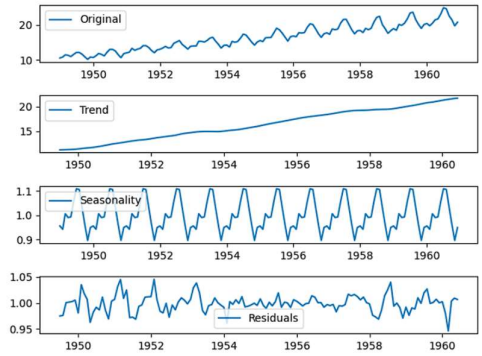
Power Transformations We can apply power transformation in data same way as that of log transformation to remove trend.

```
# perform power transform using (lambda x : x ** 0.5)

power_passengers = air_passengers["Passengers"].apply(lambda x : x ** 0.5)


# plot the curve for the same

ax1 = plt.subplot(121)
power_passengers.plot(figsize=(12,4) ,color="tab:red", title="Power Transformed Values", ax=ax1);
ax2 = plt.subplot(122)
air_passengers.plot(color="tab:red", title="Original Values", ax=ax2);
```

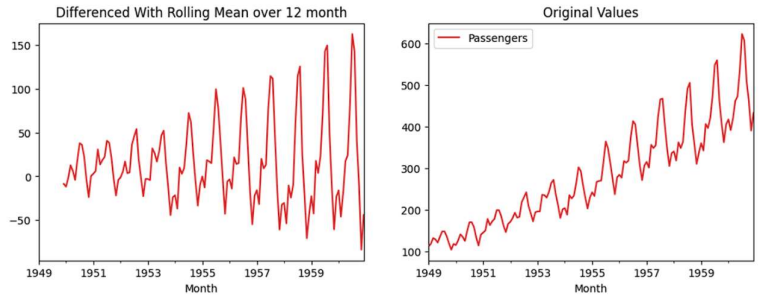


From the above first chart, we can see that we have reduced the variance of time-series data. We can look at y-values of original time-series data and power-transformed time-series data to conclude that the variance of time-series is reduced.

```
# perform seasonal decompose on power transformed
decompose_result = seasonal_decompose(power_passengers, model='multiplicative', filt=None, period=None, two_sided=True, extrapolate_trend=0)

 # perform seasonal decomposing using multiplicative model type

trend = decompose_result.trend # get trend
seasonal = decompose_result.seasonal #get seasonal
residual = decompose_result.resid #get residual

#plot the decompose result
# decompose_result.plot("Multiplicative Decomposition")

plt.subplot(411)
plt.plot(power_passengers, label='Original')
plt.legend(loc='best')

plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')

plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')

plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```

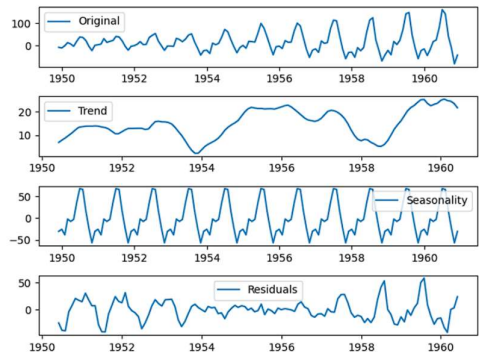


```
[ ]  1 rolling_mean = air_passengers.rolling(window = 12).mean()
     2 passengers_rolled_detrended = air_passengers['Passengers'] - rolling_mean['Passengers']
     3
     4
     5 ax1 = plt.subplot(121)
     6 passengers_rolled_detrended.plot(figsize=(12,4),color="tab:red", title="Differenced With Rolling Mean over 12 month", ax=ax1);
     7 ax2 = plt.subplot(122)
     8 air_passengers.plot(figsize=(12,4), color="tab:red", title="Original Values", ax=ax2);
```




```
[ ]  1 # perform seasonal decompose on logged transformed
     2
     3 decompose_result = seasonal_decompose(passengers_rolled_detrended.dropna(), model='additive', filt=None, period=None, two_sided=True, extrapolate_trend=0)
     4
     5 # perform seasonal decomposing using additive model type
     6
     7 trend = decompose_result.trend # get trend
     8 seasonal = decompose_result.seasonal #get seasonal
     9 residual = decompose_result.resid #get residual
    10
    11 #plot the decompose result
    12 # decompose_result.plot("Multiplicative Decomposition")
    13
    14 plt.subplot(411)
    15 plt.plot(passengers_rolled_detrended, label='Original')
    16 plt.legend(loc='best')
    17
    18 plt.subplot(412)
    19 plt.plot(trend, label='Trend')
    20 plt.legend(loc='best')
    21
    22 plt.subplot(413)
    23 plt.plot(seasonal,label='Seasonality')
    24 plt.legend(loc='best')
    25
    26 plt.subplot(414)
    27 plt.plot(residual, label='Residuals')
    28 plt.legend(loc='best')
    29 plt.tight_layout()
```

```
[ ]
```



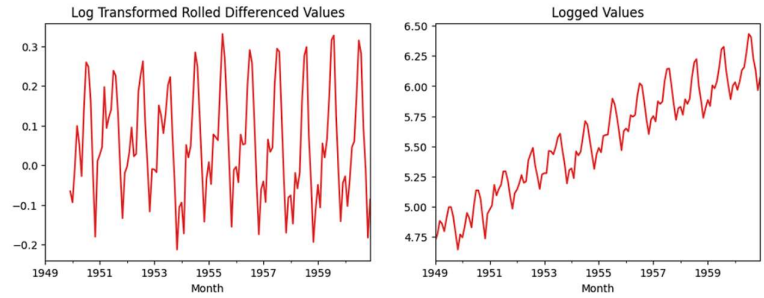
Applying Moving Window Function on Log Transformed Time-Series

We can apply more than one transformation as well. We'll first apply log transformation to time-series, then take a rolling mean over a period of 12 months and then subtract rolled time-series from log-transformed time-series to get final time-series.

```
[ ]  1 logged_passengers
```

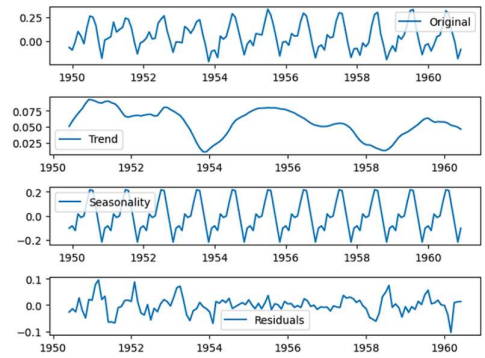
```
Month
1949-01-01    4.718499
1949-02-01    4.770685
1949-03-01    4.882802
1949-04-01    4.859812
1949-05-01    4.795791
              ...
1960-08-01    6.406880
1960-09-01    6.230481
1960-10-01    6.133398
1960-11-01    5.966147
1960-12-01    6.068426
Name: Passengers, Length: 144, dtype: float64
```

```
     1 # take log transformed and rolling mean for 12 months period time
     2 rolling_mean = logged_passengers.rolling(window = 12).mean()
     3 passengers_log_rolled_detrended = logged_passengers - rolling_mean
     4
     5
     6 # plot the passengers_log_rolled_detrended
     7 ax1 = plt.subplot(121)
     8 passengers_log_rolled_detrended.plot(figsize=(12,4) ,color="tab:red", title="Log Transformed Rolled Differenced Values", ax=ax1);
     9 ax2 = plt.subplot(122)
    10 logged_passengers.plot(color="tab:red", title="Logged Values", ax=ax2);
```



From the above the first chart, we can see that we are able to removed the trend from time-series data.

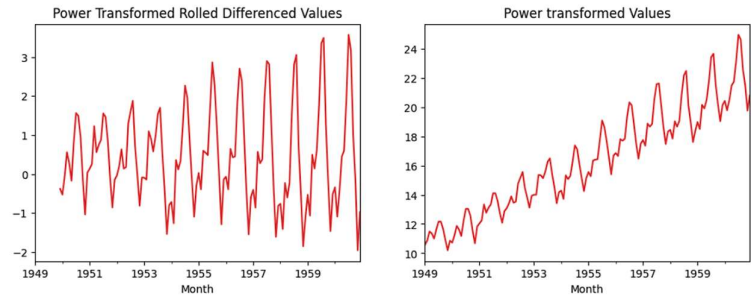
```
# perform seasonal decompose on logged transformed

decompose_result = seasonal_decompose(passengers_log_rolled_detrended.dropna(), model='additive', filt=None, period=None, two_sided=True, extrapolate_trend=0)

# perform seasonal decomposing using additive model type

trend = decompose_result.trend # get trend
seasonal = decompose_result.seasonal #get seasonal
residual = decompose_result.resid #get residual

#plot the decompose result
# decompose_result.plot("Multiplicative Decomposition")

plt.subplot(411)
plt.plot(passengers_log_rolled_detrended, label='Original')
plt.legend(loc='best')

plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')

plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')

plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```



Applying Moving Window Function on Power Transformed Time-Series

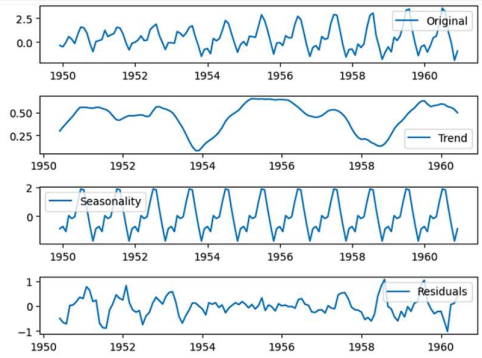
We can apply more than one transformation as well. We'll first apply power transformation to time-series, then take a rolling mean over a period of 12 months and then subtract rolled time-series from power-transformed time-series to get final time-series.

```
rolling_mean = power_passengers.rolling(window = 12).mean()
passengers_pow_rolled_detrended = power_passengers - rolling_mean  # take the difference of power transformed and rolling mean on powered transformed

# plot the passengers_log_rolled_detrended
ax1 = plt.subplot(121)
passengers_pow_rolled_detrended.plot(figsize=(12,4) ,color="tab:red", title="Power Transformed Rolled Differenced Values", ax=ax1);
ax2 = plt.subplot(122)
power_passengers.plot(color="tab:red", title="Power transformed Values", ax=ax2);
```



From the above the first chart, we can see that we are able to remove the trend from time-series data.

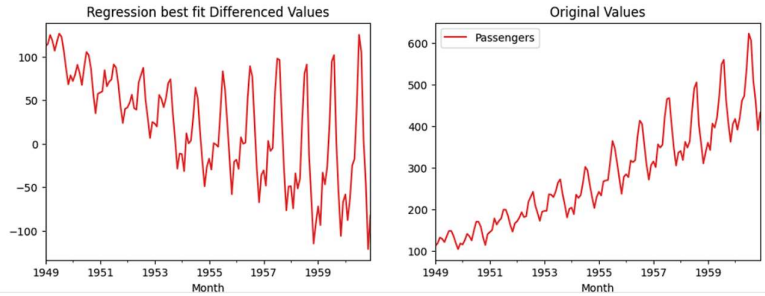
```
# perform seasonal decompose on logged transformed

decompose_result = seasonal_decompose(passengers_pow_rolled_detrended.dropna(), model='additive', filt=None, period=None, two_sided=True, extrapolate_trend=0)

# perform seasonal decomposing using additive model type

trend = decompose_result.trend # get trend
seasonal = decompose_result.seasonal #get seasonal
residual = decompose_result.resid #get residual

#plot the decompose result
# decompose_result.plot("Multiplicative Decomposition")

plt.subplot(411)
plt.plot(passengers_pow_rolled_detrended, label='Original')
plt.legend(loc='best')

plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')

plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')

plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```



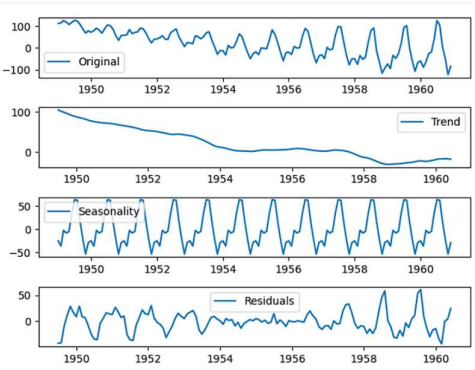
Applying Linear Regression to Remove Trend

We can also apply a linear regression model to remove the trend. Below we are fitting a linear regression model to our time-series data. We are then using a fit model to predict time-series values from beginning to end. We are then subtracting predicted values from original time-series to remove the trend.

```
from statsmodels.regression.linear_model import OLS

least_squares = OLS(air_passengers["Passengers"].values, list(range(air_passengers.shape[0])))
result = least_squares.fit()

fit = pd.Series(result.predict(list(range(air_passengers.shape[0]))), index = air_passengers.index)

passengers_ols_detrended = air_passengers["Passengers"] - fit


# plot the reggressed model
ax1 = plt.subplot(121)
passengers_ols_detrended.plot(figsize=(12,4) ,color="tab:red", title="Regression best fit Differenced Values", ax=ax1);
ax2 = plt.subplot(122)
air_passengers.plot(color="tab:red", title="Original Values", ax=ax2);
```

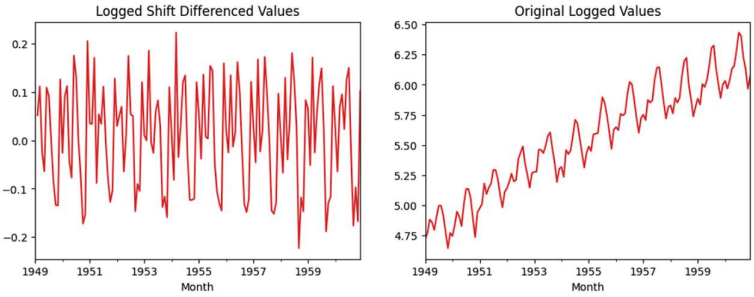



```
# perform seasonal decompose on logged transformed

decompose_result = seasonal_decompose(passengers_ols_detrended, model='additive', filt=None, period=None, two_sided=True, extrapolate_trend=0)

# perform seasonal decomposing using additive model type

trend = decompose_result.trend # get trend
seasonal = decompose_result.seasonal #get seasonal
residual = decompose_result.resid #get residual

#plot the decompose result
# decompose_result.plot("Multiplicative Decomposition")

plt.subplot(411)
plt.plot(passengers_ols_detrended, label='Original')
plt.legend(loc='best')

plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')

plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')

plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```



Differencing Over Log Transformed Time-Series We have applied differencing to log-transformed time-series by shifting its value by 1 period and subtracting it from original log-transformed time-series

```
[ ]  1 logged_passengers_diff = logged_passengers - logged_passengers.shift()
     2
     3 # plot the logged_passengers_diff
     4 ax1 = plt.subplot(121)
     5 logged_passengers_diff.plot(figsize=(12,4) ,color="tab:red", title="Logged Shift Differenced Values", ax=ax1);
     6 ax2 = plt.subplot(122)
     7 logged_passengers.plot(color="tab:red", title="Original Logged Values", ax=ax2);
```




```
[ ]  1 # perform the ADF test
     2 from statsmodels.tsa.stattools import adfuller
     3
     4 dftest = adfuller(logged_passengers_diff.dropna(), autolag = 'AIC')
     5
     6 print("1. ADF : ",dftest[0])
     7 print("2. P-Value : ", dftest[1])
     8 print("3. Num Of Lags : ", dftest[2])
     9 print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
    10 print("5. Critical Values :")
    11 for key, val in dftest[4].items():
    12     print("\t",key, ": ", val)
```

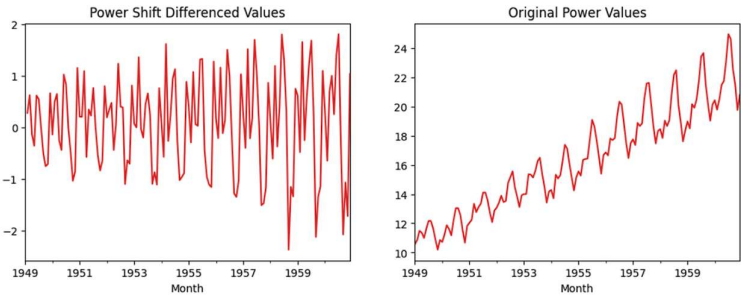
```
1. ADF :  -2.717130598388114
2. P-Value :  0.07112054815086184
3. Num Of Lags :  14
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 128
5. Critical Values :
        1% :  -3.48250069398879 97
        5% :  -2.884397984161377
        10% :  -2.578960197753906
```

From our dicky-fuller test results, we can confirm that time-series is NOT STATIONARY due to the p-value of 0.07 greater than 0.05.

Differencing Over Power Transformed Time-Series

We have applied differencing to power transformed time-series by shifting its value by 1 period and subtracting it from original power transformed time-series

```
[ ]  1 powered_passengers_diff = power_passengers - power_passengers.shift()
     2
     3 # plot the logged_passengers_diff
     4 ax1 = plt.subplot(121)
     5 powered_passengers_diff.plot(figsize=(12,4) ,color="tab:red", title="Power Shift Differenced Values", ax=ax1);
     6 ax2 = plt.subplot(122)
     7 power_passengers.plot(color="tab:red", title="Original Power Values", ax=ax2);
```



```
# perform the ADF test
from statsmodels.tsa.stattools import adfuller

dftest = adfuller(powered_passengers_diff.dropna(), autolag = 'AIC')

print("1. ADF : ",dftest[0])
print("2. P-Value : ", dftest[1])
print("3. Num Of Lags : ", dftest[2])
print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
print("5. Critical Values :")
for key, val in dftest[4].items():
    print("\t",key, ": ", val)
```

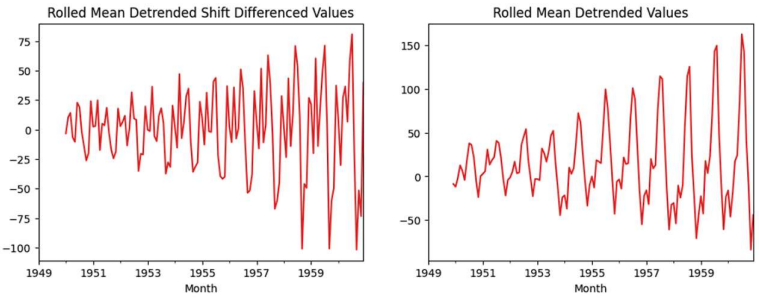
```
1. ADF :  -3.1864222911641904
2. P-Value :  0.020784185571268328
3. Num Of Lags :  12
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 130
5. Critical Values :
	1% :  -3.4816817173418295
	5% :  -2.884041834319526 7
	10% :  -2.578770059171598
```

From our dicky-fuller test results, we can confirm that time-series is STATIONARY due to a p-value of 0.02 less than 0.05.

Differencing Over Time-Series with Rolling Mean taken over 12 Months

We have applied differencing to mean rolled time-series by shifting its value by 1 period and subtracting it from original mean rolled time-series

```
# plot and calculate the rolled detrended diff
passengers_rolled_detrended_diff = passengers_rolled_detrended - passengers_rolled_detrended.shift()

# plot the logged_passengers_diff
ax1 = plt.subplot(121)
passengers_rolled_detrended_diff.plot(figsize=(12,4) ,color="tab:red", title="Rolled Mean Detrended Shift Differenced Values", ax=ax1);
ax2 = plt.subplot(122)
passengers_rolled_detrended.plot(color="tab:red", title="Rolled Mean Detrended Values", ax=ax2);
```

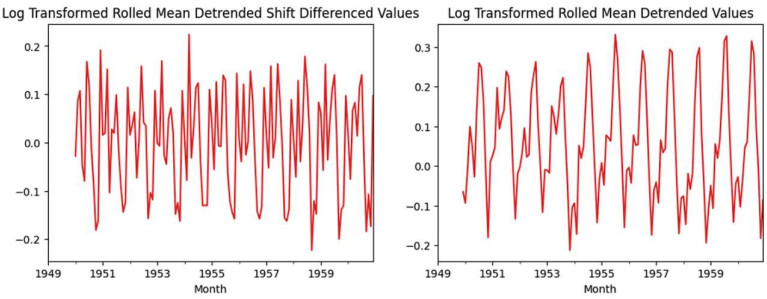



```
# perform the ADF test
from statsmodels.tsa.stattools import adfuller

dftest = adfuller(passengers_rolled_detrended_diff.dropna(), autolag = 'AIC')

print("1. ADF : ",dftest[0])
print("2. P-Value : ", dftest[1])
print("3. Num Of Lags : ", dftest[2])
print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
print("5. Critical Values :")
for key, val in dftest[4].items():
    print("\t",key, ": ", val)
```

```
1. ADF :  -3.1544826348635753
2. P-Value :  0.022775264967859258
3. Num Of Lags :  12
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 119
5. Critical Values :
	1% :  -3.4865346059036564
	5% :  -2.8861509858476264
	10% :  -2.579896092790057
```

From our dicky-fuller test results, we can confirm that time-series is STATIONARY due to a p-value of 0.02 less than 0.05.

```
# diffrence the log rolled detrended diff and plot
passengers_log_rolled_detrended_diff = passengers_log_rolled_detrended - passengers_log_rolled_detrended.shift()

# plot the logged_passengers_diff
ax1 = plt.subplot(121)
passengers_log_rolled_detrended_diff.plot(figsize=(12,4) ,color="tab:red", title="Log Transformed Rolled Mean Detrended Shift Differenced Values", ax=ax1);
ax2 = plt.subplot(122)
passengers_log_rolled_detrended.plot(color="tab:red", title="Log Transformed Rolled Mean Detrended Values", ax=ax2);
```



```
[ ]  1 # perform the ADF test
     2 from statsmodels.tsa.stattools import adfuller
     3
     4 dftest = adfuller(passengers_log_rolled_detrended_diff.dropna(), autolag = 'AIC')
     5
     6 print("1. ADF : ",dftest[0])
     7 print("2. P-Value : ", dftest[1])
     8 print("3. Num Of Lags : ", dftest[2])
     9 print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
    10 print("5. Critical Values :")
    11 for key, val in dftest[4].items():
    12     print("\t",key, ": ", val)
```

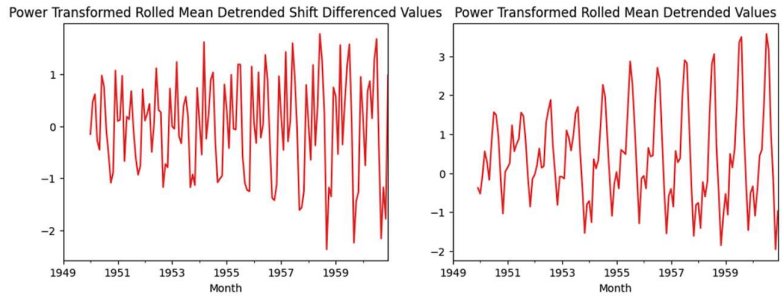
```
1. ADF :  -3.9129812454195174
2. P-Value :  0.0019413623769364548
3. Num Of Lags :  13
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 118
5. Critical Values :
        1% :  -3.4870216863700767
        5% :  -2.8863625166643136
        10% :  -2.580009026141913
```

From our dicky-fuller test results, we can confirm that time-series is STATIONARY due to a p-value of 0.001 less than 0.05.

Differencing Over Power Transformed & Mean Rolled Time-Series

We have applied differencing to power transformed & mean rolled time-series by shifting its value by 1 period and subtracting it from original time-series

```
[ ]  1 # diffrence the power rolled detrended diff and plot
     2 passengers_pow_rolled_detrended_diff = passengers_pow_rolled_detrended - passengers_pow_rolled_detrended.shift()
     3
     4 # plot the power_passengers_diff
     5 ax1 = plt.subplot(121)
     6 passengers_pow_rolled_detrended_diff.plot(figsize=(12,4) ,color="tab:red", title="Power Transformed Rolled Mean Detrended Shift Differenced Values", ax=ax1);
     7 ax2 = plt.subplot(122)
     8 passengers_pow_rolled_detrended.plot(color="tab:red", title="Power Transformed Rolled Mean Detrended Values", ax=ax2);
```




```
[ ]  1 # perform the ADF test
     2 from statsmodels.tsa.stattools import adfuller
     3
     4 dftest = adfuller(passengers_pow_rolled_detrended_diff.dropna(), autolag = 'AIC')
     5
     6 print("1. ADF : ",dftest[0])
     7 print("2. P-Value : ", dftest[1])
     8 print("3. Num Of Lags : ", dftest[2])
     9 print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
    10 print("5. Critical Values :")
    11 for key, val in dftest[4].items():
    12     print("\t",key, ": ", val)
```

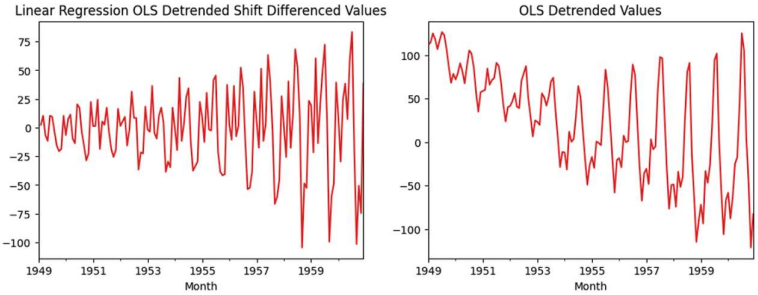
```
1. ADF :  -3.622485336190725
2. P-Value :  0.005345423958262613
3. Num Of Lags :  13
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 118
5. Critical Values :
        1% :  -3.4870216863700767
        5% :  -2.8863625166643136
        10% :  -2.580009026141913
```

From our dicky-fuller test results, we can confirm that time-series is STATIONARY due to a p-value of 0.005 less than 0.05.

Differencing Over Linear Regression Transformed Time-Series

We have applied differencing to linear regression transformed time-series by shifting it's value by 1 period and subtracting it from original log-transformed time-series

```
[ ]  1 # take the differrence of linear regressed transformed with a shift and then plot
     2 passengers_ols_detrended_diff = passengers_ols_detrended - passengers_ols_detrended.shift()
     3
     4 # plot the power_passengers_diff
     5 ax1 = plt.subplot(121)
     6 passengers_ols_detrended_diff.plot(figsize=(12,4) ,color="tab:red", title="Linear Regression OLS Detrended Shift Differenced Values", ax=ax1);
     7 ax2 = plt.subplot(122)
     8 passengers_ols_detrended.plot(color="tab:red", title="OLS Detrended Values", ax=ax2);
```



```
[ ]  1 # perform ADF test
     2 # perform the ADF test
     3 from statsmodels.tsa.stattools import adfuller
     4
     5 dftest = adfuller(passengers_ols_detrended_diff.dropna(), autolag = 'AIC')
     6
     7 print("1. ADF : ",dftest[0])
     8 print("2. P-Value : ", dftest[1])
     9 print("3. Num Of Lags : ", dftest[2])
    10 print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
    11 print("5. Critical Values :")
    12 for key, val in dftest[4].items():
    13     print("\t",key, ": ", val)
```

```
1. ADF :  -2.829266824170043
2. P-Value :  0.05421329028382486
3. Num Of Lags :  12
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 130
5. Critical Values :
        1% :  -3.4816817173418295
        5% :  -2.8840418343195267
        10% :  -2.578770059171598
```

From our dicky-fuller test results, we can confirm that time-series is NOT STATIONARY due to the p-value of 0.054 greater than 0.05.

RESULT

The implementation was successful.

Experiment 4

AIM

To Remove Seasonality (Differencing Over Log Transformed Time-Series, Differencing Over Power Transformed Time-Series, Differencing Over Time-Series with Rolling Mean taken over 12 Months, Differencing Over Power Transformed & Mean Rolled Time-Series, Differencing Over Linear Regression Transformed Time-Series),), Dicky-Fuller Test for Stationarity.

SOFTWARE USED

Google Colaboratory

CODE AND OUTPUT

Hitesh_Lab4_August_17_2023.ipynb

File Edit View Insert Runtime Tools Help Last edited on August 27

Comment Share

+ Code + Text

Connect

TSA LAB 4: ADF and KPSS Test and Implement AR Model - Google Stock Price Data

```
[ ]  1 import pandas as pd
     2 import numpy as np
     3 import matplotlib.pyplot as plt
     4 import statsmodels.api as sm
     5 from statsmodels.tsa.seasonal import seasonal_decompose
     6 from statsmodels.tsa.stattools import adfuller
     7 from statsmodels.tsa.stattools import kpss
     8 from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
     9 from sklearn.linear_model import LinearRegression
    10 from sklearn.metrics import mean_squared_error
    11 from statsmodels.tsa.arima.model import ARIMA
```

```
[ ]  1 %matplotlib inline
```

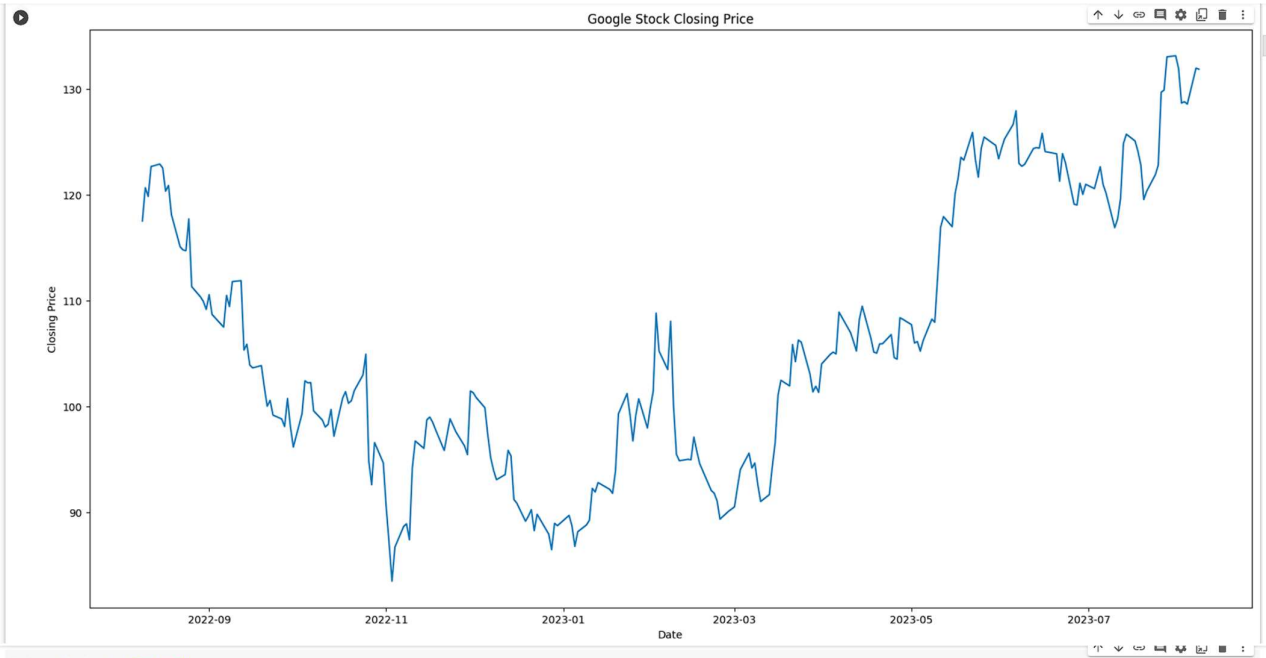
```
[ ]  1 file_path = "/content/GOOG.csv"
```

```
[ ]  1 df = pd.read_csv(file_path, index_col=0, parse_dates=True)
```

```
[ ]  1 df.head()
```

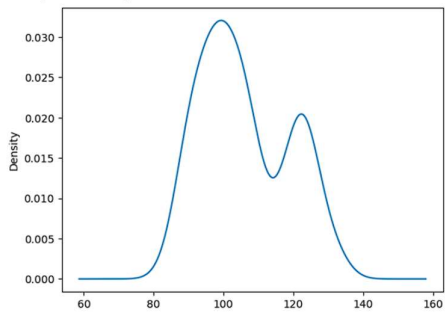
Date	Open	High	Low	Close	Adj Close	Volume
2022-08-09	117.989998	118.199997	116.559998	117.500000	117.500000	15424300
2022-08-10	119.589996	121.779999	119.360001	120.650002	120.650002	20497000
2022-08-11	122.080002	122.339996	119.550003	119.820000	119.820000	16671600
2022-08-12	121.160004	122.650002	120.400002	122.650002	122.650002	16121100
2022-08-15	122.209999	123.260002	121.570000	122.879997	122.879997	15525000

```
[ ]  1 # Plot the closing price
     2
     3 plt.rc("font", size=10)
     4
     5 plt.figure(figsize=(20, 10))
     6 plt.plot(df['Close'])
     7 plt.title('Google Stock Closing Price')
     8 plt.xlabel('Date')
     9 plt.ylabel('Closing Price')
    10 plt.show()
```




```
[ ]  1 df_close = df['Close']
     2 df_close.plot(kind='kde')
```

```
<Axes: ylabel='Density'>
```



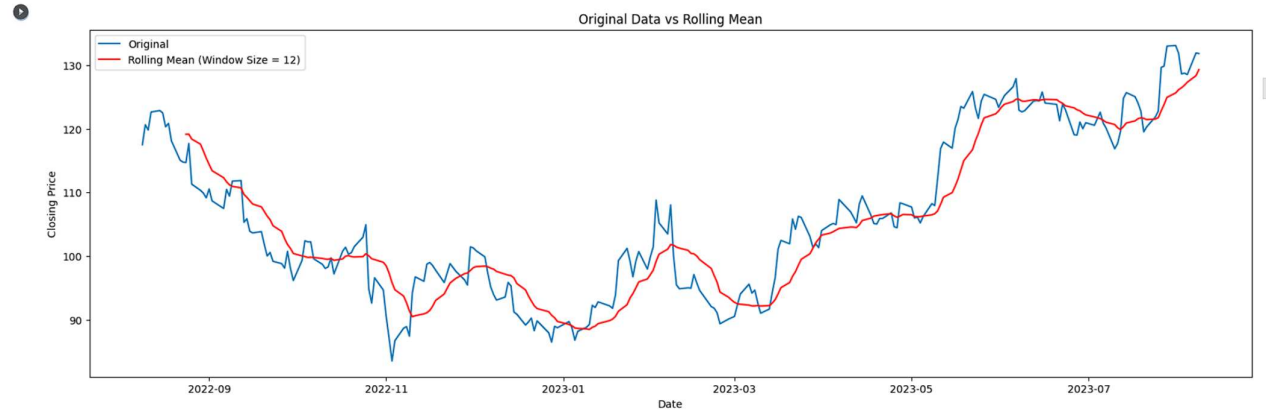

```
[ ]  1 # Split the data into 3 parts
     2 num_splits = 3
     3 split_size = len(df) // num_splits
     4 data_splits = [df.iloc[i * split_size: (i + 1) * split_size] for i in range(num_splits)]
     5
```

```
[ ]  1 # Calculate the mean and variance for each split
     2 for i, data_split in enumerate(data_splits):
     3     mean = np.mean(data_split['Close'])
     4     variance = np.var(data_split['Close'])
     5     print(f"Split {i + 1} - Mean: {mean:.2f}, Variance: {variance:.2f}")
     6
```

```
Split 1 - Mean: 102.91, Variance: 82.40
Split 2 - Mean: 95.75, Variance: 33.28
Split 3 - Mean: 118.68, Variance: 67.52
```

```
[ ]  1 # Calculate the rolling mean and plot the graph
     2 rolling_mean = df['Close'].rolling(window=12).mean()
```

```
     1 plt.figure(figsize=(20, 6))
     2 plt.plot(df['Close'], label='Original')
     3 plt.plot(rolling_mean, label='Rolling Mean (Window Size = 12)', color='red')
     4 plt.legend()
     5 plt.title('Original Data vs Rolling Mean')
     6 plt.xlabel('Date')
     7 plt.ylabel('Closing Price')
     8 plt.show()
```

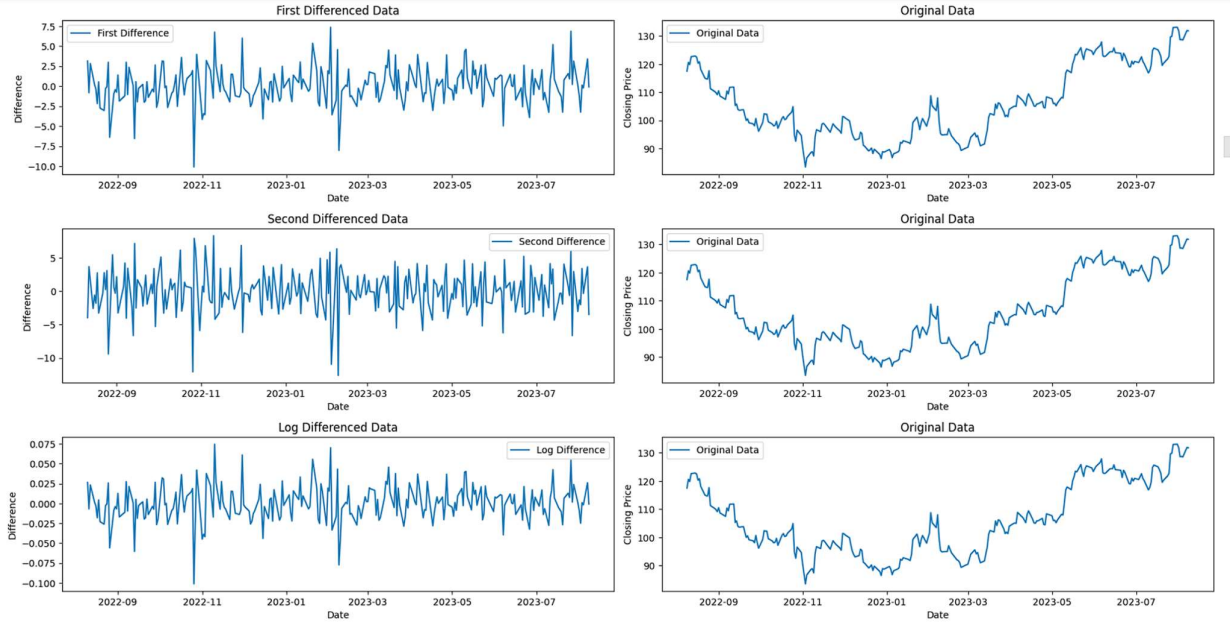


```
X = df["Close"].values
result = adfuller(X)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))

if result[0] < result[4]["5%"]:
    print ("Reject Ho - Time Series is Stationary")
else:
    print ("Failed to Reject Ho - Time Series is Non-Stationary")
```

```
ADF Statistic: -0.893703
p-value: 0.790039
Critical Values:
	1%: -3.457
	5%: -2.873
	10%: -2.573
Failed to Reject Ho - Time Series is Non-Stationary
```

```
# Apply differencing methods
df_diff1 = df['Close'].diff().dropna()  # First difference

df_diff2 = df['Close'].diff().diff().dropna()  # Second difference

df_log_diff = df['Close'].apply(np.log).diff().dropna()  # Log difference
```

```
# Plot the differenced data
plt.figure(figsize=(18, 12))


plt.subplot(421)
plt.plot(df_diff1, label='First Difference')
plt.legend(loc='best')
plt.title('First Differenced Data')
plt.xlabel('Date')
plt.ylabel('Difference')


plt.subplot(422)
plt.plot(df['Close'], label='Original Data')
plt.legend(loc='best')
plt.title('Original Data')
plt.xlabel('Date')
plt.ylabel('Closing Price')


plt.subplot(423)
plt.plot(df_diff2, label='Second Difference')
plt.legend(loc='best')
plt.title('Second Differenced Data')
plt.xlabel('Date')
plt.ylabel('Difference')


plt.subplot(424)
plt.plot(df['Close'], label='Original Data')
plt.legend(loc='best')
plt.title('Original Data')
plt.xlabel('Date')
plt.ylabel('Closing Price')


plt.subplot(425)
plt.plot(df_log_diff, label='Log Difference')
plt.legend(loc='best')
plt.title('Log Differenced Data')
plt.xlabel('Date')
plt.ylabel('Difference')


plt.subplot(426)
plt.plot(df['Close'], label='Original Data')
plt.legend(loc='best')
plt.title('Original Data')
plt.xlabel('Date')
plt.ylabel('Closing Price')


plt.tight_layout()
```



```python
# ADF and KPSS tests for stationarity
def test_stationarity(timeseries):
    # ADF test

    adf_test = adfuller(timeseries, autolag='AIC')
    print("ADF Test Results:")
    print(f'ADF Test Statistic: {adf_test[0]}')
    print(f'p-value: {adf_test[1]}')
    # print(f'Critical Values: {adf_test[4]}')

    # result = adfuller(timeseries, autolag='AIC')
    # print('ADF Statistic: %f' % result[0])
    # print('p-value: %f' % result[1])
    print('Critical Values:')
    for key, value in adf_test[4].items():
        print(f'\t{key}: {value}')

    if adf_test[0] < adf_test[4]["5%"]:
        print ("Reject Ho - Time Series is Stationary")
    else:
        print ("Failed to Reject Ho - Time Series is Non-Stationary")


    # KPSS test
    kpss_test = kpss(timeseries, nlags='auto')
    print("\nKPSS Test Results:")
    print(f'KPSS Test Statistic: {kpss_test[0]}')
    print(f'p-value: {kpss_test[1]}')
    print(f'Critical Values: {kpss_test[3]}')
```

```python
print("\nFirst Difference:")
test_stationarity(df_diff1)
```

```
First Difference:
ADF Test Results:
ADF Test Statistic: -11.971825846310379
p-value: 3.882320630634504e-22
Critical Values:
	1%: -3.4569962781990573
	5%: -2.8732659015936024
	10%: -2.573018897632674
Reject Ho - Time Series is Stationary

KPSS Test Results:
KPSS Test Statistic: 0.3514938486705948
p-value: 0.09806299626267465
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
```

```python
print("\nSecond Difference:")
test_stationarity(df_diff2)
```

```
Second Difference:
ADF Test Results:
ADF Test Statistic: -8.353684562399444
p-value: 2.94802700957099e-13
Critical Values:
	1%: -3.4582467982399105
	5%: -2.8738137461081323
	10%: -2.5733111490323846
Reject Ho - Time Series is Stationary

KPSS Test Results:
KPSS Test Statistic: 0.13153242844233876
p-value: 0.1
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
<ipython-input-20-92d4b23f3d2b>:25: InterpolationWarning: The test statistic is outside of the range of p-values available in the
look-up table. The actual p-value is greater than the p-value returned.

  kpss_test = kpss(timeseries, nlags='auto')
```

```python
print("\nLog Difference:")
test_stationarity(df_log_diff)
```

```
Log Difference:
ADF Test Results:
ADF Test Statistic: -12.110976964109183
p-value: 1.9197212216094285e-22
Critical Values:
	1%: -3.4569962781990573
	5%: -2.8732659015936024
	10%: -2.573018897632674
Reject Ho - Time Series is Stationary

KPSS Test Results:
KPSS Test Statistic: 0.13381152797403866
p-value: 0.1
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
<ipython-input-20-92d4b23f3d2b>:25: InterpolationWarning: The test statistic is outside of the range of p-values available in the
look-up table. The actual p-value is greater than the p-value returned.

  kpss_test = kpss(timeseries, nlags='auto')
```

Auto Regressive Model

$$Y_t = \alpha + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \ldots + \beta_p Y_{t-p} + \epsilon_1$$

```python
df
```

Date	Open	High	Low	Close	Adj Close	Volume
2022-08-09	117.989998	118.199997	116.559998	117.500000	117.500000	15424300
2022-08-10	119.589996	121.779999	119.360001	120.650002	120.650002	20497000
2022-08-11	122.080002	122.339996	119.550003	119.820000	119.820000	16671600
2022-08-12	121.160004	122.650002	120.400002	122.650002	122.650002	16121100
2022-08-15	122.209999	123.260002	121.570000	122.879997	122.879997	15525000
...	...	...	...	...	...	...
2023-08-02	129.839996	130.419998	127.849998	128.639999	128.639999	22705800
2023-08-03	128.369995	129.770004	127.775002	128.770004	128.770004	15018100
2023-08-04	129.600006	131.929993	128.315002	128.539993	128.539993	20509500
2023-08-07	129.509995	132.059998	129.429993	131.940002	131.940002	17621000
2023-08-08	130.979996	131.940002	130.130005	131.839996	131.839996	16828300

251 rows × 6 columns

```python
sdi = df.asfreq('D')
sdi = sdi[['Close']]
sdi
```

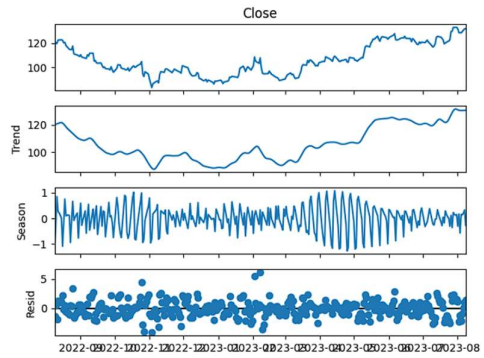
Date	Close
2022-08-09	117.500000
2022-08-10	120.650002
2022-08-11	119.820000
2022-08-12	122.650002
2022-08-13	NaN
...	...
2023-08-04	128.539993
2023-08-05	NaN
2023-08-06	NaN
2023-08-07	131.940002
2023-08-08	131.839996

365 rows × 1 columns

```
[ ] 1 from statsmodels.tsa.seasonal import STL
```

```
[ ] 1 # in this place "use linear interpolation here" use the commands for linear interpolation.
    2 # sdi = Seasonal Decomposition and Interpolation to interpolate missing values in the daily frequency of data
    3
    4 res = STL(sdi['Close'].interpolate(method="linear"), seasonal=13).fit()
```

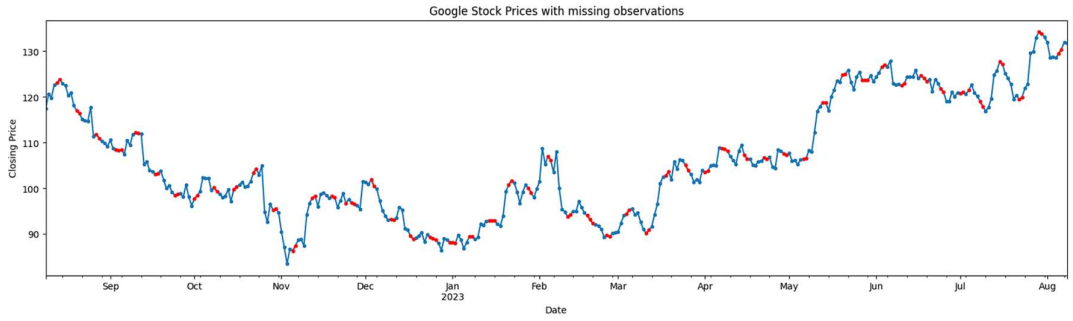
```
[ ] 1 res.plot()
```




```
[ ] 1 seasonal_component = res.seasonal
    2 seasonal_component.head()
```

```
Date
2022-08-09   -1.224121
2022-08-10   -0.260225
2022-08-11    0.852518
2022-08-12    0.272024
2022-08-13    0.208928
Freq: D, Name: season, dtype: float64
```

```
1 sdi_deseasonalised = sdi['Close'] - seasonal_component
2 sdi_deseasonalised_imputed = sdi_deseasonalised.interpolate(method="spline",order=3)
3 sdi_imputed = sdi_deseasonalised_imputed + seasonal_component
4
5 sdi_imputed = sdi_imputed.to_frame().rename(columns={0: 'Close'})
6
7 # plot
8 ax = sdi_imputed.plot(linestyle="-", marker=".", figsize=[20, 5], legend=None)
9 ax = sdi_imputed[sdi.isnull()].plot(ax=ax, legend=None, marker=".", color="r")
10 ax.set_title('Google Stock Prices with missing observations')
11 ax.set_ylabel('Closing Price')
12 ax.set_xlabel('Date')
```

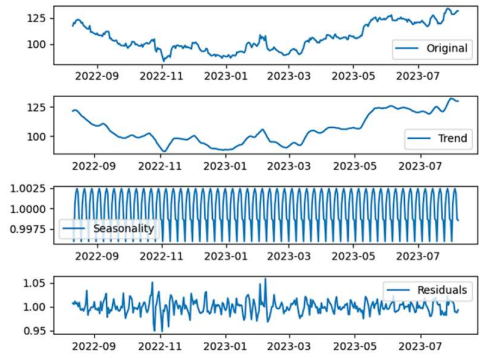



```
1 sdi_imputed
```

Date	Close
2022-08-09	117.500000
2022-08-10	120.650002
2022-08-11	119.820000
2022-08-12	122.650002
2022-08-13	123.090103
...	...
2023-08-04	128.539993
2023-08-05	129.406770
2023-08-06	130.415954
2023-08-07	131.940002
2023-08-08	131.839996

365 rows × 1 columns

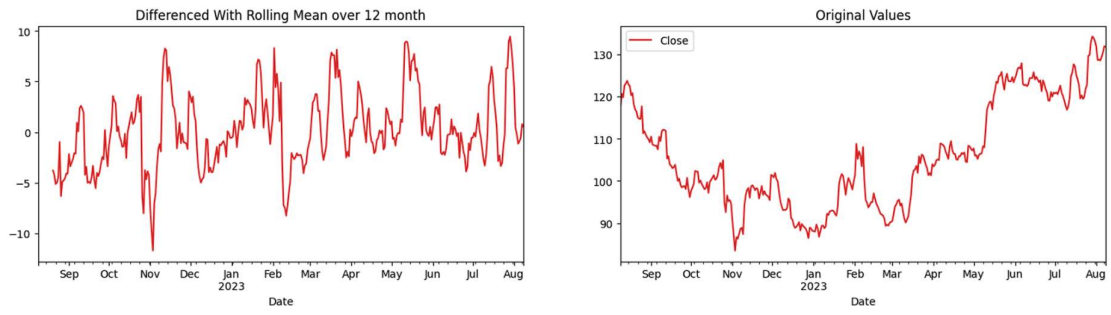
```
# perform seasonal decompose on logged transformed

decompose_result = seasonal_decompose(sdi_imputed, model='multiplicative', filt=None, period=None, two_sided=True, extrapolate_trend=0)

# perform seasonal decomposing using additive model type

trend = decompose_result.trend # get trend
seasonal = decompose_result.seasonal #get seasonal
residual = decompose_result.resid #get residual

#plot the decompose result
# decompose_result.plot("Multiplicative Decomposition")

plt.subplot(411)
plt.plot(sdi_imputed, label='Original')
plt.legend(loc='best')

plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')

plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')

plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```




```
# perform the ADF test
from statsmodels.tsa.stattools import adfuller

dftest = adfuller(sdi_imputed, autolag = 'AIC')

print("1. ADF : ",dftest[0])
print("2. P-Value : ", dftest[1])
print("3. Num Of Lags : ", dftest[2])
print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
print("5. Critical Values :")
for key, val in dftest[4].items():
    print("\t",key, ": ", val)
```

```
1. ADF :  -1.0295989868277595
2. P-Value :  0.7423162782004037
3. Num Of Lags :  4
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 360
5. Critical Values :
	1% :  -3.448645946352023
	5% :  -2.869602139060357
	10% :  -2.5710650077160495
```

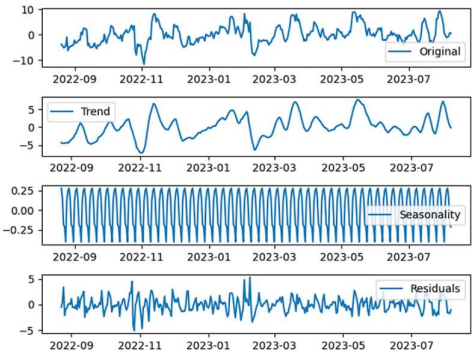
```
rolling_mean = sdi_imputed.rolling(window = 12).mean()
sdi_rolled_detrended = sdi_imputed['Close'] - rolling_mean['Close']


ax1 = plt.subplot(121)
sdi_rolled_detrended.plot(figsize=(18,4),color="tab:red", title="Differenced With Rolling Mean over 12 month", ax=ax1);
ax2 = plt.subplot(122)
sdi_imputed.plot(figsize=(18,4), color="tab:red", title="Original Values", ax=ax2);
```



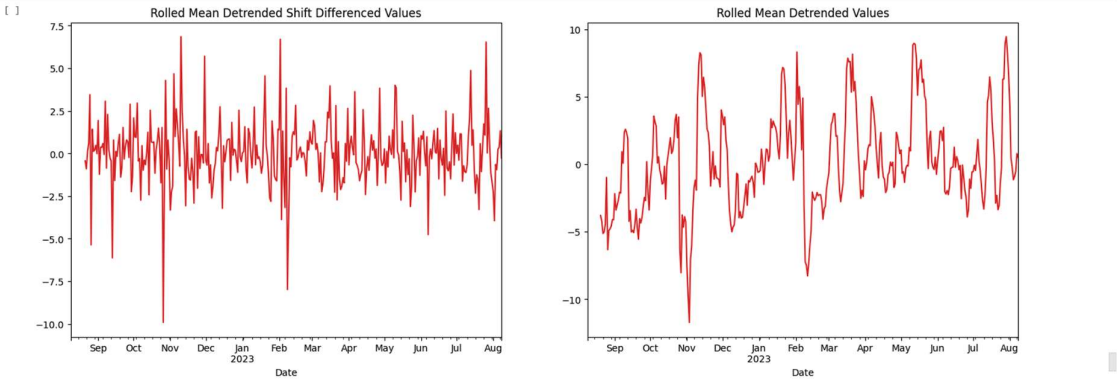
```
[ ]  1 sdi_rolled_detrended
```

```
Date
2022-08-09         NaN
2022-08-10         NaN
2022-08-11         NaN
2022-08-12         NaN
2022-08-13         NaN
                   ...
2023-08-04   -1.153109
2023-08-05   -0.913563
2023-08-06   -0.539876
2023-08-07    0.794173
2023-08-08    0.530000
Freq: D, Name: Close, Length: 365, dtype: float64
```

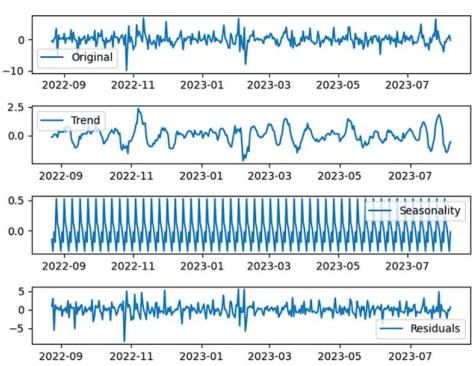
```
 1 decompose_result = seasonal_decompose(sdi_rolled_detrended.dropna(), model='additive', filt=None, period=None, two_sided=True, extrapolate_trend=0)
 2
 3 # perform seasonal decomposing using multiplicative model type
 4
 5 trend = decompose_result.trend # get trend
 6 seasonal = decompose_result.seasonal #get seasonal
 7 residual = decompose_result.resid #get residual
 8
 9 #plot the decompose result
10 # decompose_result.plot("Multiplicative Decomposition")
11
12 plt.subplot(411)
13 plt.plot(sdi_rolled_detrended, label='Original')
14 plt.legend(loc='best')
15
16 plt.subplot(412)
17 plt.plot(trend, label='Trend')
18 plt.legend(loc='best')
19
20 plt.subplot(413)
21 plt.plot(seasonal,label='Seasonality')
22 plt.legend(loc='best')
23
24 plt.subplot(414)
25 plt.plot(residual, label='Residuals')
26 plt.legend(loc='best')
27 plt.tight_layout()
```




```
 1 sdi_rolled_detrended_diff = sdi_rolled_detrended - sdi_rolled_detrended.shift()
 2
 3 # plot the logged_passengers_diff
 4 ax1 = plt.subplot(121)
 5 sdi_rolled_detrended_diff.plot(figsize=(18,6) ,color="tab:red", title="Rolled Mean Detrended Shift Differenced Values", ax=ax1);
 6 ax2 = plt.subplot(122)
 7 sdi_rolled_detrended.plot(color="tab:red", title="Rolled Mean Detrended Values", ax=ax2);
```

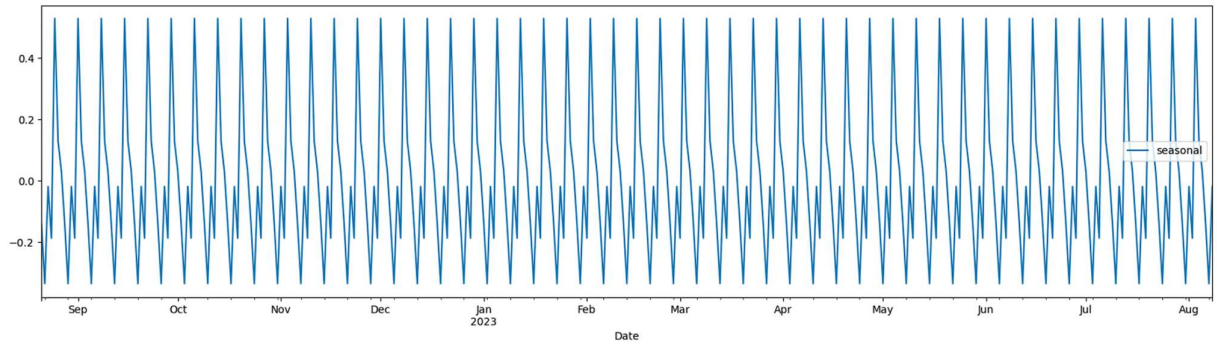



```
 1 decompose_result = seasonal_decompose(sdi_rolled_detrended_diff.dropna(), model='additive', filt=None, period=None, two_sided=True, extrapolate_trend=0)
 2
 3 # perform seasonal decomposing using multiplicative model type
 4
 5 trend = decompose_result.trend # get trend
 6 seasonal = decompose_result.seasonal #get seasonal
 7 residual = decompose_result.resid #get residual
 8
 9 #plot the decompose result
10 # decompose_result.plot("Multiplicative Decomposition")
11
12 plt.subplot(411)
13 plt.plot(sdi_rolled_detrended_diff, label='Original')
14 plt.legend(loc='best')
15
16 plt.subplot(412)
17 plt.plot(trend, label='Trend')
18 plt.legend(loc='best')
19
20 plt.subplot(413)
21 plt.plot(seasonal,label='Seasonality')
22 plt.legend(loc='best')
23
24 plt.subplot(414)
25 plt.plot(residual, label='Residuals')
26 plt.legend(loc='best')
27 plt.tight_layout()
```




```
[ ]  1 plt.figure(figsize=(20,5))
     2 seasonal.plot()
     3 plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7965a46f0fd0>
```




```
[ ]  1 # perform the ADF test
     2 from statsmodels.tsa.stattools import adfuller
     3
     4 dftest = adfuller(sdi_rolled_detrended_diff.dropna(), autolag = 'AIC')
     5
     6 print("1. ADF : ",dftest[0])
     7 print("2. P-Value : ", dftest[1])
     8 print("3. Num Of Lags : ", dftest[2])
     9 print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
    10 print("5. Critical Values :")
    11 for key, val in dftest[4].items():
    12     print("\t",key, ": ", val)
```

```
1. ADF :  -9.207865120768997
2. P-Value :  1.9230229748700873e-15
3. Num Of Lags :  9
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 343
5. Critical Values :
         1% :  -3.449559661646851
         5% :  -2.8700035112469626
         10% :  -2.5712790073013796
```

AR Modelling

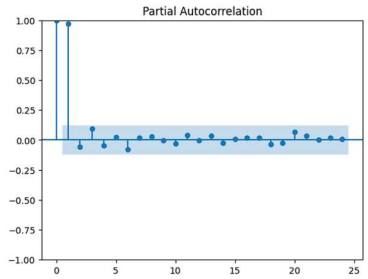
```
[ ]  1 # df = pd.DataFrame(sdi_rolled_detrended_diff)
```

```
[ ]  1 # X = sdi_rolled_detrended_diff.dropna()
```

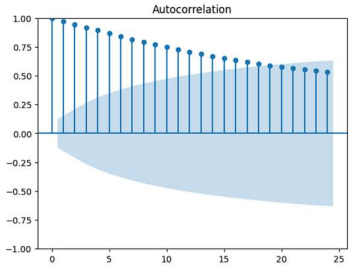
```
[ ]  1 X = df.Close
```

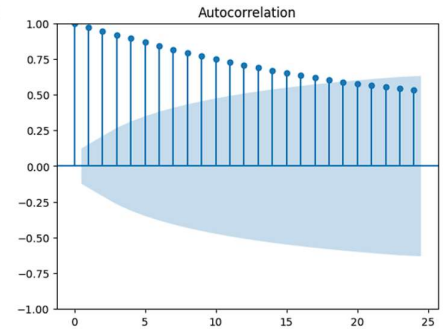
+ Code + Text

```
[ ]  1 plot_pacf(X);
```




```
[ ]  1 plot_acf(X)
```






```
df["Close_shifted"]  = df["Close"].shift()
```

```
df.head()
```

Date	Open	High	Low	Close	Adj Close	Volume	Close_shifted
2022-08-09	117.989998	118.199997	116.559998	117.500000	117.500000	15424300	NaN
2022-08-10	119.589996	121.779999	119.360001	120.650002	120.650002	20497000	117.500000
2022-08-11	122.080002	122.339996	119.550003	119.820000	119.820000	16671600	120.650002
2022-08-12	121.160004	122.650002	120.400002	122.650002	122.650002	16121100	119.820000
2022-08-15	122.209999	123.260002	121.570000	122.879997	122.879997	15525000	122.650002

```
columns_to_remove = df.columns.drop(['Close','Close_shifted'])
```

```
df.drop(columns_to_remove, axis = 1, inplace=True)
```

+ Code + Text

```
df.dropna(inplace=True)
```

+ Code + Text

```
df.head()
```

Date	Close	Close_shifted
2022-08-10	120.650002	117.500000
2022-08-11	119.820000	120.650002
2022-08-12	122.650002	119.820000
2022-08-15	122.879997	122.650002
2022-08-16	122.510002	122.879997

```
y = df.Close.values
X = df.Close_shifted.values
```

```
train_size = int(len(X) * 0.80)
```

```
X_train, X_test = X[0:train_size], X[train_size:len(X)]
y_train, y_test = y[0:train_size], y[train_size:len(X)]
```

```
X_train = X_train.reshape(-1,1)
X_test = X_test.reshape(-1,1)
```

```
lr = LinearRegression()
lr.fit(X_train, y_train)
```

```
LinearRegression()
```

```
lr.coef_
```

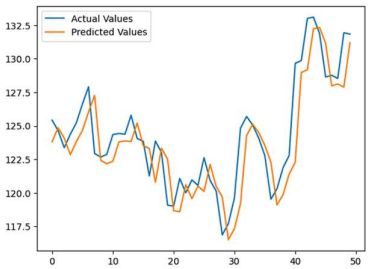
```
array([0.97480486])
```

```
lr.intercept_
```

```
2.5871238391784317
```

```
y_pred = lr.predict(X_test)
```

```
plt.plot(y_test, label="Actual Values")
plt.plot(y_pred, label="Predicted Values")
plt.legend()
plt.show()
```




```
model = ARIMA(y_train, order=(3,0,0))
```

```
model_fit = model.fit()
```

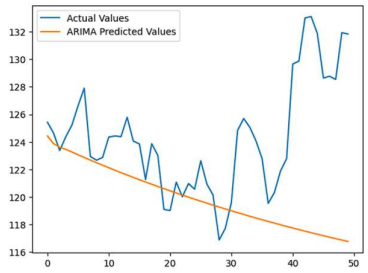
```
print(model_fit.summary())
```

```
                               SARIMAX Results
==========================================================================
Dep. Variable:                      y   No. Observations:                  200
Model:                 ARIMA(3, 0, 0)   Log Likelihood                -456.234
Date:                Sun, 27 Aug 2023   AIC                            922.468
Time:                        09:45:20   BIC                            938.960
Sample:                             0   HQIC                           929.142
                                  - 200
Covariance Type:                  opg
==========================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------
const        109.8846     11.293      9.730      0.000      87.751     132.018
ar.L1          1.0238      0.068     15.117      0.000       0.891       1.156
ar.L2         -0.1622      0.102     -1.587      0.113      -0.362       0.038
ar.L3          0.1225      0.077      1.592      0.111      -0.028       0.273
sigma2         5.5150      0.411     13.427      0.000       4.710       6.320
===================================================================================
Ljung-Box (L1) (Q):                   0.02   Jarque-Bera (JB):                40.41
Prob(Q):                              0.90   Prob(JB):                         0.00
Heteroskedasticity (H):               0.57   Skew:                            -0.24
Prob(H) (two-sided):                  0.02   Kurtosis:                         5.15
===================================================================================

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

```
arima_y_pred = model_fit.forecast(steps=len(X_test))
```

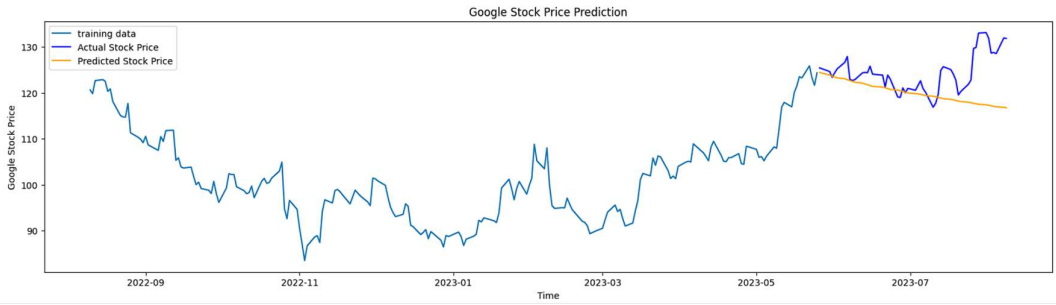
+ Code + Text

```
plt.plot(y_test, label="Actual Values")
plt.plot(arima_y_pred, label="ARIMA Predicted Values")
plt.legend()
plt.show()
```

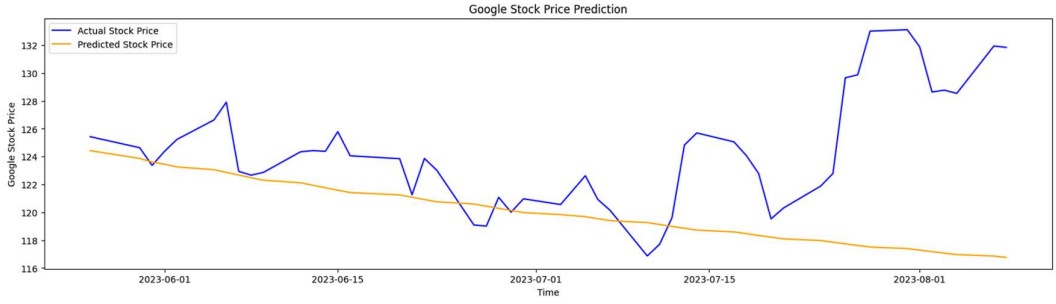



```
index = df.Close[train_size:len(X)].index
y_train = df.Close[0:train_size]
y_test = df.Close[train_size:len(X)]
```

```
# Forecast

# Make as pandas series
fc_series = pd.Series(arima_y_pred, index=index)

# Plot
plt.figure(figsize=(20,5), dpi=100)

plt.plot(y_train, label='training data')

plt.plot(y_test, color = 'blue', label='Actual Stock Price')

plt.plot(fc_series, color = 'orange',label='Predicted Stock Price')

plt.title('Google Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Google Stock Price')
plt.legend()
plt.show()
```




```
# Plot
plt.figure(figsize=(20,5), dpi=100)

plt.plot(y_test, color = 'blue', label='Actual Stock Price')

plt.plot(fc_series, color = 'orange',label='Predicted Stock Price')

plt.title('Google Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Google Stock Price')
plt.legend()
plt.show()
```



RESULT

The implementation was successful.

Experiment 5

AIM

Implementation of Auto regression (AR) model, and using Auto correlation function (ACF) to find the order of AR model.

SOFTWARE USED

Google Colaboratory

CODE AND OUTPUT

TSA LAB 5: ADF and KPSS Test and Implement ARIMA Model - Google Stock Price Data

```
[ ]  1 import pandas as pd
     2 import numpy as np
     3 import matplotlib.pyplot as plt
     4 import statsmodels.api as sm
     5 from statsmodels.tsa.seasonal import seasonal_decompose
     6 from statsmodels.tsa.stattools import adfuller
     7 from statsmodels.tsa.stattools import kpss
     8 from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
     9 from sklearn.linear_model import LinearRegression
    10 from sklearn.metrics import mean_squared_error
    11 from statsmodels.tsa.arima.model import ARIMA
```

```
[ ]  1 %matplotlib inline
```

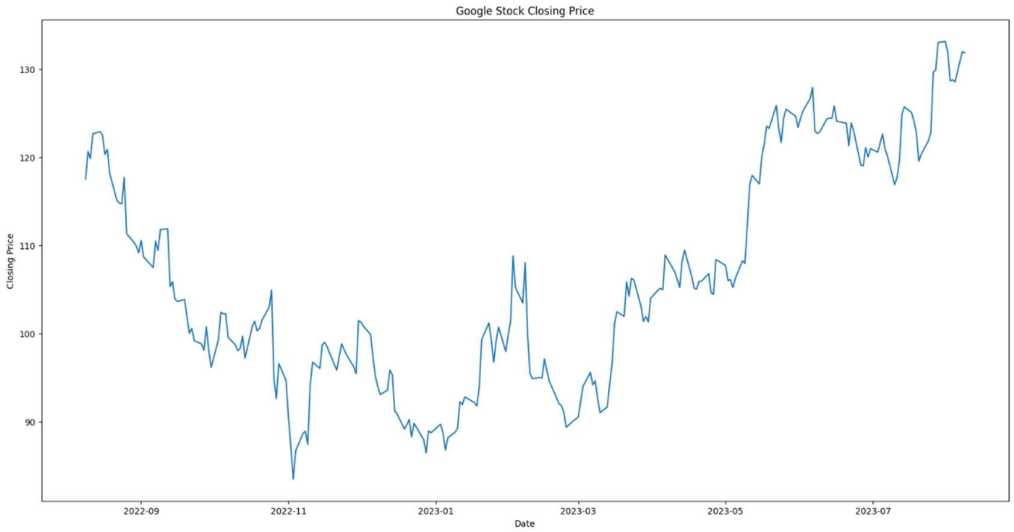
```
[ ]  1 file_path = "/content/GOOG.csv"
```

```
[ ]  1 df = pd.read_csv(file_path, index_col=0, parse_dates=True)
```

```
[ ]  1 df.head()
```

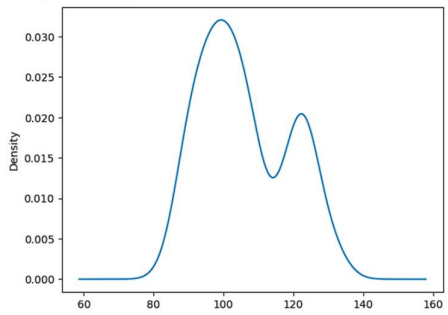
Date	Open	High	Low	Close	Adj Close	Volume
2022-08-09	117.989998	118.199997	116.559998	117.500000	117.500000	15424300
2022-08-10	119.589996	121.779999	119.360001	120.650002	120.650002	20497000
2022-08-11	122.080002	122.339996	119.550003	119.820000	119.820000	16671600
2022-08-12	121.160004	122.650002	120.400002	122.650002	122.650002	16121100
2022-08-15	122.209999	123.260002	121.570000	122.879997	122.879997	15525000

```
[ ]  1 # Plot the closing price
     2
     3 plt.rc("font", size=10)
     4
     5 plt.figure(figsize=(20, 10))
     6 plt.plot(df['Close'])
     7 plt.title('Google Stock Closing Price')
     8 plt.xlabel('Date')
     9 plt.ylabel('Closing Price')
    10 plt.show()
```



```
[ ]  1 df_close = df['Close']
     2 df_close.plot(kind='kde')
```

```
<Axes: ylabel='Density'>
```



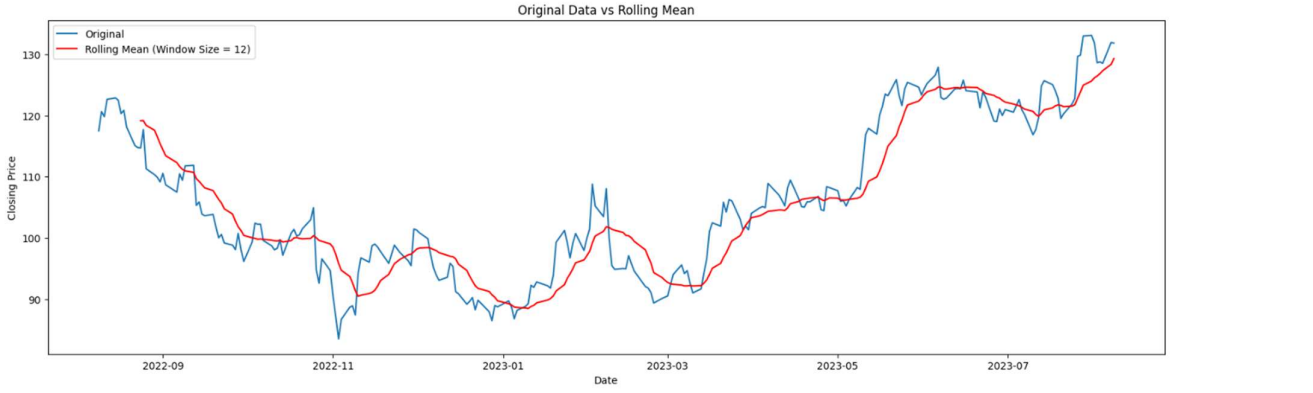

```
[ ]  1 # Split the data into 3 parts
     2 num_splits = 3
     3 split_size = len(df) // num_splits
     4 data_splits = [df.iloc[i * split_size: (i + 1) * split_size] for i in range(num_splits)]
     5
```

```
[ ]  1 # Calculate the mean and variance for each split
     2 for i, data_split in enumerate(data_splits):
     3     mean = np.mean(data_split['Close'])
     4     variance = np.var(data_split['Close'])
     5     print(f"Split {i + 1} - Mean: {mean:.2f}, Variance: {variance:.2f}")
     6
```

```
Split 1 - Mean: 102.91, Variance: 82.40
Split 2 - Mean: 95.75, Variance: 33.28
Split 3 - Mean: 118.68, Variance: 67.52
```

```
[ ]  1 # Calculate the rolling mean and plot the graph
     2 rolling_mean = df['Close'].rolling(window=12).mean()
```

```
[ ]  1 plt.figure(figsize=(20, 6))
     2 plt.plot(df['Close'], label='Original')
     3 plt.plot(rolling_mean, label='Rolling Mean (Window Size = 12)', color='red')
     4 plt.legend()
     5 plt.title('Original Data vs Rolling Mean')
     6 plt.xlabel('Date')
     7 plt.ylabel('Closing Price')
     8 plt.show()
```

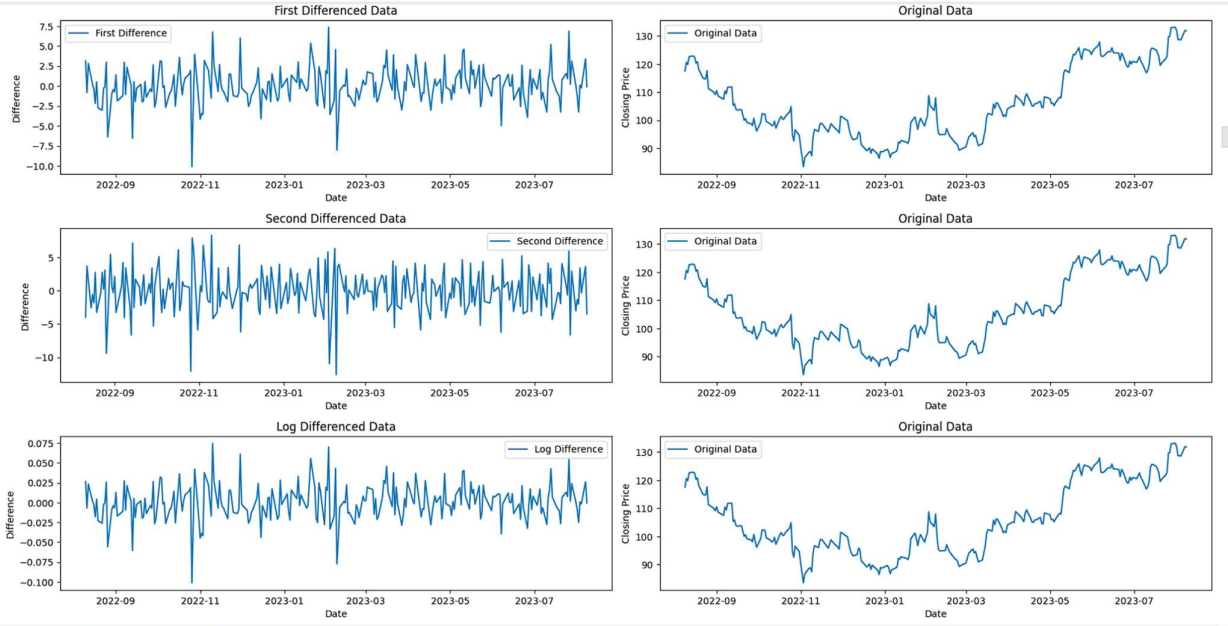



```
[ ]  1 X = df["Close"].values
     2 result = adfuller(X)
     3 print('ADF Statistic: %f' % result[0])
     4 print('p-value: %f' % result[1])
     5 print('Critical Values:')
     6 for key, value in result[4].items():
     7     print('\t%s: %.3f' % (key, value))
     8
     9 if result[0] < result[4]["5%"]:
    10     print ("Reject Ho - Time Series is Stationary")
    11 else:
    12     print ("Failed to Reject Ho - Time Series is Non-Stationary")
```

```
ADF Statistic: -0.893703
p-value: 0.790039
Critical Values:
        1%: -3.457
        5%: -2.873
        10%: -2.573
Failed to Reject Ho - Time Series is Non-Stationary
```

```
[ ]  1 # Apply differencing methods
     2 df_diff1 = df['Close'].diff().dropna()  # First difference
     3
     4 df_diff2 = df['Close'].diff().diff().dropna()  # Second difference
     5
     6 df_log_diff = df['Close'].apply(np.log).diff().dropna()  # Log difference
```

```
# Plot the differenced data
plt.figure(figsize=(18, 12))


plt.subplot(421)
plt.plot(df_diff1, label='First Difference')
plt.legend(loc='best')
plt.title('First Differenced Data')
plt.xlabel('Date')
plt.ylabel('Difference')


plt.subplot(422)
plt.plot(df['Close'], label='Original Data')
plt.legend(loc='best')
plt.title('Original Data')
plt.xlabel('Date')
plt.ylabel('Closing Price')


plt.subplot(423)
plt.plot(df_diff2, label='Second Difference')
plt.legend(loc='best')
plt.title('Second Differenced Data')
plt.xlabel('Date')
plt.ylabel('Difference')


plt.subplot(424)
plt.plot(df['Close'], label='Original Data')
plt.legend(loc='best')
plt.title('Original Data')
plt.xlabel('Date')
plt.ylabel('Closing Price')


plt.subplot(425)
plt.plot(df_log_diff, label='Log Difference')
plt.legend(loc='best')
plt.title('Log Differenced Data')
plt.xlabel('Date')

plt.subplot(426)
plt.plot(df['Close'], label='Original Data')
plt.legend(loc='best')
plt.title('Original Data')
plt.xlabel('Date')
plt.ylabel('Closing Price')


plt.tight_layout()
```




```
# ADF and KPSS tests for stationarity
def test_stationarity(timeseries):
    # ADF test

    adf_test = adfuller(timeseries, autolag='AIC')
    print("ADF Test Results:")
    print(f'ADF Test Statistic: {adf_test[0]}')
    print(f'p-value: {adf_test[1]}')
    # print(f'Critical Values: {adf_test[4]}')

    # result = adfuller(timeseries, autolag='AIC')
    # print('ADF Statistic: %f' % result[0])
    # print('p-value: %f' % result[1])
    print('Critical Values:')
    for key, value in adf_test[4].items():
        print(f'\t{key}: {value}')

    if adf_test[0] < adf_test[4]["5%"]:
        print ("Reject Ho - Time Series is Stationary")
    else:
        print ("Failed to Reject Ho - Time Series is Non-Stationary")


    # KPSS test
    kpss_test = kpss(timeseries, nlags='auto')
    print("\nKPSS Test Results:")
    print(f'KPSS Test Statistic: {kpss_test[0]}')
    print(f'p-value: {kpss_test[1]}')
    print(f'Critical Values: {kpss_test[3]}')
```

```
print("\nFirst Difference:")
test_stationarity(df_diff1)
```

```
First Difference:
ADF Test Results:
ADF Test Statistic: -11.971825846310379
p-value: 3.882320630634504e-22
Critical Values:
        1%: -3.4569962781990573
        5%: -2.8732659015936024
        10%: -2.573018897632674
Reject Ho - Time Series is Stationary

KPSS Test Results:
KPSS Test Statistic: 0.3514938486705948
p-value: 0.09806299626267465
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
```

+ Code + Text

```
1 print("\nSecond Difference:")
2 test_stationarity(df_diff2)
```

```
Second Difference:
ADF Test Results:
ADF Test Statistic: -8.353684562399444
p-value: 2.94802700957099e-13
Critical Values:
        1%: -3.4582467982399105
        5%: -2.8738137461081323
        10%: -2.5733111490323846
Reject Ho - Time Series is Stationary

KPSS Test Results:
KPSS Test Statistic: 0.13153242844233876
p-value: 0.1
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
<ipython-input-16-92d4b23f3d2b>:25: InterpolationWarning: The test statistic is outside of the range of p-values available in the
look-up table. The actual p-value is greater than the p-value returned.

  kpss_test = kpss(timeseries, nlags='auto')
```

```
1 print("\nLog Difference:")
2 test_stationarity(df_log_diff)
```

```
Log Difference:
ADF Test Results:
ADF Test Statistic: -12.110976964109183
p-value: 1.9197212216094285e-22
Critical Values:
        1%: -3.4569962781990573
        5%: -2.8732659015936024
        10%: -2.573018897632674
Reject Ho - Time Series is Stationary

KPSS Test Results:
KPSS Test Statistic: 0.31381152797403866
p-value: 0.1
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
<ipython-input-16-92d4b23f3d2b>:25: InterpolationWarning: The test statistic is outside of the range of p-values available in the
look-up table. The actual p-value is greater than the p-value returned.

  kpss_test = kpss(timeseries, nlags='auto')
```

```
1 sdi = df.asfreq('D')
2 sdi = sdi[['Close']]
3 sdi
```

Date	Close
2022-08-09	117.500000
2022-08-10	120.650002
2022-08-11	119.820000
2022-08-12	122.650002
2022-08-13	NaN
...	...
2023-08-04	128.539993
2023-08-05	NaN
2023-08-06	NaN
2023-08-07	131.940002
2023-08-08	131.839996

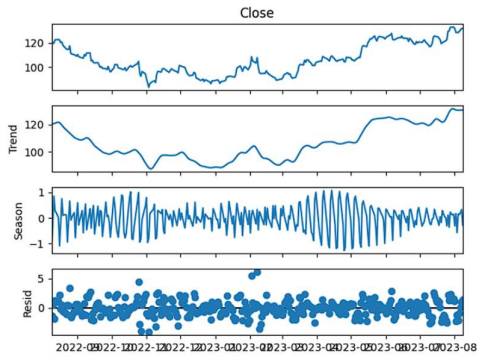
365 rows × 1 columns

+ Code + Text

```
1 from statsmodels.tsa.seasonal import STL
```

```
1 # in this place "use linear interpolation here" use the commands for linear interpolation.
2 # sdi = Seasonal Decomposition and Interpolation to interpolate missing values in the daily frequency of data
3
4 res = STL(sdi['Close'].interpolate(method="linear"), seasonal=13).fit()
```

```
1 res.plot()
```




```
1 seasonal_component = res.seasonal
2 seasonal_component.head()
```

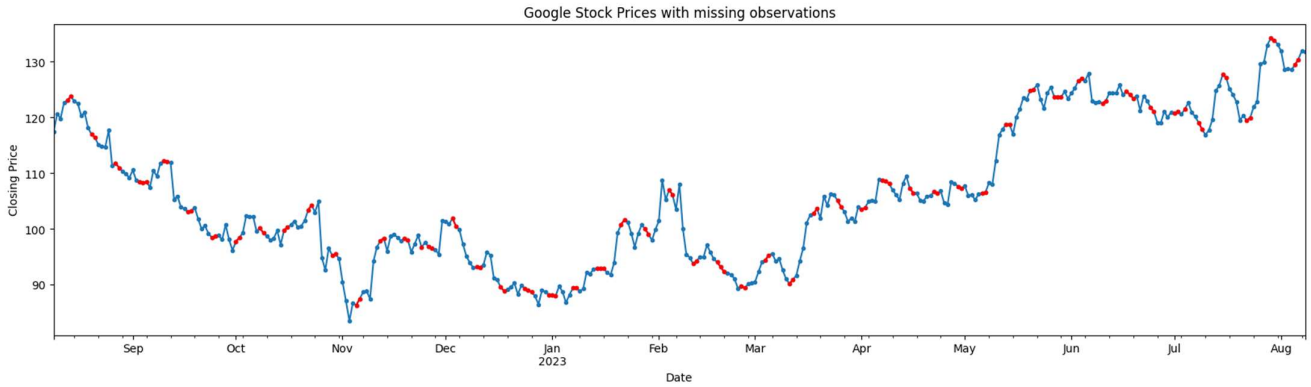
```
Date
2022-08-09   -1.224121
2022-08-10   -0.260225
2022-08-11    0.852518
2022-08-12    0.272024
2022-08-13    0.208928
Freq: D, Name: season, dtype: float64
```

```
[ ]  1 sdi
```

Date	Close
2022-08-09	117.500000
2022-08-10	120.650002
2022-08-11	119.820000
2022-08-12	122.650002
2022-08-13	NaN
...	...
2023-08-04	128.539993
2023-08-05	NaN
2023-08-06	NaN
2023-08-07	131.940002
2023-08-08	131.839996

365 rows × 1 columns

```
[ ]  1 sdi_deseasonalised = sdi['Close'] - seasonal_component
     2 sdi_deseasonalised_imputed = sdi_deseasonalised.interpolate(method="spline",order=3)
     3 sdi_imputed = sdi_deseasonalised_imputed + seasonal_component
     4
     5 sdi_imputed = sdi_imputed.to_frame().rename(columns={0: 'Close'})
     6
     7 # plot
     8 ax = sdi_imputed.plot(linestyle="-", marker=".", figsize=[20, 5], legend=None)
     9 ax = sdi_imputed[sdi.isnull()].plot(ax=ax, legend=None, marker=".", color="r")
    10 ax.set_title('Google Stock Prices with missing observations')
    11 ax.set_ylabel('Closing Price')
    12 ax.set_xlabel('Date')
```

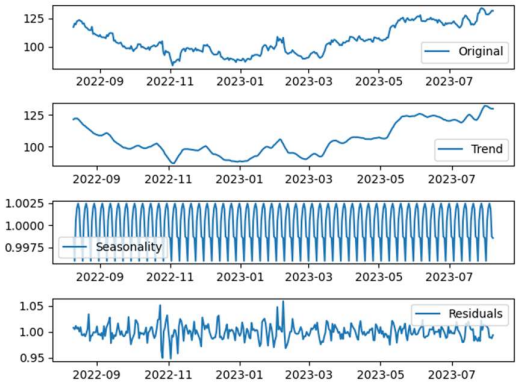



```
[ ]  1 sdi_imputed
```

Date	Close
2022-08-09	117.500000
2022-08-10	120.650002
2022-08-11	119.820000
2022-08-12	122.650002
2022-08-13	123.090103
...	...
2023-08-04	128.539993
2023-08-05	129.406770
2023-08-06	130.415954
2023-08-07	131.940002
2023-08-08	131.839996

365 rows × 1 columns

```
# perform seasonal decompose on logged transformed

decompose_result = seasonal_decompose(sdi_imputed, model='multiplicative', filt=None, period=None, two_sided=True, extrapolate_trend=0)

# perform seasonal decomposing using additive model type

trend = decompose_result.trend # get trend
seasonal = decompose_result.seasonal #get seasonal
residual = decompose_result.resid #get residual

#plot the decompose result
# decompose_result.plot("Multiplicative Decomposition")

plt.subplot(411)
plt.plot(sdi_imputed, label='Original')
plt.legend(loc='best')

plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')

plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')

plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```



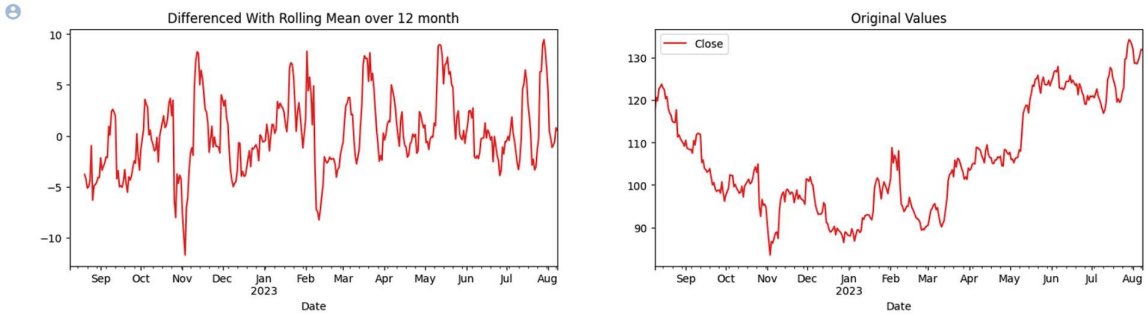

```
# perform the ADF test
from statsmodels.tsa.stattools import adfuller

dftest = adfuller(sdi_imputed, autolag = 'AIC')

print("1. ADF : ",dftest[0])
print("2. P-Value : ", dftest[1])
print("3. Num Of Lags : ", dftest[2])
print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
print("5. Critical Values :")
for key, val in dftest[4].items():
    print("\t",key, ": ", val)
```

```
1. ADF :  -1.0295989868277595
2. P-Value :  0.7423162782004037
3. Num Of Lags :  4
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 360
5. Critical Values :
	1% :  -3.448645946352023
	5% :  -2.869602139060357
	10% :  -2.5710650077160495
```

```
sdi_imputed.isnull().any()
```

```
Close    False
dtype: bool
```

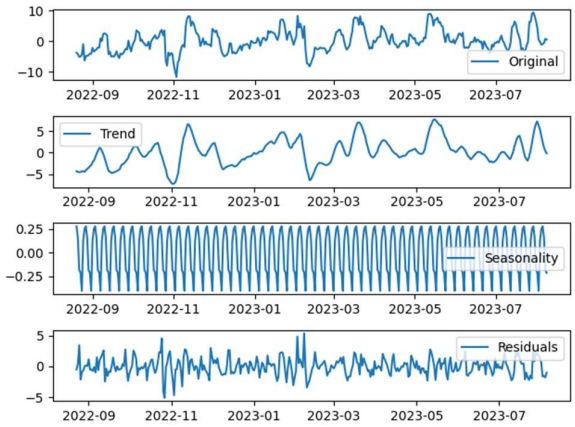
```
rolling_mean = sdi_imputed.rolling(window = 12).mean()
sdi_rolled_detrended = sdi_imputed['Close'] - rolling_mean['Close']


ax1 = plt.subplot(121)
sdi_rolled_detrended.plot(figsize=(18,4),color="tab:red", title="Differenced With Rolling Mean over 12 month", ax=ax1);
ax2 = plt.subplot(122)
sdi_imputed.plot(figsize=(18,4), color="tab:red", title="Original Values", ax=ax2);
```



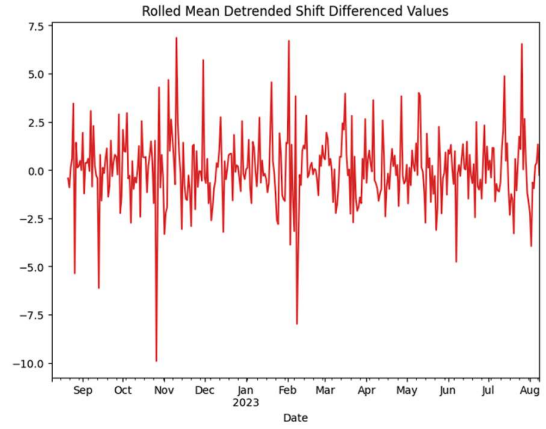

```
sdi_rolled_detrended
```

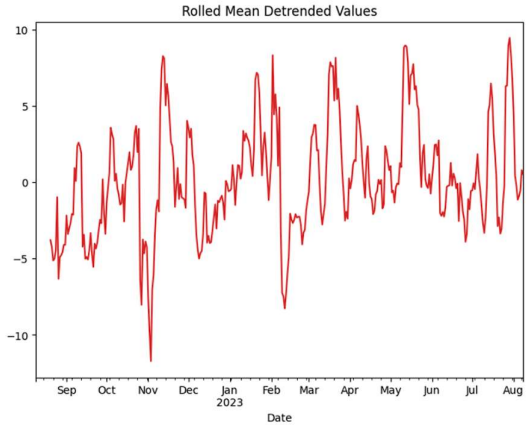
```
Date
2022-08-09         NaN
2022-08-10         NaN
2022-08-11         NaN
2022-08-12         NaN
2022-08-13         NaN
                ...
2023-08-04   -1.153109
2023-08-05   -0.913563
2023-08-06   -0.539876
2023-08-07    0.794173
2023-08-08    0.530000
Freq: D, Name: Close, Length: 365, dtype: float64
```

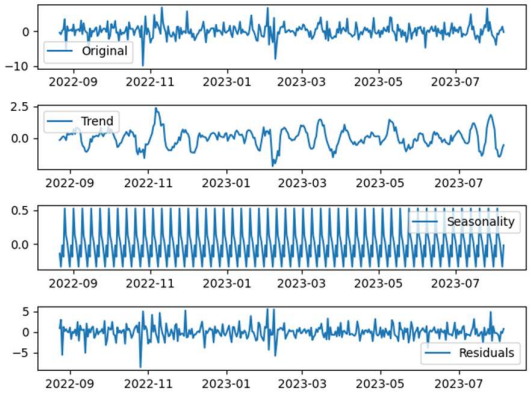
```
decompose_result = seasonal_decompose(sdi_rolled_detrended.dropna(), model='additive', filt=None, period=None, two_sided=True, extrapolate_trend=0)

# perform seasonal decomposing using multiplicative model type

trend = decompose_result.trend # get trend
seasonal = decompose_result.seasonal #get seasonal
residual = decompose_result.resid #get residual

#plot the decompose result
# decompose_result.plot("Multiplicative Decomposition")

plt.subplot(411)
plt.plot(sdi_rolled_detrended, label='Original')
plt.legend(loc='best')

plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')

plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')

plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```




```
[ ] sdi_rolled_detrended_diff = sdi_rolled_detrended - sdi_rolled_detrended.shift()

# plot the logged_passengers_diff
ax1 = plt.subplot(121)
sdi_rolled_detrended_diff.plot(figsize=(18,6) ,color="tab:red", title="Rolled Mean Detrended Shift Differenced Values", ax=ax1);
ax2 = plt.subplot(122)
sdi_rolled_detrended.plot(color="tab:red", title="Rolled Mean Detrended Values", ax=ax2);
```

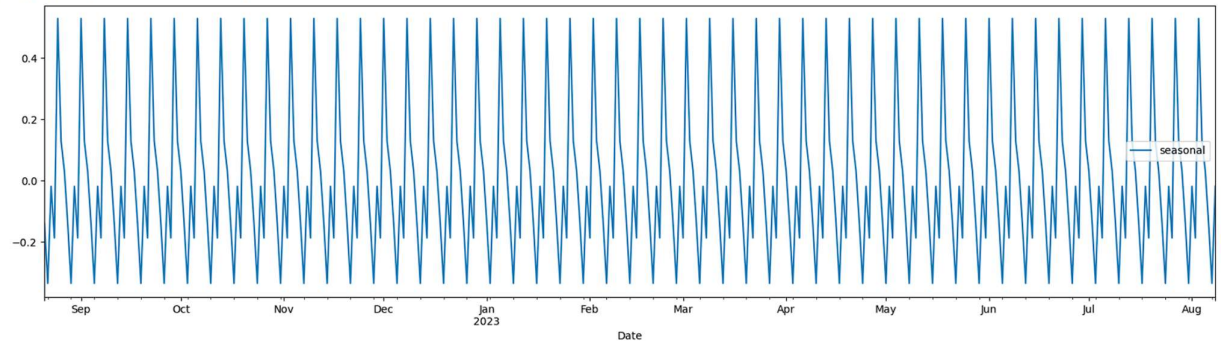






```
plt.figure(figsize=(20,5))
seasonal.plot()
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7d11c1da5b10>
```




```
# perform the ADF test
from statsmodels.tsa.stattools import adfuller

dftest = adfuller(sdi_rolled_detrended_diff.dropna(), autolag = 'AIC')

print("1. ADF : ",dftest[0])
print("2. P-Value : ", dftest[1])
print("3. Num Of Lags : ", dftest[2])
print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", dftest[3])
print("5. Critical Values :")
for key, val in dftest[4].items():
    print("\t",key, ": ", val)
```

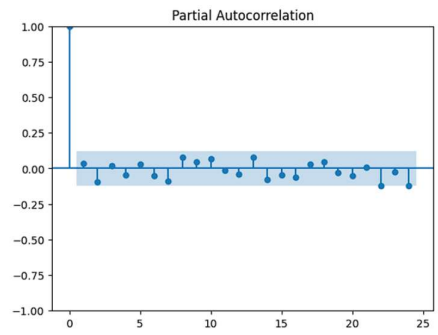
```
1. ADF :  -9.207865120768997
2. P-Value :  1.9230229748700873e-15
3. Num Of Lags :  9
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 343
5. Critical Values :
	1% :  -3.449559661646851
	5% :  -2.8700035112469626
	10% :  -2.571279073013796
```

```
# df = pd.DataFrame(sdi_rolled_detrended_diff)
```

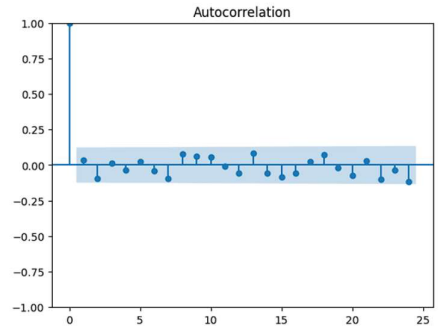
```
# X = sdi_rolled_detrended_diff.dropna()
```

```
X = df
```

```
plot_pacf(X.Close.diff().dropna());
```




```
plot_acf(X.Close.diff().dropna());
```



```
[ ] 1 df["Close_shifted"] = df["Close"].shift()
```

```
1 df.head()
```

Date	Open	High	Low	Close	Adj Close	Volume	Close_shifted
2022-08-09	117.989998	118.199997	116.559998	117.500000	117.500000	15424300	NaN
2022-08-10	119.589996	121.779999	119.360001	120.650002	120.650002	20497000	117.500000
2022-08-11	122.080002	122.339996	119.550003	119.820000	119.820000	16671600	120.650002
2022-08-12	121.160004	122.650002	120.400002	122.650002	122.650002	16121100	119.820000
2022-08-15	122.209999	123.260002	121.570000	122.879997	122.879997	15525000	122.650002

+ Code + Text

```
[ ] 1 columns_to_remove = df.columns.drop(['Close','Close_shifted'])
```

+ Code + Text

```
[ ] 1 df.drop(columns_to_remove, axis = 1, inplace=True)
```

```
[ ] 1 df.dropna(inplace=True)
```

```
[ ] 1 df.head()
```

Date	Close	Close_shifted
2022-08-10	120.650002	117.500000
2022-08-11	119.820000	120.650002
2022-08-12	122.650002	119.820000
2022-08-15	122.879997	122.650002
2022-08-16	122.510002	122.879997

```
[ ] 1 y = df.Close.values
    2 X = df.Close_shifted.values
```

```
[ ] 1 train_size = int(len(X) * 0.80)
```

+ Code + Text

```
[ ] 1 X_train, X_test = X[0:train_size], X[train_size:len(X)]
    2 y_train, y_test = y[0:train_size], y[train_size:len(X)]
```

```
[ ] 1 X_train = X_train.reshape(-1,1)
    2 X_test = X_test.reshape(-1,1)
```

```
[ ] 1 lr = LinearRegression()
    2 lr.fit(X_train, y_train)
```

▾ LinearRegression
LinearRegression()

```
[ ] 1 lr.coef_
```

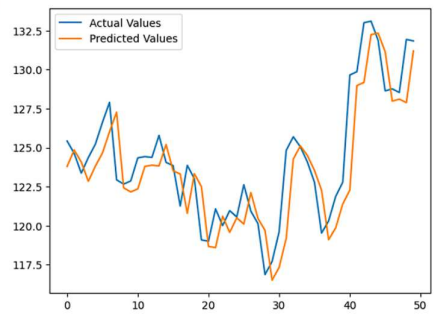
```
array([0.97480486])
```

```
[ ] 1 lr.intercept_
```

```
2.5871238391784317
```

```
[ ] 1 y_pred = lr.predict(X_test)
```

```
1 plt.plot(y_test, label="Actual Values")
2 plt.plot(y_pred, label="Predicted Values")
3 plt.legend()
4 plt.show()
```




```
[ ] 1 model = ARIMA(y_train, order=(1,1,2))
```

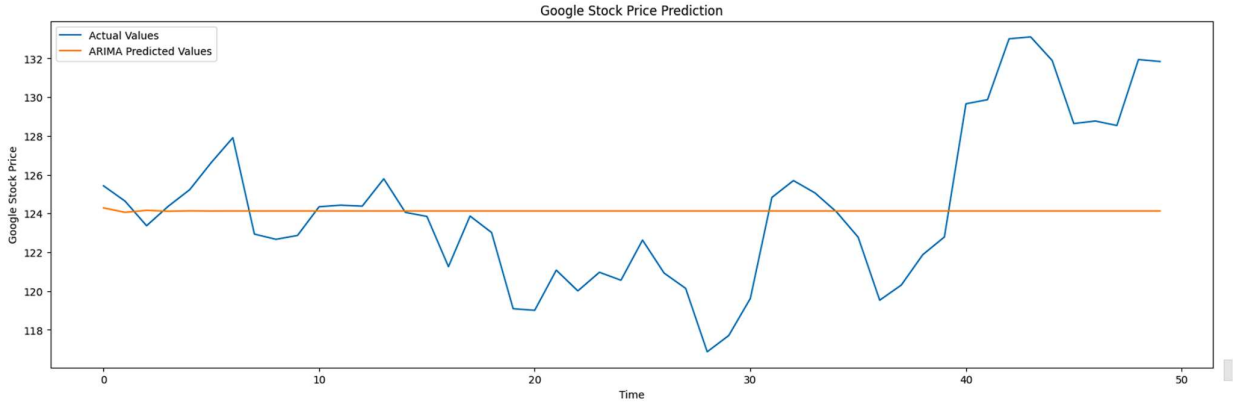
```
[ ] 1 model_fit = model.fit()
```

```
[ ]  1 print(model_fit.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                  200
Model:                 ARIMA(1, 1, 2)   Log Likelihood                -451.844
Date:                Sun, 27 Aug 2023   AIC                            911.689
Time:                        10:55:36   BIC                            924.862
Sample:                             0   HQIC                           917.020
                                  - 200
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.4444      0.301     -1.474      0.140      -1.035       0.146
ma.L1          0.4964      0.311      1.598      0.110      -0.113       1.105
ma.L2         -0.1230      0.095     -1.294      0.195      -0.309       0.063
sigma2         5.4898      0.396     13.852      0.000       4.713       6.267
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):                43.59
Prob(Q):                              0.98   Prob(JB):                         0.00
Heteroskedasticity (H):               0.56   Skew:                            -0.22
Prob(H) (two-sided):                  0.02   Kurtosis:                         5.25
===================================================================================

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

```
[ ]  1 arima_y_pred = model_fit.forecast(steps=len(y_test))
```

```
1 plt.figure(figsize=(20,6))
2 plt.plot(y_test, label="Actual Values")
3 plt.plot(arima_y_pred, label="ARIMA Predicted Values")
4 plt.title('Google Stock Price Prediction')
5 plt.xlabel('Time')
6 plt.ylabel('Google Stock Price')
7 plt.legend()
8 plt.show()
```




```
[ ]  1 import math
     2 from sklearn.metrics import mean_squared_error, mean_absolute_error
```

```
1 # report performance
2 mse = mean_squared_error(y_test, arima_y_pred)
3 print('MSE: '+str(mse))
4
5 mae = mean_absolute_error(y_test, arima_y_pred)
6 print('MAE: '+str(mae))
7
8 rmse = math.sqrt(mean_squared_error(y_test, arima_y_pred))
9 print('RMSE: '+str(rmse))
10
11 mape = np.mean(np.abs(arima_y_pred - y_test)/np.abs(y_test))
12 print('MAPE: '+str(mape))
```

```
MSE: 15.96584803707989
MAE: 3.092057823292232
RMSE: 3.995728724160324
MAPE: 0.024711890161486717
```

RESULT

The implementation was successful.

Experiment 6

AIM

Implementing Autoregressive integrated moving average (ARIMA) model, also implement Auto- ARIMA model.

SOFTWARE USED

Google Colaboratory

CODE AND OUTPUT

TSA LAB 6: Auto ARIMA Model - Air Passengers Dataset

+ Code + Text

About the Dataset

The dataset contains number of air passengers of each month from the year 1949 to 1960.

```
[ ]  1 import pandas, numpy, matplotlib,seaborn as sns
     2
     3
     4 sns.set(style="whitegrid", color_codes=True, font_scale=1.3)
     5
     6 import warnings
     7 warnings.filterwarnings('ignore')
```

First let´s load the dataset and get a general overview.

```
[ ]  1 original = pandas.read_csv("/content/AirPassengers (1).csv", index_col=['Month'], parse_dates=['Month'])
     2
     3 # read data csv file using pandas and make Month column as index
     4 #original = original.set_index('Month')
     5 # preview data using head command
     6 original.head()
```

Month	#Passengers
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

```
[ ]  1 original.rename(columns = {'#Passengers' : 'Passengers'}, inplace = True)
```

```
[ ]  1 # preview data using head command
     2 original.head()
```

Month	Passengers
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

```
[ ]  1 # check data shape
     2 original.shape
```

```
(144, 1)
```

```
[ ]  1 # use describe command to describe your data
     2 original.describe()
```

	Passengers
count	144.000000
mean	280.298611
std	119.966317
min	104.000000
25%	180.000000
50%	265.500000
75%	360.500000
max	622.000000

```
[ ]  1 import matplotlib.pyplot as plt
     2 from statsmodels.tsa.arima.model import ARIMA
     3 import pandas as pd
```

```
[ ]  1 data = original
```

+ Code + Text

```
[ ]  1 data.dropna(inplace=True)
```

```
[ ]  1 train_size = int(len(data) * 0.80)
```

```
[ ]  1 y_train, y_test = data[0:train_size], data[train_size:]
```

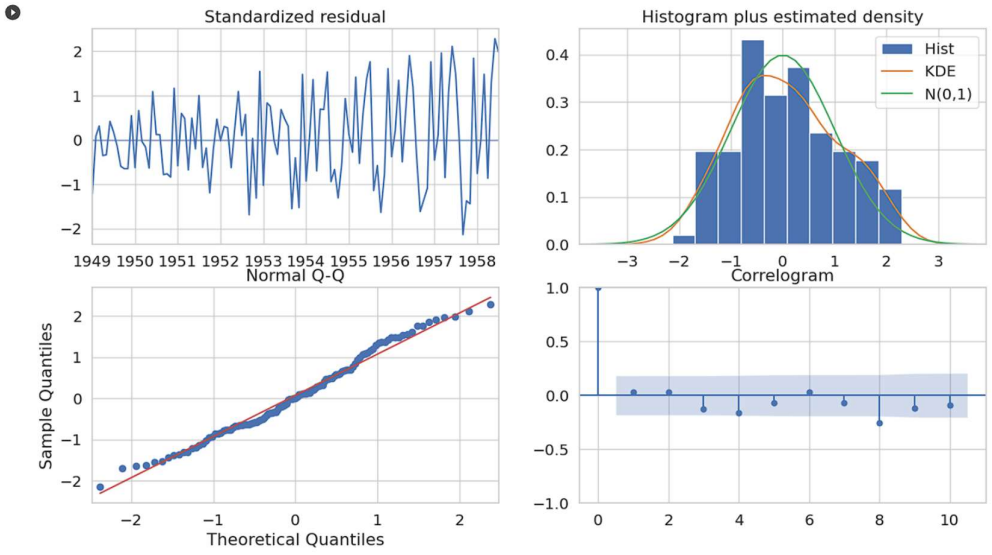
+ Code + Text

```
[ ]  1 !pip install pmdarima
     2
     3 from pmdarima.arima import auto_arima
```

```
[ ]  1 model_autoARIMA = auto_arima(y_train, start_p=0, start_q=0,
     2                       test='adf',       # use adftest to find optimal 'd'
     3                       max_p=20, max_q=20, # maximum p and q
     4                       m=1,              # frequency of series
     5                       d=None,           # let model determine 'd'
     6                       seasonal=False,   # No Seasonality
     7                       start_P=0,
     8                       D=0,
     9                       trace=True,
    10                       error_action='ignore',
    11                       suppress_warnings=True,
    12                       stepwise=True)
    13
    14
    15 print(model_autoARIMA.summary())
    16 model_autoARIMA.plot_diagnostics(figsize=(15,8))
    17 plt.show()
    18
```

```
Performing stepwise search to minimize aic
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=1604.291, Time=0.02 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=inf, Time=0.04 sec
 ARIMA(0,0,1)(0,0,0)[0]             : AIC=inf, Time=0.07 sec
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=1079.817, Time=0.06 sec
 ARIMA(2,0,1)(0,0,0)[0]             : AIC=1074.529, Time=0.07 sec
 ARIMA(2,0,0)(0,0,0)[0]             : AIC=inf, Time=0.05 sec
 ARIMA(3,0,1)(0,0,0)[0]             : AIC=1075.775, Time=0.11 sec
 ARIMA(2,0,2)(0,0,0)[0]             : AIC=1075.654, Time=0.12 sec
 ARIMA(1,0,2)(0,0,0)[0]             : AIC=1076.970, Time=0.08 sec
 ARIMA(3,0,0)(0,0,0)[0]             : AIC=inf, Time=0.08 sec
 ARIMA(3,0,2)(0,0,0)[0]             : AIC=inf, Time=0.28 sec
 ARIMA(2,0,1)(0,0,0)[0] intercept   : AIC=1073.718, Time=0.38 sec
 ARIMA(1,0,1)(0,0,0)[0] intercept   : AIC=1078.074, Time=0.14 sec
 ARIMA(2,0,0)(0,0,0)[0] intercept   : AIC=1083.134, Time=0.21 sec
 ARIMA(3,0,1)(0,0,0)[0] intercept   : AIC=1074.610, Time=0.41 sec
 ARIMA(2,0,2)(0,0,0)[0] intercept   : AIC=1074.541, Time=0.44 sec
 ARIMA(1,0,0)(0,0,0)[0] intercept   : AIC=1092.214, Time=0.08 sec
 ARIMA(1,0,2)(0,0,0)[0] intercept   : AIC=1076.299, Time=0.22 sec
 ARIMA(3,0,0)(0,0,0)[0] intercept   : AIC=1081.054, Time=0.22 sec
 ARIMA(3,0,2)(0,0,0)[0] intercept   : AIC=inf, Time=0.36 sec

Best model:  ARIMA(2,0,1)(0,0,0)[0] intercept
Total fit time: 3.495 seconds
```

```
Best model:  ARIMA(2,0,1)(0,0,0)[0] intercept
Total fit time: 3.495 seconds
                               SARIMAX Results
==========================================================================
Dep. Variable:                      y   No. Observations:              115
Model:               SARIMAX(2, 0, 1)   Log Likelihood              -531.859
Date:                Thu, 14 Sep 2023   AIC                        1073.718
Time:                        05:00:54   BIC                        1087.443
Sample:                    01-01-1949   HQIC                       1079.289
                         - 07-01-1958
Covariance Type:                  opg
==========================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------
intercept     12.5015     12.703      0.984      0.325     -12.396      37.399
ar.L1          0.4758      0.115      4.136      0.000       0.250       0.701
ar.L2          0.4765      0.122      3.912      0.000       0.238       0.715
ma.L1          0.9077      0.060     15.192      0.000       0.791       1.025
sigma2       587.7947    102.211      5.751      0.000     387.465     788.124
===================================================================================
Ljung-Box (L1) (Q):                   0.12   Jarque-Bera (JB):                 3.24
Prob(Q):                              0.73   Prob(JB):                         0.20
Heteroskedasticity (H):               4.80   Skew:                             0.20
Prob(H) (two-sided):                  0.00   Kurtosis:                         2.28
===================================================================================
```



```
[ ] 1 data.index.freq = 'MS'
```

```
1 #Modeling
2 # Build Model
3 model = ARIMA(y_train, order=(3,0,2))
4 fitted = model.fit()
5 print(fitted.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:             Passengers   No. Observations:                  115
Model:                 ARIMA(3, 0, 2)   Log Likelihood                -531.307
Date:                Thu, 14 Sep 2023   AIC                           1076.614
Time:                        05:04:16   BIC                           1095.828
Sample:                    01-01-1949   HQIC                          1084.413
                         - 07-01-1958
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        240.0202     73.518      3.265      0.001      95.928     384.113
ar.L1          0.3270      0.975      0.335      0.737      -1.584       2.238
ar.L2          0.6475      0.417      1.554      0.120      -0.169       1.464
ar.L3         -0.0392      0.530     -0.074      0.941      -1.078       1.000
ma.L1          1.1032      0.987      1.118      0.264      -0.831       3.038
ma.L2          0.1447      0.924      0.157      0.876      -1.666       1.956
sigma2       581.4949    105.575      5.508      0.000     374.571     788.419
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):                 3.94
Prob(Q):                              0.95   Prob(JB):                         0.14
Heteroskedasticity (H):               5.11   Skew:                             0.32
Prob(H) (two-sided):                  0.00   Kurtosis:                         2.36
===================================================================================
```

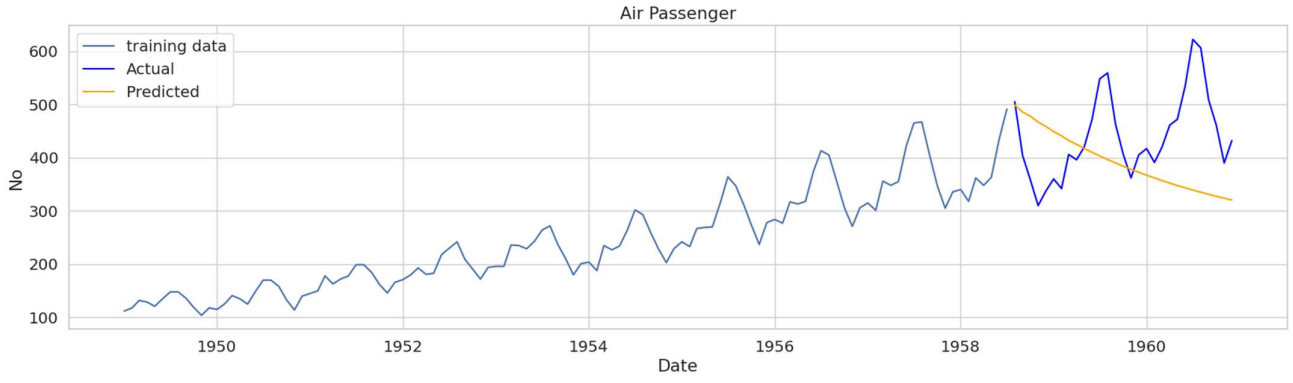
```
[ ] 1 data
```

Month	Passengers
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121
...	...
1960-08-01	606
1960-09-01	508
1960-10-01	461
1960-11-01	390
1960-12-01	432

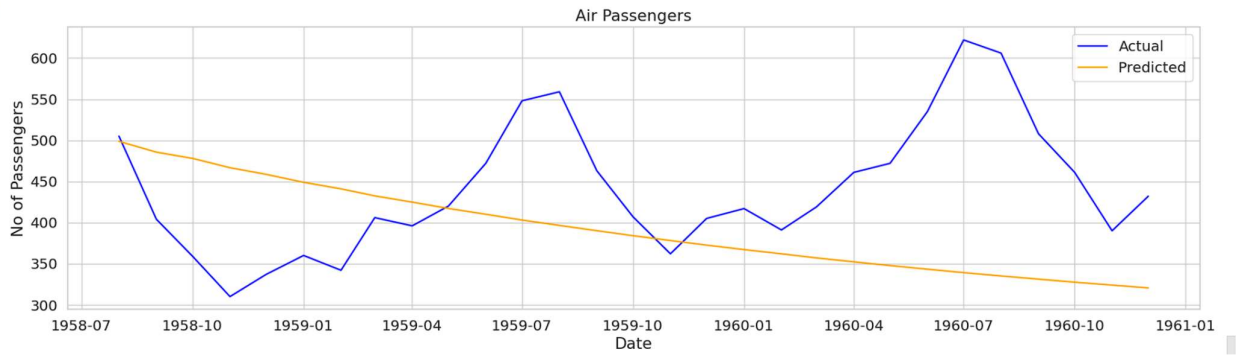
144 rows × 1 columns

```
[ ] 1 index = data[train_size:].index
2 y_train = data[0:train_size]
3 y_test = data[train_size:]
```

```
1 # Forecast
2 fc = fitted.forecast(steps=len(y_test))  # 95% conf
3 # Make as pandas series
4 fc_series = pd.Series(fc, index=index)
5 # Plot
6 plt.figure(figsize=(20,5), dpi=100)
7
8 plt.plot(y_train, label='training data')
9
10 plt.plot(y_test, color = 'blue', label='Actual')
11
12 plt.plot(fc_series, color = 'orange',label='Predicted')
13
14 plt.title('Air Passenger')
15 plt.xlabel('Date')
16 plt.ylabel('No')
17 plt.legend()
18 plt.show()
19
```



```
# Plot
plt.figure(figsize=(20,5), dpi=100)

plt.plot(y_test, color = 'blue', label='Actual')

plt.plot(fc_series, color = 'orange',label='Predicted')

plt.title('Air Passengers')
plt.xlabel('Date')
plt.ylabel('No of Passengers')
plt.legend()
plt.show()
```



RESULT

The implementation was successful.

Experiment 7

AIM

Implementing Random Forest Regressor Model for time series Forecasting

SOFTWARE USED

Google Colaboratory

CODE AND OUTPUT

TSA LAB 7: Random Forest Regressor Model - Female Birth Dataset

```
import pandas as pd

url="https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-total-female-births.csv"
c=pd.read_csv(url)
```

```
c.head()
```

	Date	Births
0	1959-01-01	35
1	1959-01-02	32
2	1959-01-03	30
3	1959-01-04	31
4	1959-01-05	44

```
y = c.Births
y.rename("y",inplace=True)
```

```
0      35
1      32
2      30
3      31
4      44
       ..
360    37
361    52
362    48
363    55
364    50
Name: y, Length: 365, dtype: int64
```

```
x = y.shift()

x.rename("x", inplace=True)
```

```
0       NaN
1      35.0
2      32.0
3      30.0
4      31.0
       ...
360    34.0
361    37.0
362    52.0
363    48.0
364    55.0
Name: x, Length: 365, dtype: float64
```

```
data = pd.DataFrame([x,y])
```

+ Code + Text

```
data
```

	0	1	2	3	4	5	6	7	8	9	...	355	356	357	358	359	360	361	362	363	364
x	NaN	35.0	32.0	30.0	31.0	44.0	29.0	45.0	43.0	38.0	...	53.0	39.0	40.0	38.0	44.0	34.0	37.0	52.0	48.0	55.0
y	35.0	32.0	30.0	31.0	44.0	29.0	45.0	43.0	38.0	27.0	...	39.0	40.0	38.0	44.0	34.0	37.0	52.0	48.0	55.0	50.0

2 rows × 365 columns

```
data = data.T
```

```
data
```

	x	y
0	NaN	35.0
1	35.0	32.0
2	32.0	30.0
3	30.0	31.0
4	31.0	44.0
...	...	...
360	34.0	37.0
361	37.0	52.0
362	52.0	48.0
363	48.0	55.0
364	55.0	50.0

365 rows × 2 columns

```
[ ]  1 data.dropna(inplace=True)
```

+ Code + Text

```
[ ]  1 train_size = int(len(data.x) * 0.80)
     2 train = data[0:train_size]
     3 test = data[train_size:len(data)]
```

```
[ ]  1 from sklearn.ensemble import RandomForestRegressor
     2 import numpy as np
```

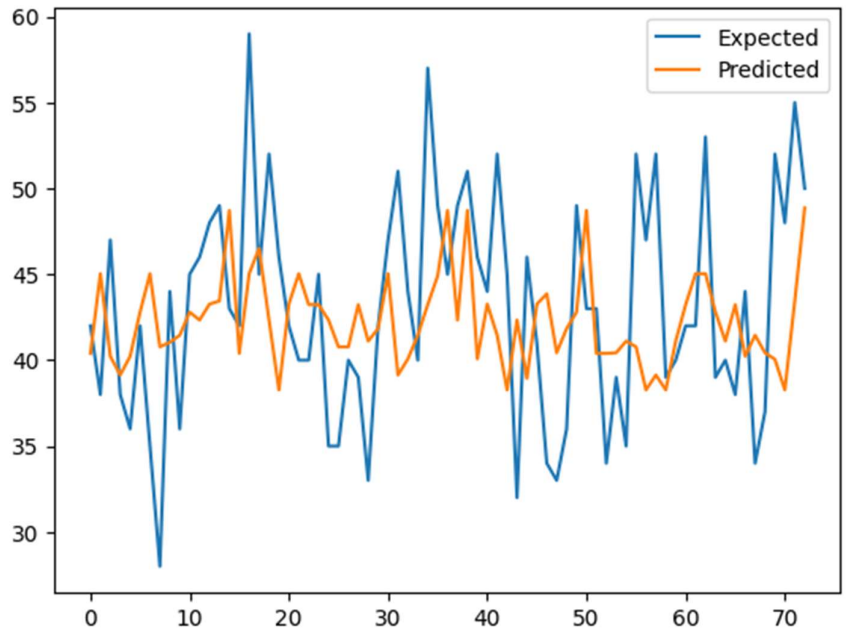
```
[ ]  1 train = np.asarray(train)
     2 trainX, trainy = train[:, :-1], train[:, -1]
     3 model = RandomForestRegressor(n_estimators=1000)
     4 model.fit(trainX, trainy)
```

```
RandomForestRegressor
RandomForestRegressor(n_estimators=1000)
```

```
[ ]  1 test = np.asarray(test)
     2 testX, testy = test[:, :-1], test[:, -1]
     3 yhat = model.predict(testX)
```

```
[ ]  1 import matplotlib.pyplot as plt
```

```
[ ]  1 plt.plot(testy, label='Expected')
     2 plt.plot(yhat, label='Predicted')
     3 plt.legend()
     4 plt.show()
```



RESULT

The implementation was successful.

Experiment 8

AIM

Implementing 1D CNN for time series Forecasting.

SOFTWARE USED

Google Colaboratory

CODE AND OUTPUT

```
[ ]  1 from google.colab import drive
     2 drive.mount('/content/drive/')
```

```
Drive already mounted at /content/drive/; to attempt to forcibly remount, call drive.mount("/content/drive/", force_remount=True).
```

```
[ ]  1 import pandas as pd
     2 import numpy as np
```

```
[ ]  1 data = pd.read_csv('/content/drive/MyDrive/Hitesh TSA (CSE471) LAB Work/DATASETs/LAB 8/webtraffic (2).csv')
```

```
[ ]  1 data.shape
```

```
(4896, 2)
```

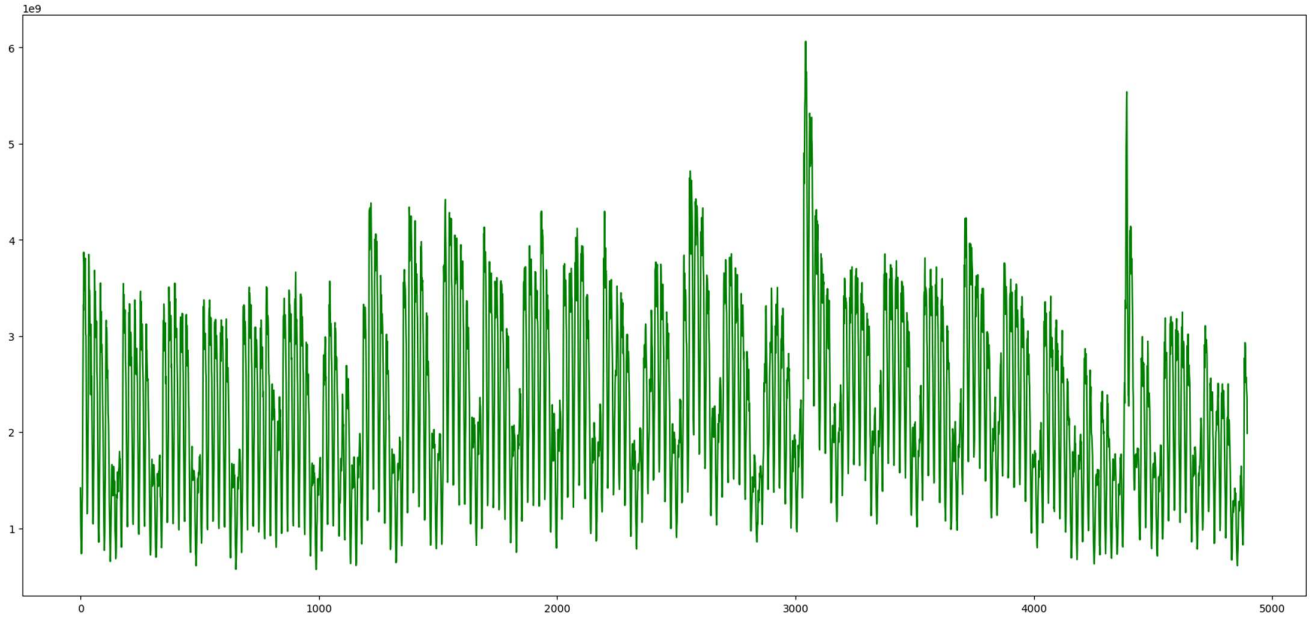
```
[ ]  1 data.head()
```

	Hour Index	Sessions
0	0	1418159421
1	1	1113769116
2	2	919158921
3	3	822352824
4	4	735526737

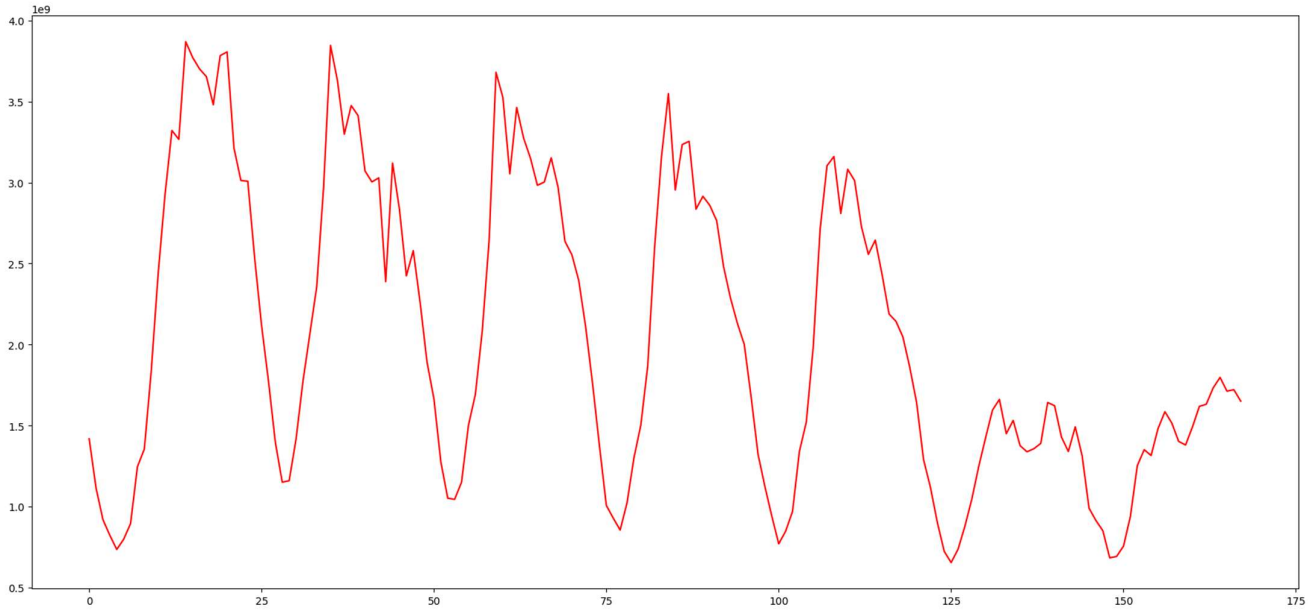
```
[ ]  1 import matplotlib.pyplot as plt
```

```
[ ]  1 sessions = data['Sessions']
```

```
     1 ar = np.arange(len(sessions))
     2 plt.figure(figsize=(22,10))
     3 plt.plot(ar, sessions,'g')
     4 plt.show()
```




```
     1 #first week web traffic
     2 sample = sessions[:168]
     3 ar = np.arange(len(sample))
     4 plt.figure(figsize=(22,10))
     5 plt.plot(ar, sample,'r')
     6 plt.show()
```




```
def prepare_data(seq,num):
  x=[]
  y=[]
  for i in range(0,(len(seq)-num),1):

    input_ = seq[i:i+num]
    output  = seq[i+num]

    x.append(input_)
    y.append(output)

  return np.array(x), np.array(y)
```

+ Code + Text

```
num=168
x,y= prepare_data(sessions,num)
print(len(x))
```

```
4728
```

```
ind = int(0.9 * len(x))
x_tr = x[:ind]
y_tr = y[:ind]
x_val=x[ind:]
y_val=y[ind:]
```

+ Code + Text

```
from sklearn.preprocessing import StandardScaler
```

```
#normalize the inputs
x_scaler= StandardScaler()
x_tr = x_scaler.fit_transform(x_tr)
x_val= x_scaler.transform(x_val)
#reshaping the output for normalization
y_tr=y_tr.reshape(len(y_tr),1)
y_val=y_val.reshape(len(y_val),1)
#normalize the output
y_scaler=StandardScaler()
y_tr = y_scaler.fit_transform(y_tr)[:,0]
y_val = y_scaler.transform(y_val)[:,0]

```

```
#reshaping input data
x_tr= x_tr.reshape(x_tr.shape[0],x_tr.shape[1],1)
x_val= x_val.reshape(x_val.shape[0],x_val.shape[1],1)
print(x_tr.shape)
```

```
(4255, 168, 1)
```

```
from keras.models import Sequential
from keras.layers import *
from keras.callbacks import *
```

```
model= Sequential()
model.add(Conv1D(64, 3, padding='same', activation='relu',input_shape=(num,1)))
model.add(Conv1D(32, 5, padding='same', activation='relu',input_shape=(num,1)))
model.add(Flatten())
model.add(Dense(64,activation='relu'))
model.add(Dense(1,activation='linear'))
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_2 (Conv1D)           (None, 168, 64)           256

 conv1d_3 (Conv1D)           (None, 168, 32)           10272

 flatten_1 (Flatten)         (None, 5376)              0

 dense_4 (Dense)             (None, 64)                344128

 dense_5 (Dense)             (None, 1)                 65

=================================================================
Total params: 354721 (1.35 MB)
Trainable params: 354721 (1.35 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
#Define the optimizer and loss:
model.compile(loss='mse',optimizer='adam')
# Define the callback to save the best model during the training
mc = ModelCheckpoint('best_model.hdf5', monitor='val_loss', verbose=1,
        save_best_only=True, mode='min')
```

```
# Train the model for 30 epochs with batch size of 32:
history=model.fit(x_tr, y_tr ,epochs=30, batch_size=32, validation_data=(x_val,y_val),
            callbacks=[mc])
```

```
Epoch 1/30
131/133 [============================>.] - ETA: 0s - loss: 0.0867
Epoch 1: val_loss improved from inf to 0.03510, saving model to best_model.hdf5
133/133 [==============================] - 2s 14ms/step - loss: 0.0857 - val_loss: 0.0351
Epoch 2/30
 10/133 [=>............................] - ETA: 1s - loss: 0.0302/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3000: UserWarning: You are saving your model as an HDF5 file via `mode
  saving_api.save_model(
129/133 [============================>.] - ETA: 0s - loss: 0.0254
Epoch 2: val_loss improved from 0.03510 to 0.02028, saving model to best_model.hdf5
133/133 [==============================] - 2s 12ms/step - loss: 0.0252 - val_loss: 0.0203
Epoch 3/30
130/133 [============================>.] - ETA: 0s - loss: 0.0169
Epoch 3: val_loss improved from 0.02028 to 0.01930, saving model to best_model.hdf5
133/133 [==============================] - 2s 12ms/step - loss: 0.0171 - val_loss: 0.0193
Epoch 4/30
131/133 [============================>.] - ETA: 0s - loss: 0.0147
Epoch 4: val_loss improved from 0.01930 to 0.01536, saving model to best_model.hdf5
133/133 [==============================] - 2s 15ms/step - loss: 0.0147 - val_loss: 0.0154
Epoch 5/30
131/133 [============================>.] - ETA: 0s - loss: 0.0139
Epoch 5: val_loss did not improve from 0.01536
133/133 [==============================] - 2s 17ms/step - loss: 0.0139 - val_loss: 0.0158
Epoch 6/30
132/133 [============================>.] - ETA: 0s - loss: 0.0123
Epoch 6: val_loss improved from 0.01536 to 0.01410, saving model to best_model.hdf5
133/133 [==============================] - 2s 13ms/step - loss: 0.0123 - val_loss: 0.0141
Epoch 7/30
133/133 [==============================] - ETA: 0s - loss: 0.0110
Epoch 7: val_loss did not improve from 0.01410
133/133 [==============================] - 2s 14ms/step - loss: 0.0110 - val_loss: 0.0146
Epoch 8/30
133/133 [==============================] - ETA: 0s - loss: 0.0109
Epoch 8: val_loss did not improve from 0.01410
133/133 [==============================] - 2s 18ms/step - loss: 0.0109 - val_loss: 0.0182
Epoch 9/30
130/133 [============================>.] - ETA: 0s - loss: 0.0100
Epoch 9: val_loss did not improve from 0.01410
133/133 [==============================] - 2s 12ms/step - loss: 0.0100 - val_loss: 0.0160
Epoch 10/30
130/133 [============================>.] - ETA: 0s - loss: 0.0087
Epoch 10: val_loss did not improve from 0.01410
133/133 [==============================] - 2s 12ms/step - loss: 0.0087 - val_loss: 0.0148
Epoch 11/30
130/133 [============================>.] - ETA: 0s - loss: 0.0077
Epoch 11: val_loss did not improve from 0.01410
133/133 [==============================] - 2s 15ms/step - loss: 0.0077 - val_loss: 0.0160
```

```
Epoch 12: val_loss improved from 0.01410 to 0.01341, saving model to best_model.hdf5
133/133 [==============================] - 2s 16ms/step - loss: 0.0076 - val_loss: 0.0134
Epoch 13/30
131/133 [============================>.] - ETA: 0s - loss: 0.0066
Epoch 13: val_loss did not improve from 0.01341
133/133 [==============================] - 2s 12ms/step - loss: 0.0066 - val_loss: 0.0154
Epoch 14/30
130/133 [============================>.] - ETA: 0s - loss: 0.0059
Epoch 14: val_loss did not improve from 0.01341
```

```
model.load_weights('best_model.hdf5')
```

```
mse = model.evaluate(x_val,y_val)
print("Mean Square Error:",mse)
```

```
15/15 [==============================] - 0s 5ms/step - loss: 0.0134
Mean Square Error: 0.013408123515546322
```

```
def forecast(x_val, no_of_pred, ind):
  predictions=[]
  #intialize the array with previous weeks data
  temp=x_val[ind]
  for i in range(no_of_pred):
    #predict for the next hour
    pred=model.predict(temp.reshape(1,-1,1))[0][0]

    #append the prediction as the last element of array
    temp = np.insert(temp,len(temp),pred)
    predictions.append(pred)
    #ignore the first element of array
    temp = temp[1:]
  return predictions
```

```
no_of_pred =24
ind=72
y_pred = forecast(x_val,no_of_pred,ind)
y_true = y_val[ind:ind+(no_of_pred)]
```

```
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 15ms/step
```

```
type(y_pred)
```

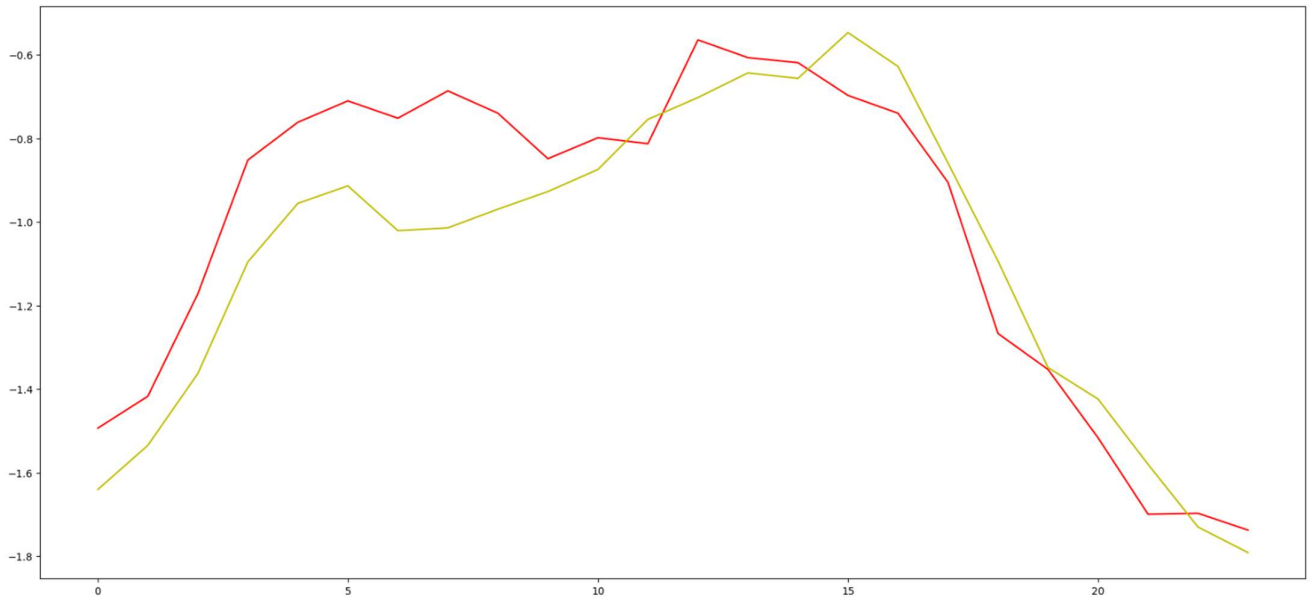
```
list
```

```
y_true = np.array(y_true)
y_pred = np.array(y_pred)
```

```
# y_true= y_scaler.inverse_transform(y_true)
# y_pred= y_scaler.inverse_transform(y_pred)

y_true = y_true.reshape(-1, 1)
y_pred = y_pred.reshape(-1, 1)
```

```
[ ]  1 def plot(y_true,y_pred):
     2   ar = np.arange(len(y_true))
     3   plt.figure(figsize=(22,10))
     4   plt.plot(ar, y_true,'r')
     5   plt.plot(ar, y_pred,'y')
     6   plt.show()
```

```
1 plot(y_true, y_pred)
```



RESULT

The implementation was successful.

Experiment 9

AIM

Implementation of multivariate forecasting using Vector AutoRegressive (VAR) model.

SOFTWARE USED

Google Colaboratory

CODE AND OUTPUT

1. Examine the data

```
# import necessary libraries (numpy, pandas, matplotlib)

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

+ Code + Text

```
from google.colab import drive
drive.mount('/content/drive/')
```

```
Drive already mounted at /content/drive/; to attempt to forcibly remount, call drive.mount("/content/drive/", force_remount=True).
```

```
#Now read the dataset

money_df = pd.read_csv('/content/drive/MyDrive/Hitesh TSA (CSE471) LAB Work/DATASETs/LAB 9/M2SLMoneyStock.csv', parse_dates = ['Date']) #read 'M2SLMoneyStock.csv'
spending_df = pd.read_csv('/content/drive/MyDrive/Hitesh TSA (CSE471) LAB Work/DATASETs/LAB 9/PCEPersonalSpending.csv', parse_dates = ['Date']) #read 'PCEPersonalSpending.csv'
df = money_df.merge(spending_df)
# df.to_csv('var_data.csv')
```

```
df
```

	Date	Money	Spending
0	1995-01-01	3492.4	4851.2
1	1995-02-01	3489.9	4850.8
2	1995-03-01	3491.1	4885.4
3	1995-04-01	3499.2	4890.2
4	1995-05-01	3524.2	4933.1
...	...	...	...
247	2015-08-01	12096.8	12394.0
248	2015-09-01	12153.8	12392.8
249	2015-10-01	12187.7	12416.1
250	2015-11-01	12277.4	12450.1
251	2015-12-01	12335.9	12469.1

252 rows × 3 columns

```
df = df.set_index('Date')
```

```
df.index = pd.to_datetime(df.index)
```

```
# preview using head
df.head()
```

Date	Money	Spending
1995-01-01	3492.4	4851.2
1995-02-01	3489.9	4850.8
1995-03-01	3491.1	4885.4
1995-04-01	3499.2	4890.2
1995-05-01	3524.2	4933.1

2.Check for Stationarity

```
# Check both the features (Money,Spending ) whether they are stationary or not. using ADF test
# perform differencing if non stationary
```

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

```
def adf_test(series,title=''):
    """
    Pass in a time series and an optional title, returns an ADF report
    """
    print(f'Augmented Dickey-Fuller Test: {title}')
    result = adfuller(series.dropna(),autolag='AIC') # .dropna() handles differenced data
    labels = ['ADF test statistic','p-value','# lags used','# observations']
    out = pd.Series(result[0:4],index=labels)
    for key,val in result[4].items():
        out[f'critical value ({key})']=val
    print(out.to_string())          # .to_string() removes the line "dtype: float64"
    if result[1] <= 0.05:
        print("Strong evidence against the null hypothesis")
        print("Reject the null hypothesis")
        print("Data has no unit root and is stationary")
    else:
        print("Weak evidence against the null hypothesis")
        print("Fail to reject the null hypothesis")
        print("Data has a unit root and is non-stationary")
```

```
adf_test(df['Money'])
```

```
Augmented Dickey-Fuller Test:
ADF test statistic        4.239022
p-value                   1.000000
# lags used               4.000000
# observations          247.000000
critical value (1%)      -3.457105
critical value (5%)      -2.873314
critical value (10%)     -2.573044
Weak evidence against the null hypothesis
Fail to reject the null hypothesis
Data has a unit root and is non-stationary
```

```
adf_test(df['Spending'])
```

```
Augmented Dickey-Fuller Test:
ADF test statistic        0.149796
p-value                   0.969301
# lags used               3.000000
# observations          248.000000
critical value (1%)      -3.456996
critical value (5%)      -2.873266
critical value (10%)     -2.573019
Weak evidence against the null hypothesis
Fail to reject the null hypothesis
Data has a unit root and is non-stationary
```

+ Code + Text

```
df_difference = df.diff()
```

```
adf_test(df_difference['Money'])

```

```
Augmented Dickey-Fuller Test:
ADF test statistic       -2.057404
p-value                   0.261984
# lags used              15.000000
# observations          235.000000
critical value (1%)      -3.458487
critical value (5%)      -2.873919
critical value (10%)     -2.573367
Weak evidence against the null hypothesis
Fail to reject the null hypothesis
Data has a unit root and is non-stationary
```

```
adf_test(df_difference['Spending'])

```

```
Augmented Dickey-Fuller Test:
ADF test statistic     -7.226974e+00
p-value                 2.041027e-10
# lags used             2.000000e+00
# observations          2.480000e+02
critical value (1%)    -3.456996e+00
critical value (5%)    -2.873266e+00
critical value (10%)   -2.573019e+00
Strong evidence against the null hypothesis
Reject the null hypothesis
Data has no unit root and is stationary
```

```
df_difference['Money'] = df_difference['Money'].diff()
```

```
adf_test(df_difference['Money'])
```

```
Augmented Dickey-Fuller Test:
ADF test statistic     -7.077471e+00
p-value                 4.760675e-10
# lags used             1.400000e+01
# observations          2.350000e+02
critical value (1%)    -3.458487e+00
critical value (5%)    -2.873919e+00
critical value (10%)   -2.573367e+00
Strong evidence against the null hypothesis
Reject the null hypothesis
Data has no unit root and is stationary
```

3. Train-Test Split

```
#use the last 1 year of data as a test set (last 12 months).
df_difference = df_difference.dropna()
```

```
test_obs = 12
train = df_difference[:-test_obs]
test = df_difference[-test_obs:]
```

4. Grid Search for Order P

```
[ ] 1 from statsmodels.tsa.api import VAR
    2
```

+ Code + Text

```
[ ] 1 for i in [1,2,3,4,5,6,7,8,9,10]:
    2     model = VAR(train)
    3     results = model.fit(i)
    4     print('Order =', i)
    5     print('AIC: ', results.aic)
    6     print('BIC: ', results.bic)
    7     print()
```

```
Order = 1
AIC:  13.893730044437806
BIC:  13.981529035352619

Order = 2
AIC:  13.788978767127503
BIC:  13.935751301238756

Order = 3
AIC:  13.800828407726483
BIC:  14.00693137452656

Order = 4
AIC:  13.705254945442732
BIC:  13.971048877393326

Order = 5
AIC:  13.702894884622165
BIC:  14.028744008993176

Order = 6
AIC:  13.716410512160271
BIC:  14.102682803812531

Order = 7
AIC:  13.722469221920354
BIC:  14.169536457053054
```

```
[ ] Order = 8
AIC:  13.72242698338018
BIC:  14.230664794264479

Order = 9
AIC:  13.706168588518498
BIC:  14.275956519239289

Order = 10
AIC:  13.744769881188871
BIC:  14.37649144441732
```

5. Fit VAR(5) Model

```
[ ] 1 #fit the model and check the summary
    2 result = model.fit(5)
    3 result.summary()
```

```
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Thu, 12, Oct, 2023
Time:                     05:17:04
--------------------------------------------------------------------
No. of Equations:         2.00000    BIC:                    14.0287
Nobs:                     233.000    HQIC:                   13.8343
Log likelihood:          -2235.61    FPE:                    893620.
AIC:                      13.7029    Det(Omega_mle):         814863.
--------------------------------------------------------------------
Results for equation Money
================================================================================
                   coefficient       std. error           t-stat            prob
--------------------------------------------------------------------------------
const             -1.884757         3.499904           -0.539           0.590
L1.Money          -0.643407         0.068285           -9.422           0.000
L1.Spending       -0.087249         0.054567           -1.599           0.110
L2.Money          -0.474297         0.075784           -6.259           0.000
L2.Spending       -0.054360         0.053759           -1.011           0.312
L3.Money          -0.211375         0.079796           -2.649           0.008
L3.Spending        0.046893         0.053758            0.872           0.383
L4.Money          -0.275523         0.074641           -3.691           0.000
L4.Spending        0.174997         0.053884            3.248           0.001
L5.Money          -0.150267         0.066905           -2.246           0.025
L5.Spending       -0.002255         0.055359           -0.041           0.968
================================================================================
Results for equation Spending
================================================================================
                   coefficient       std. error           t-stat            prob
--------------------------------------------------------------------------------
const             21.154909         4.395485            4.813           0.000
L1.Money           0.181967         0.085758            2.122           0.034
L1.Spending        0.006734         0.068530            0.098           0.922
L2.Money           0.094763         0.095176            0.996           0.319
L2.Spending        0.159329         0.067515            2.360           0.018
L3.Money          -0.009984         0.100215           -0.100           0.921
L3.Spending        0.144107         0.067514            2.134           0.033
L4.Money          -0.173935         0.093740           -1.855           0.064
L4.Spending        0.050403         0.067672            0.745           0.456
L5.Money          -0.111008         0.084025           -1.321           0.186
L5.Spending       -0.054571         0.069525           -0.785           0.432
================================================================================

Correlation matrix of residuals
               Money  Spending
Money       1.000000 -0.246978
Spending   -0.246978  1.000000
```

6 Predict Test Data

```
# VAR .forecast() function requires that we pass in a lag order number of previous observations.
lagged_Values = train.values[-8:]
pred = result.forecast(y=lagged_Values, steps=12)

idx = pd.date_range('2015-01-01', periods=12, freq='MS')
df_forecast=pd.DataFrame(data=pred, index=idx, columns=['money2d', 'spending2d'])
```

7. Invert the transformation

```
#We have to note that the forecasted value is a second-order difference. To get it similar to original data we have to roll back each difference. This can be done by taking the most recent values of the
```

```
df_forecast['Money1d'] = (df['Money'].iloc[-test_obs-1]-df['Money'].iloc[-test_obs-2]) + df_forecast['money2d'].cumsum()
df_forecast['MoneyForecast'] = df['Money'].iloc[-test_obs-1] + df_forecast['Money1d'].cumsum()

```

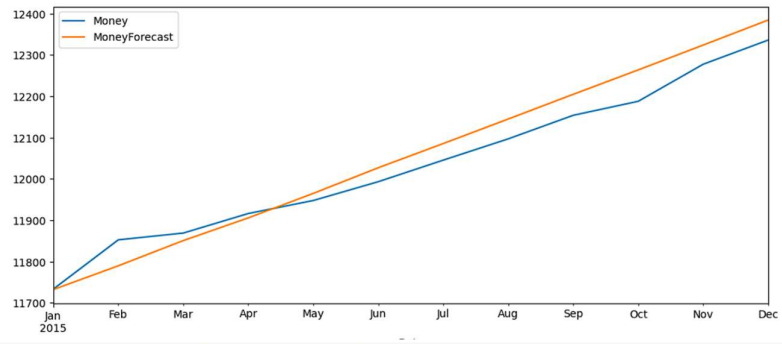
```
df_forecast['Spending1d'] = (df['Spending'].iloc[-test_obs-1]-df['Spending'].iloc[-test_obs-2]) + df_forecast['spending2d'].cumsum()
df_forecast['SpendingForecast'] = df['Spending'].iloc[-test_obs-1] + df_forecast['Spending1d'].cumsum()
```

Plot the result

+ Code + Text

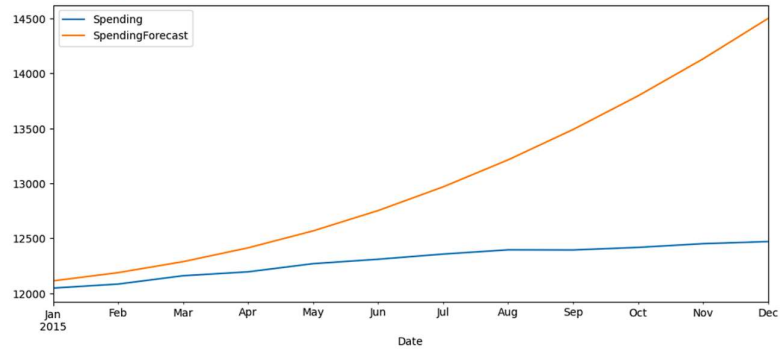
```
#plot the predicted v/s original values of 'Money' and 'Spending' for test data.

test_original = df[-test_obs:]
test_original.index = pd.to_datetime(test_original.index)
test_original['Money'].plot(figsize=(12,5),legend=True)
df_forecast['MoneyForecast'].plot(legend=True)
```

```
<Axes: xlabel='Date'>
```




```
test_original['Spending'].plot(figsize=(12,5),legend=True)
df_forecast['SpendingForecast'].plot(legend=True)
```

```
<Axes: xlabel='Date'>
```



RESULT

The implementation was done successfully.

Experiment 10

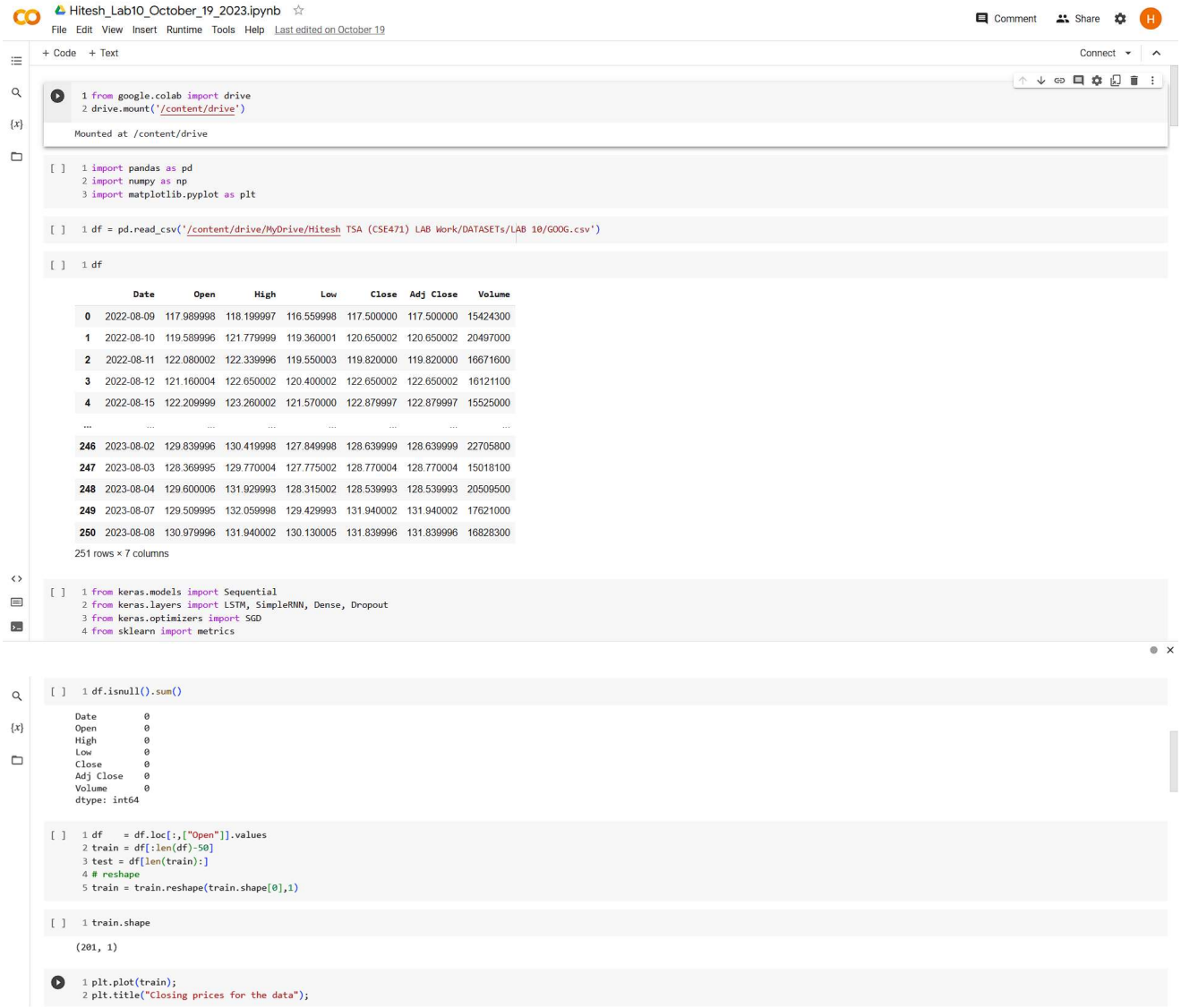
AIM

Time series forecasting using Recurrent neural network (RNN) and LSTM (Long short-term memory).

SOFTWARE USED

Google Colaboratory

CODE AND OUTPUT



Hitesh_Lab10_October_19_2023.ipynb

File Edit View Insert Runtime Tools Help Last edited on October 19

Comment Share

+ Code + Text

Connect

```
1 from google.colab import drive
2 drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
[ ] 1 import pandas as pd
    2 import numpy as np
    3 import matplotlib.pyplot as plt
```

```
[ ] 1 df = pd.read_csv('/content/drive/MyDrive/Hitesh TSA (CSE471) LAB Work/DATASETs/LAB 10/GOOG.csv')
```

```
[ ] 1 df
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2022-08-09	117.989998	118.199997	116.559998	117.500000	117.500000	15424300
1	2022-08-10	119.589996	121.779999	119.360001	120.650002	120.650002	20497000
2	2022-08-11	122.080002	122.339996	119.550003	119.820000	119.820000	16671600
3	2022-08-12	121.160004	122.650002	120.400002	122.650002	122.650002	16121100
4	2022-08-15	122.209999	123.260002	121.570000	122.879997	122.879997	15525000
...	...	...	...	...	...	...	...
246	2023-08-02	129.839996	130.419998	127.849998	128.639999	128.639999	22705800
247	2023-08-03	128.369995	129.770004	127.775002	128.770004	128.770004	15018100
248	2023-08-04	129.600006	131.929993	128.315002	128.539993	128.539993	20509500
249	2023-08-07	129.509995	132.059998	129.429993	131.940002	131.940002	17621000
250	2023-08-08	130.979996	131.940002	130.130005	131.839996	131.839996	16828300

251 rows × 7 columns

```
[ ] 1 from keras.models import Sequential
    2 from keras.layers import LSTM, SimpleRNN, Dense, Dropout
    3 from keras.optimizers import SGD
    4 from sklearn import metrics
```

```
[ ] 1 df.isnull().sum()
```

```
Date         0
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

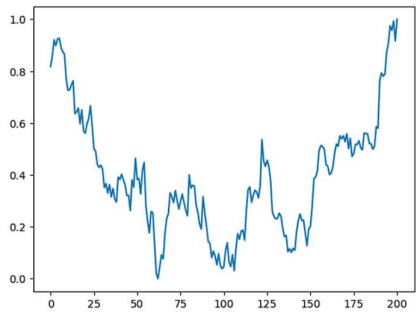
```
[ ] 1 df    = df.loc[:,["Open"]].values
    2 train = df[:len(df)-50]
    3 test = df[len(train):]
    4 # reshape
    5 train = train.reshape(train.shape[0],1)
```

```
[ ] 1 train.shape
```

```
(201, 1)
```

```
1 plt.plot(train);
2 plt.title("Closing prices for the data");
```


```
[ ]  1 from sklearn.preprocessing import MinMaxScaler
     2 scaler = MinMaxScaler(feature_range= (0,1)) # defining of Scaler
     3 train_scaled = scaler.fit_transform(train) # applying to Scaler to train
     4
     5 plt.plot(train_scaled)
     6 plt.show()
```




```
[ ]  1 # We add first 50 location to "X_train" and we 51. location to "y_train" .
     2 X_train = []
     3 y_train = []
     4 timesteps = 50
     5
     6 for i in range(timesteps, train_scaled.shape[0]):
     7     X_train.append(train_scaled[i-timesteps:i,0])
     8     y_train.append(train_scaled[i,0])
     9
    10 X_train, y_train = np.array(X_train), np.array(y_train)
    11
    12
    13 # Reshaping
    14 X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)  # Dimension of array is 3.
```

```
[ ]  1 # Initialising the RNN
     2 regressor = Sequential()
     3
     4 # Adding the first RNN layer and some Dropout regularisation
     5 regressor.add(SimpleRNN(units = 50,activation='tanh', return_sequences = True, input_shape = (X_train.shape[1], 1)))
     6 regressor.add(Dropout(0.2))
     7
     8 # Adding a second RNN layer and some Dropout regularisation.
     9 regressor.add(SimpleRNN(units = 50,activation='tanh', return_sequences = True))
    10 regressor.add(Dropout(0.2))
    11
    12 # Adding a third RNN layer and some Dropout regularisation.
    13 regressor.add(SimpleRNN(units = 50,activation='tanh', return_sequences = True))
    14 regressor.add(Dropout(0.2))
    15
    16 # Adding a fourth RNN layer and some Dropout regularisation.
    17 regressor.add(SimpleRNN(units = 50))
    18 regressor.add(Dropout(0.2))
    19
    20
    21 # Adding the output layer
    22 regressor.add(Dense(units = 1))
    23
    24 # Compiling the RNN
    25 regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')
    26
```

```
[ ]  27 # Fitting the RNN to the Training set
     28 regressor.fit(X_train, y_train, epochs = 100, batch_size = 32)
```

```
Epoch 1/100
5/5 [==============================] - 8s 49ms/step - loss: 0.5492
Epoch 2/100
5/5 [==============================] - 0s 48ms/step - loss: 0.3743
Epoch 3/100
5/5 [==============================] - 0s 45ms/step - loss: 0.4023
Epoch 4/100
5/5 [==============================] - 0s 67ms/step - loss: 0.2737
Epoch 5/100
5/5 [==============================] - 0s 83ms/step - loss: 0.2703
Epoch 6/100
5/5 [==============================] - 0s 90ms/step - loss: 0.2487
Epoch 7/100
5/5 [==============================] - 0s 82ms/step - loss: 0.3351
Epoch 8/100
5/5 [==============================] - 0s 78ms/step - loss: 0.2058
Epoch 9/100
5/5 [==============================] - 0s 80ms/step - loss: 0.2488
Epoch 10/100
5/5 [==============================] - 0s 87ms/step - loss: 0.3697
Epoch 11/100
5/5 [==============================] - 0s 93ms/step - loss: 0.2268
Epoch 12/100
5/5 [==============================] - 0s 91ms/step - loss: 0.2558
Epoch 13/100
5/5 [==============================] - 0s 84ms/step - loss: 0.2247
Epoch 14/100
5/5 [==============================] - 0s 86ms/step - loss: 0.2522
Epoch 15/100
5/5 [==============================] - 0s 43ms/step - loss: 0.2040
Epoch 16/100
5/5 [==============================] - 0s 41ms/step - loss: 0.2210
```

```
Epoch 17/100
5/5 [==============================] - 0s 47ms/step - loss: 0.1628
Epoch 18/100
5/5 [==============================] - 0s 43ms/step - loss: 0.1975
Epoch 19/100
5/5 [==============================] - 0s 44ms/step - loss: 0.1766
Epoch 20/100
5/5 [==============================] - 0s 46ms/step - loss: 0.1917
Epoch 21/100
5/5 [==============================] - 0s 41ms/step - loss: 0.1418
Epoch 22/100
5/5 [==============================] - 0s 46ms/step - loss: 0.1707
Epoch 23/100
5/5 [==============================] - 0s 42ms/step - loss: 0.1471
Epoch 24/100
5/5 [==============================] - 0s 45ms/step - loss: 0.1402
Epoch 25/100
5/5 [==============================] - 0s 48ms/step - loss: 0.1194
Epoch 26/100
5/5 [==============================] - 0s 50ms/step - loss: 0.1276
Epoch 27/100
5/5 [==============================] - 0s 46ms/step - loss: 0.1199
Epoch 28/100
5/5 [==============================] - 0s 42ms/step - loss: 0.1137
Epoch 29/100
5/5 [==============================] - 0s 43ms/step - loss: 0.1204
```

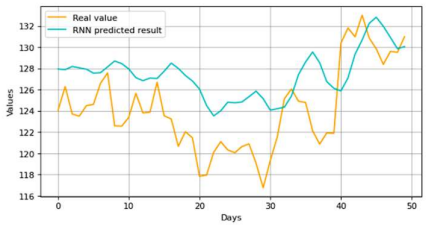
```
[ ]  1 inputs = df[len(df) - len(test) - timesteps:]
     2 inputs = scaler.transform(inputs)  # min max scaler
```

```
[ ]  1 X_test = []
     2 for i in range(timesteps, inputs.shape[0]):
     3     X_test.append(inputs[i-timesteps:i, 0]) # 0 dan 50 ye, 1 den 51 e gibi kaydirarark 50 eleman aliyoruz
     4 X_test = np.array(X_test)
     5 X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)
```

```
[ ]  1 predicted_data = regressor.predict(X_test)
     2 predicted_data = scaler.inverse_transform(predicted_data)
```

```
2/2 [==============================] - 1s 29ms/step
```

```
1 plt.figure(figsize=(8,4), dpi=80, facecolor='w', edgecolor='k')
2 plt.plot(test,color="orange",label="Real value")
3 plt.plot(predicted_data,color="c",label="RNN predicted result")
4 plt.legend()
5 plt.xlabel("Days")
6 plt.ylabel("Values")
7 plt.grid(True)
8 plt.show()
```




```
[ ]  1 from keras.models import Sequential
     2 from keras.layers import Dense
     3 from keras.layers import LSTM
     4 from sklearn.preprocessing import MinMaxScaler
     5 from sklearn.metrics import mean_squared_error
```

```
[ ]  1 model = Sequential()
     2 model.add(LSTM(10, input_shape=(None,1)))
     3 model.add(Dense(1))
     4 model.compile(loss="mean_squared_error",optimizer='Adam')
     5 model.fit(X_train,y_train,epochs=50, batch_size=1)
```

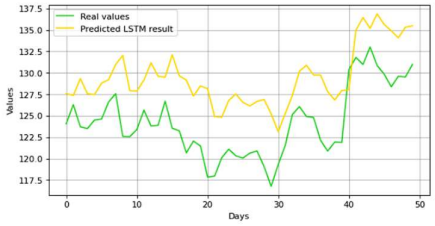
```
Epoch 1/50
151/151 [==============================] - 4s 9ms/step - loss: 0.0897
Epoch 2/50
151/151 [==============================] - 2s 15ms/step - loss: 0.0258
Epoch 3/50
151/151 [==============================] - 4s 27ms/step - loss: 0.0179
Epoch 4/50
151/151 [==============================] - 3s 21ms/step - loss: 0.0142
Epoch 5/50
151/151 [==============================] - 2s 10ms/step - loss: 0.0120
Epoch 6/50
151/151 [==============================] - 2s 10ms/step - loss: 0.0103
Epoch 7/50
151/151 [==============================] - 2s 11ms/step - loss: 0.0092
Epoch 8/50
151/151 [==============================] - 1s 9ms/step - loss: 0.0086
Epoch 9/50
151/151 [==============================] - 2s 11ms/step - loss: 0.0082
Epoch 10/50
151/151 [==============================] - 2s 13ms/step - loss: 0.0073
Epoch 11/50
151/151 [==============================] - 4s 25ms/step - loss: 0.0071
Epoch 12/50
151/151 [==============================] - 2s 12ms/step - loss: 0.0064
Epoch 13/50
151/151 [==============================] - 1s 9ms/step - loss: 0.0063
Epoch 14/50
151/151 [==============================] - 1s 9ms/step - loss: 0.0058
Epoch 15/50
151/151 [==============================] - 1s 9ms/step - loss: 0.0057
Epoch 16/50
151/151 [==============================] - 2s 10ms/step - loss: 0.0053
Epoch 17/50
```

```
Epoch 18/50
151/151 [==============================] - 2s 13ms/step - loss: 0.0053
Epoch 19/50
151/151 [==============================] - 2s 16ms/step - loss: 0.0049
Epoch 20/50
151/151 [==============================] - 2s 11ms/step - loss: 0.0049
Epoch 21/50
151/151 [==============================] - 1s 9ms/step - loss: 0.0047
Epoch 22/50
151/151 [==============================] - 1s 10ms/step - loss: 0.0046
Epoch 23/50
151/151 [==============================] - 1s 9ms/step - loss: 0.0044
Epoch 24/50
151/151 [==============================] - 2s 11ms/step - loss: 0.0043
Epoch 25/50
151/151 [==============================] - 2s 10ms/step - loss: 0.0043
Epoch 26/50
151/151 [==============================] - 2s 11ms/step - loss: 0.0042
Epoch 27/50
151/151 [==============================] - 4s 29ms/step - loss: 0.0042
Epoch 28/50
151/151 [==============================] - 2s 10ms/step - loss: 0.0038
Epoch 29/50
151/151 [==============================] - 1s 9ms/step - loss: 0.0039
```

```
predicted_data2=model.predict(X_test)
predicted_data2=scaler.inverse_transform(predicted_data2)
```

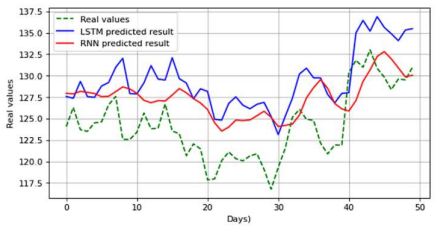
```
2/2 [==============================] - 0s 9ms/step
```

```
plt.figure(figsize=(8,4), dpi=80, facecolor='w', edgecolor='k')
plt.plot(test,color="LimeGreen",label="Real values")
plt.plot(predicted_data2,color="Gold",label="Predicted LSTM result")
plt.legend()
plt.xlabel("Days")
plt.ylabel("Values")
plt.grid(True)
plt.show()
```



+ Code + Text

```
plt.figure(figsize=(8,4), dpi=80, facecolor='w', edgecolor='k')
plt.plot(test,color="green", linestyle='dashed',label="Real values")
plt.plot(predicted_data2,color="blue", label="LSTM predicted result")
plt.plot(predicted_data,color="red",label="RNN predicted result") # ben ekledim
plt.legend()
plt.xlabel("Days)")
plt.ylabel("Real values")
plt.grid(True)
plt.show()
```



RESULT

The implementation was done successfully.