Report on Mini Project

Time Series Analysis (DJ19DSC5012)
AY: 2022-23

Milk Production Analysis

Dev Patel (60009200016)
Mitali Chotalia (60009200017)
Alistair Saldanha (60009200024)

Guided By
Prof. Pradnya Saval

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Time series analysis is a specific way of analyzing a sequence of data points collected over an interval of time.

This project is intended to be a concise report to explain the Time Series analysis performed on a dataset containing information about Milk Production

The idea is to analyze the trend and seasonality of the production of Milk from 1962 until 1975, decomposing the time series and analyzing the remainder.

CHAPTER 2. DATA DESCRIPTION

The Milk production dataset has been selected from Kaggle. It is a time series data which contains the monthly production of milk in pounds per cow starting in January, 1962 and ending in December, 1975. The time series contains 169 observations, for each month between the said period. The dataset contains 2 columns:

1. Month: includes the month and the year of the observation
2. Monthly milk production: pounds per cow: pounds of milk produced at the center per cow in that month.

CHAPTER 3. OBJECTIVE

The main objective of using Milk production dataset is to apply various Time series models on it.

We can use these models on the Milk Production dataset to forecast future values ahead of the time series i.e. future values of monthly milk produced per cow can be forecasted. We can also analyze the

pattern of production throughout the year and make required inferences.

CHAPTER 4. DATA CLEANING

The Milk Production dataset was mostly clean. The column 'Monthly milk production: pounds per cow' had one null value which was dropped. Its column name was changed to 'Milk_prod' for convenience. Further we carried out data wrangling for deeper insights on our data and started off our preparation for time series analysis.

After the process, we were left with 168 data points out of which the first 138 were used for training of the models and the last 30 were used for testing purposes.

Chapter 4: Data Cleaning and Wrangling

CHAPTER 5. DATA DECOMPOSITION

Decomposition is a statistical method that involves breaking down the Time Series data into many components or identifying seasonality and trend from a series of data.

Two models for decomposition are

1. Additive Model
2. Multiplicative Model

On our dataset, we have applied

1. Multiple Box Plot
2. Seasonality using Auto Correlation
3. Deseasoning the Time Series

4. Seasonal Decomposition

Chapter 5: Data Decomposition

CHAPTER 6. SMOOTHING METHODS

Smoothing is a statistical method we can use to create an approximation function to remove irregularities in data and attempt to capture significant patterns.

There are three smoothing methods:

1. **Simple Exponential Smoothing:**

   Simple Exponential Smoothing (SES) is one of the minimal models of the exponential smoothing algorithm.

   SES technique is employed for univariate data having no clear trends or seasonal patterns.

   SES is used to forecast future values by using the weighted average of all the previous values in the series.

   SES method is used to predict series that do not have trends or seasonality.

2. **Double Exponential Smoothing:**

   Double Exponential Smoothing is also called 'Holt's Exponential Smoothing.'

   This is a more reliable method for handling data that consumes trend components without seasonality.

   The basic concept is to use SES and advance it to capture trend components.

3. Triple Exponential Smoothing:

Triple Exponential Smoothing is also called 'Holt - Winter Method.'

Triple Exponential Smoothing for the data which consumes trend and seasonality over time.

It includes all smoothing component equations such as trends and seasonality.

Chapter 6: Smoothing Methods

CHAPTER 7. TESTING STATIONARITY

Stationarity is an important concept in the field of time series analysis with tremendous influence on how the data is perceived and predicted. When forecasting or predicting the future, most time series models assume that each point is independent of one another. For some models like AR, MA, ARIMA, we need to have stationary or trend stationary data in order to make appropriate analysis and forecasts. In statistics, a unit root test tests whether a time series variable is non-stationary and possesses a unit root. The Null Hypothesis is generally defined as the presence of a unit root and Alternative Hypothesis is that the data is either stationary or trend stationary. The function adfuller() in statsmodel library of python can perform ADF Test on any time series data and provides the following information:

1. Test statistic
2. P-value
3. Number of lags used
4. Number of observations used
5. Critical values of 1%, 5% and 10%

To reject the null hypothesis and say that the data is stationary (no unit roots),

The test statistic must be less than critical values (more negative) AND p-value should be less than 0.05.

Thus, we test the stationarity of our dataset in this chapter using the above method.

We observed that the original dataset was not stationary and hence, the first differencing was applied, after which the stationarity conditions were satisfied.

Before applying first differencing:

```
Results of Dickey-Fuller Test for column: Milk_Production
Test Statistic                 -1.303812
p-value                         0.627427
No Lags Used                   13.000000
No of Observations Used       154.000000
Critical Value (1%)            -3.473543
Critical Value (5%)            -2.880498
Critical Value (10%)           -2.576878
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary
```

After applying first differencing:

```
Results of Dickey-Fuller Test for column: Milk Production
Test Statistic                 -3.054996
p-value                         0.030068
No Lags Used                   14.000000
No of Observations Used       152.000000
Critical Value (1%)            -3.474121
Critical Value (5%)            -2.880750
Critical Value (10%)           -2.577013
dtype: float64
Conclusion:====>
Reject the null hypothesis
Data is stationary
```

Chapter 7: Testing Stationarity

CHAPTER 8. JUSTIFICATION WHY IT IS A TIME SERIES PROBLEM.

The data points in our dataset consist of successive measurements made from the same source over a fixed time interval and are used to track change over time. As we have monthly readings of the amount of milk produced by the center in pounds. These successive measurements are used to analyze how the values change; hence, this is a time series problem.

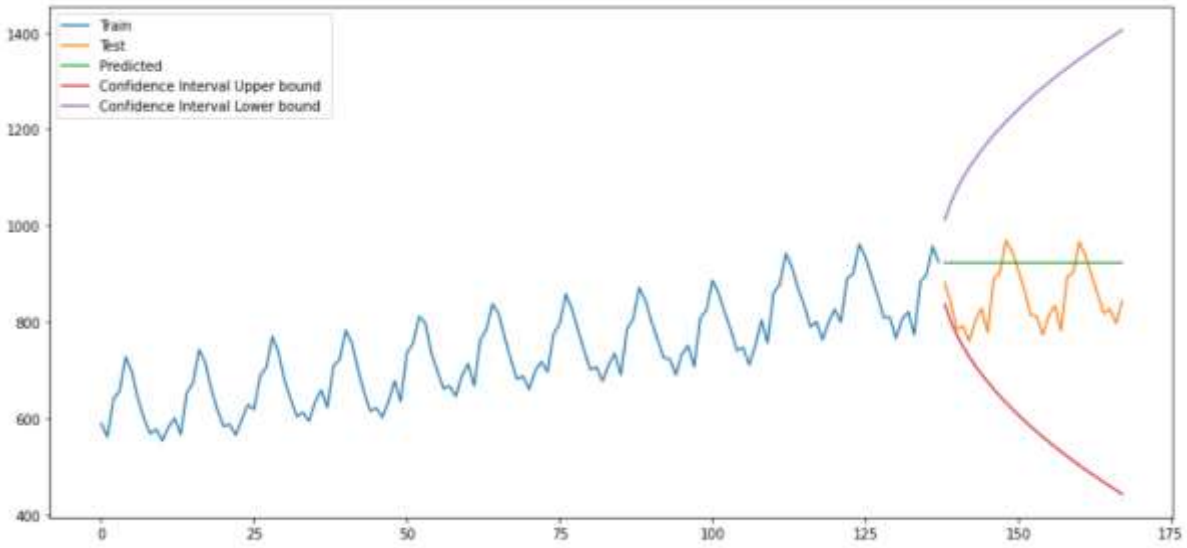
CHAPTER 9. IMPLEMENTATION AND INTERPRETATION FOR FORECAST

We implemented several models on our data to find out the best method for analysis using several types of errors. The models which were considered and implemented are:

1. ARIMA

The objective of the ARIMA model is to predict the future time series by examining the differences between values in the series instead of actual values. ARIMA models are applied in the cases where the data shows evidence of non-stationarity. As seen in Chapter 7, the dataset was non-stationary and hence ARIMA model seemed to be an appropriate choice for the same.

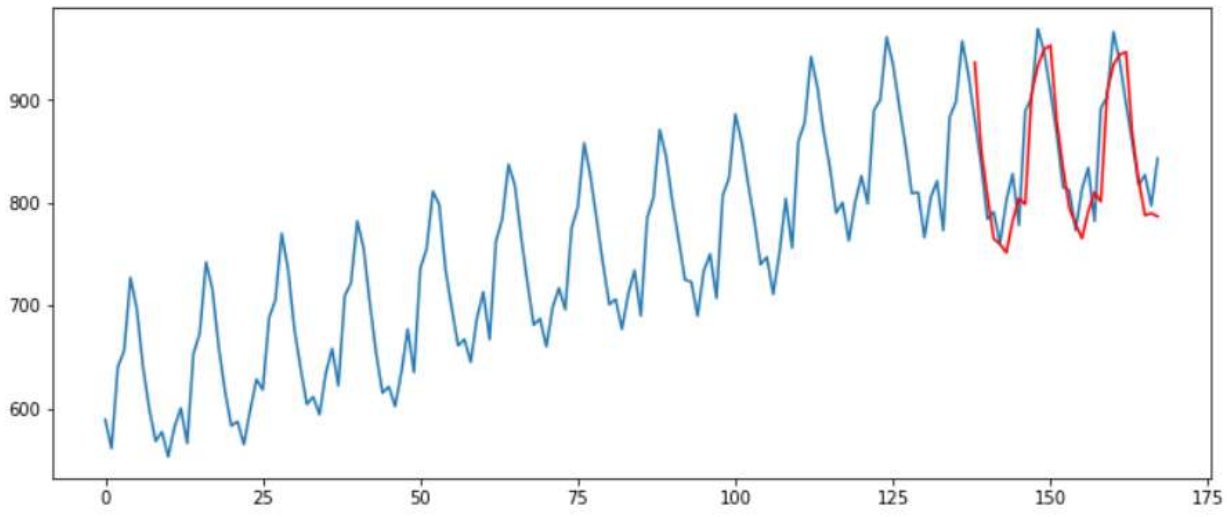
Graph depicting the forecasted values for ARIMA Model is as shown below:



2. CNN

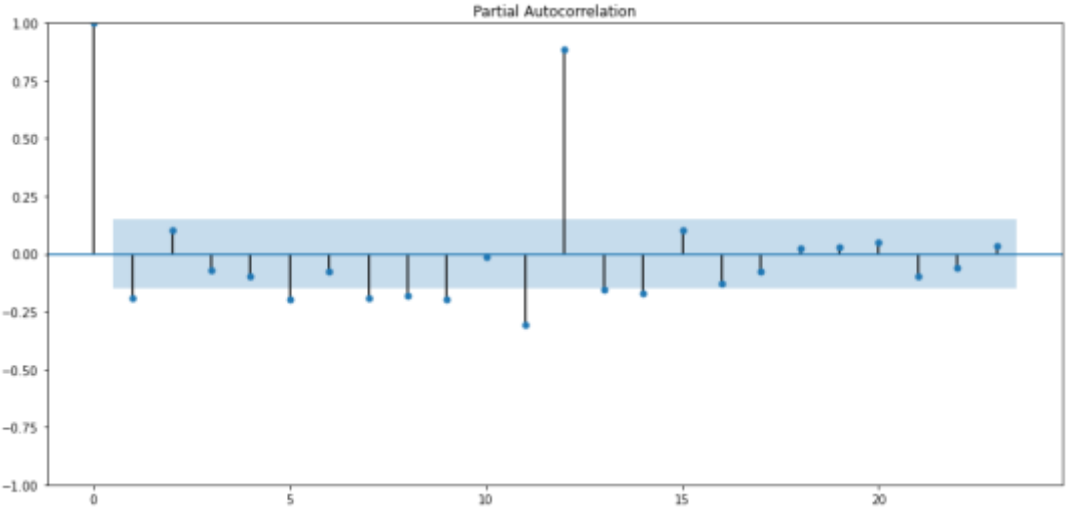
Convolutional neural networks have their roots in image processing. However, convolutional neural networks are not limited to handling images. We can also use CNN for time series prediction. We can predict the next value by using past values.

Graph depicting forecasted values for CNN Model is:



3. GARCH

GARCH (Generalized Auto-Regressive Conditional Heteroskedasticity) is used for a dataset with volatility (variable variance). We can use the PACF plot to check for volatility and the order of our GARCH model.



However, after seeing the PACF of our GARCH model we saw that there was no volatility present in our dataset as the order of GARCH came out to be (0,0). Hence, this model is not appropriate for our dataset.

4. VAR

VAR (Vector Auto-Regression) is a multivariate forecasting algorithm that is used when two or more time series influence each other. The basic requirements in order to use VAR are

1. Need at least 2 time series variables.
2. The time series should influence each other.

The first requirement itself is not fulfilled as we only have 1 variable in our dataset. Hence this model is not appropriate for our dataset.

Chapter 9: Implementation and interpretation for forecast

CHAPTER 10. REASONS FOR SELECTING THE TIME SERIES MODEL

1. Errors calculated using CNN are less as compared to other models.
2. CNNs also tend to be more computationally efficient.

CHAPTER 11. COMPARATIVE RESULT ANALYSIS

Milk Production Dataset can be modelled appropriately using **ARIMA** and **CNN**.

The forecasted values for ARIMA Model came out as follows. It gave a constant value of 924.0 for all readings. In hindsight, as our model had seasonality, ARIMA is incompetent to handle it, hence, the results.

new_index	
138	924.0
139	924.0
140	924.0
141	924.0
142	924.0
143	924.0
144	924.0
145	924.0
146	924.0
147	924.0
148	924.0
149	924.0
150	924.0

The forecasted values for CNN Model are comparatively a lot closer:

	Actual Values	Predicted values
138	881.0	936.11865
139	837.0	851.89230
140	784.0	806.44950
141	791.0	764.91800
142	760.0	759.48290
143	802.0	751.24630
144	828.0	782.10986
145	778.0	804.20420
146	889.0	798.26450
147	902.0	905.50560
148	969.0	933.19794
149	947.0	949.13770
150	908.0	952.92980

Using the actual and predicted values as above, we can distinctly infer that CNN is the most suitable Model for Milk Production Dataset.

Evaluation Metrics for both the Models is tabulated as shown below:

Model	**MSE**	**MAE**	**RMSE**	**MAPE**	**R2**
ARIMA	8889.4666	81.93333	94.2839	10.0340	-1.5745
CNN	1462.8468	29.7907	38.2471	3.6025	0.6796

RMSE Error when the dataset is modelled using ARIMA is 94.2839 and when modelled using CNN is 38.2471.

Evaluation Metrics evidently describes **CNN** as the most suitable model for our Milk Production Dataset. The graphs showing the forecasted values in Chapter 9, too, clearly suggest that CNN is the better model among the two.

CHAPTER 12. CONCLUSION

As inferred from comparative result analysis, CNN is the most suitable model for the Milk Production Dataset which gives the least RMSE error when compared to other models which are applicable for this dataset.

CNN can also be used to forecast future values ahead of the time series.

In the Milk Production Dataset, we can efficiently use CNN to predict future values of milk produced per cow monthly.

CHAPTER 13. FUTURE SCOPE

1. **Forecasting as a Service**

   As expert practitioners for Time Series forecasting are quite less, there are drives to develop time series analysis and forecasting as a service which can be easily done in an efficient way.

   For example, Amazon recently rolled out a time series prediction service where the model frames forecasting as just one step in a data pipeline.

2. **Deep Learning enhances probabilistic possibilities**

   In the past few years, many of the largest tech companies have made some information public about how they do their own forecasting for their most important services. In these cases, where the quality of the forecast is very important, companies are often giving indications that they are using deep learning with a probabilistic component.

For example, Uber has blogged about forecasting demand for car rides, and Amazon has developed a well-regarded autoregressive recurrent neural network for making predictions about product demands.

3. Increasing Combination of Statistical and Machine Learning Methodologies

Several indications point toward combining machine learning and statistical methodologies rather than simply searching for the "best" method for forecasting. This is an extension of increasing acceptance and use of ensemble methods for forecasting.

CHAPTER 14. REFERENCES

1. Milk Production Dataset:

   https://www.kaggle.com/code/bhavinmoriya/milk-production-time-series-lstm/data

2. Importance of Stationarity

   https://towardsdatascience.com/why-does-stationarity-matter-in-time-series-analysis-e2fb7be74454

3. Future Scope

   https://medium.com/oreillymedia/the-future-of-time-series-forecasting-bd83c2aca9a8

In [ ]:

```
import pandas as pd
!pip install pandasql
from pandasql import sqldf
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
from pandas.plotting import autocorrelation_plot
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.filters.hp_filter import hpfilter
import statsmodels.api as sm
from pmdarima import auto_arima
import warnings
warnings.filterwarnings("ignore")
%matplotlib inline
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publi
c/simple/
Requirement already satisfied: pandasql in /usr/local/lib/python3.8/dist-packages (0.7.3)
Requirement already satisfied: pandas in /usr/local/lib/python3.8/dist-packages (from pan
dasql) (1.3.5)
Requirement already satisfied: sqlalchemy in /usr/local/lib/python3.8/dist-packages (from
pandasql) (1.4.44)
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packages (from pand
asql) (1.21.6)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.8/dist-packages (fr
om pandas->pandasql) (2022.6)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.8/dist-pa
ckages (from pandas->pandasql) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-packages (from p
ython-dateutil>=2.7.3->pandas->pandasql) (1.15.0)
Requirement already satisfied: greenlet!=0.4.17 in /usr/local/lib/python3.8/dist-packages
(from sqlalchemy->pandasql) (2.0.1)
```

Chapter 4: Data Cleaning and Wrangling

In [ ]:

```
df = pd.read_csv('/content/drive/MyDrive/Time Series Analysis/monthly-milk-production-pou
nds.csv')
df.head()
```

Out[ ]:

	Month	Monthly milk production: pounds per cow
0	1962-01	589.0
1	1962-02	561.0
2	1962-03	640.0
3	1962-04	656.0
4	1962-05	727.0

In [ ]:

```
df = df.rename(columns = {'Monthly milk production: pounds per cow': 'Milk_Prod'})
```

In [ ]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170 entries, 0 to 169
Data columns (total 2 columns):
```

```
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Month       169 non-null    object
 1   Milk_Prod   168 non-null    float64
dtypes: float64(1), object(1)
memory usage: 2.8+ KB
```

In [ ]:

```
df.isnull().sum()
```

Out[ ]:

```
Month        1
Milk_Prod    2
dtype: int64
```

In [ ]:

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

In [ ]:

```
df.isnull().sum()
```

Out[ ]:

```
Month        0
Milk_Prod    0
dtype: int64
```

In [ ]:

```
df.describe()
```

Out[ ]:

	Milk_Prod
count	168.000000
mean	754.708333
std	102.204524
min	553.000000
25%	677.750000
50%	761.000000
75%	824.500000
max	969.000000

In [ ]:

```
Query_string = """ select * from df where Milk_Prod > 500 ORDER BY Milk_Prod DESC"""
sqldf(Query_string, globals())
```

Out[ ]:

	Month	Milk_Prod
0	1974-05	969.0
1	1975-05	966.0
2	1972-05	961.0
3	1973-05	957.0
4	1974-06	947.0
...	...	...
163	1962-09	568.0

164	1962-12	798.0
165	1963-11	565.0
166	1962-02	561.0
167	1962-11	553.0

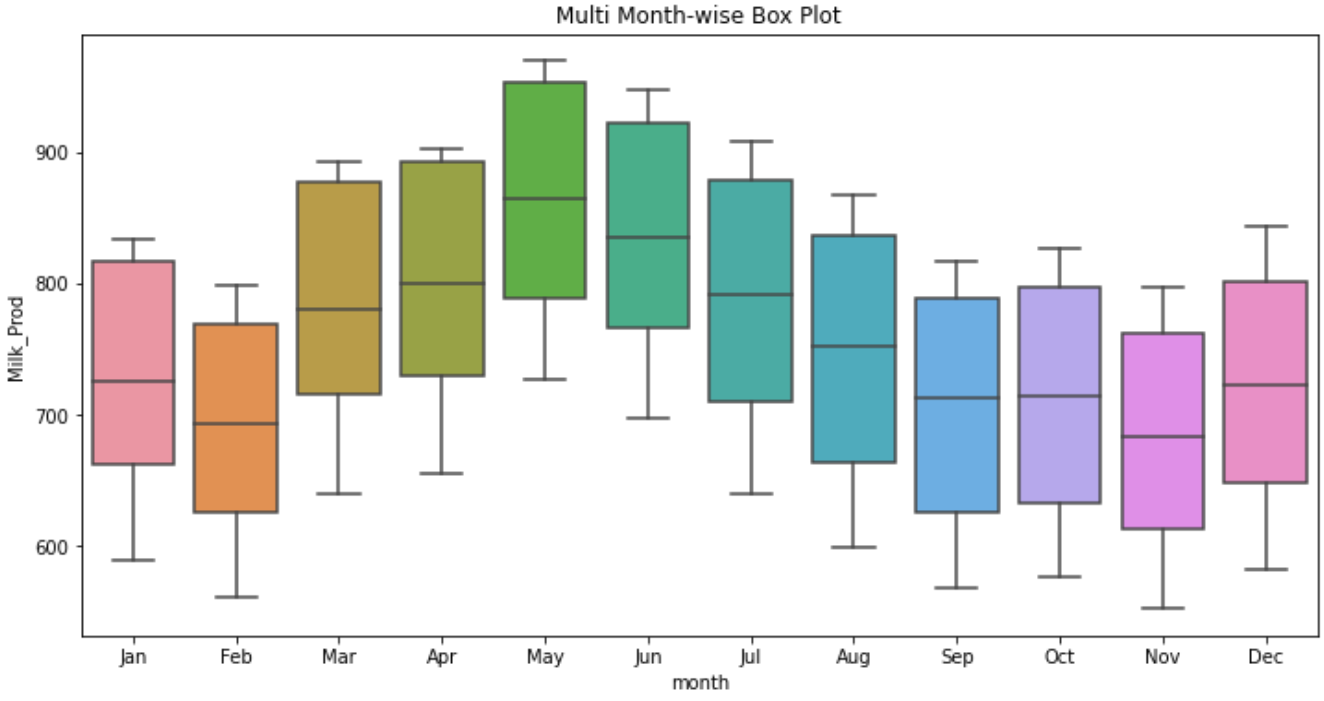
168 rows × 2 columns

Chapter 5: Data Decomposition

1) Multiple Box Plot

In [ ]:

```
df['Month'] = pd.to_datetime(df['Month'], errors='coerce')
df['month'] = df['Month'].dt.strftime('%b')

df['year'] = [d.year for d in df.Month]

df['month'] = [d.strftime('%b') for d in df.Month]

years = df['year'].unique()
plt.figure(figsize=(12,6))
sns.boxplot(x='month', y='Milk_Prod', data=df).set_title("Multi Month-wise Box Plot")
plt.show()
```



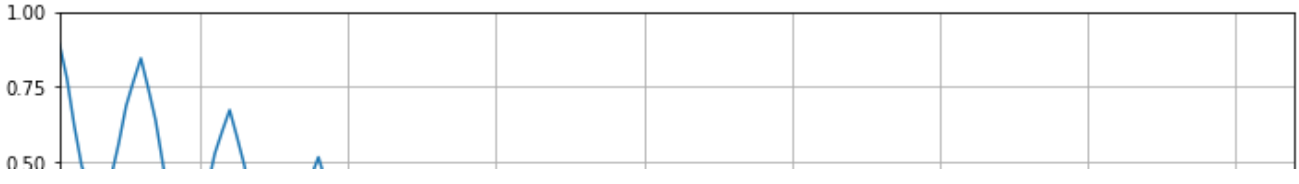
2) Seasonality using auto correlation

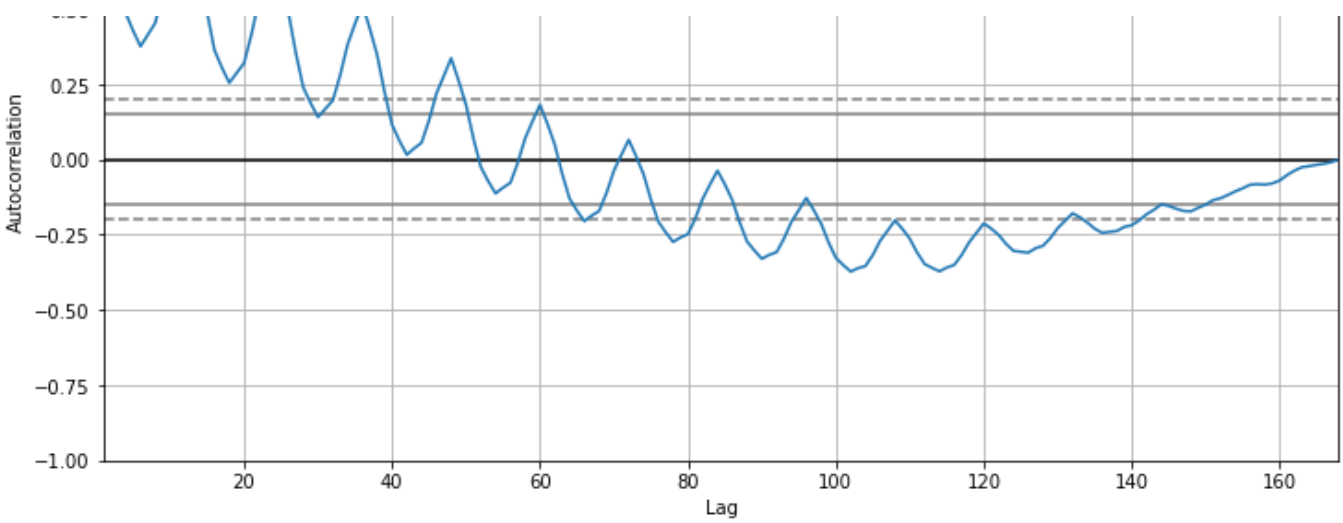
In [ ]:

```
plt.rcParams.update({'figure.figsize':(12,6)})
autocorrelation_plot(df.Milk_Prod.tolist())
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0e87e7aa30>
```



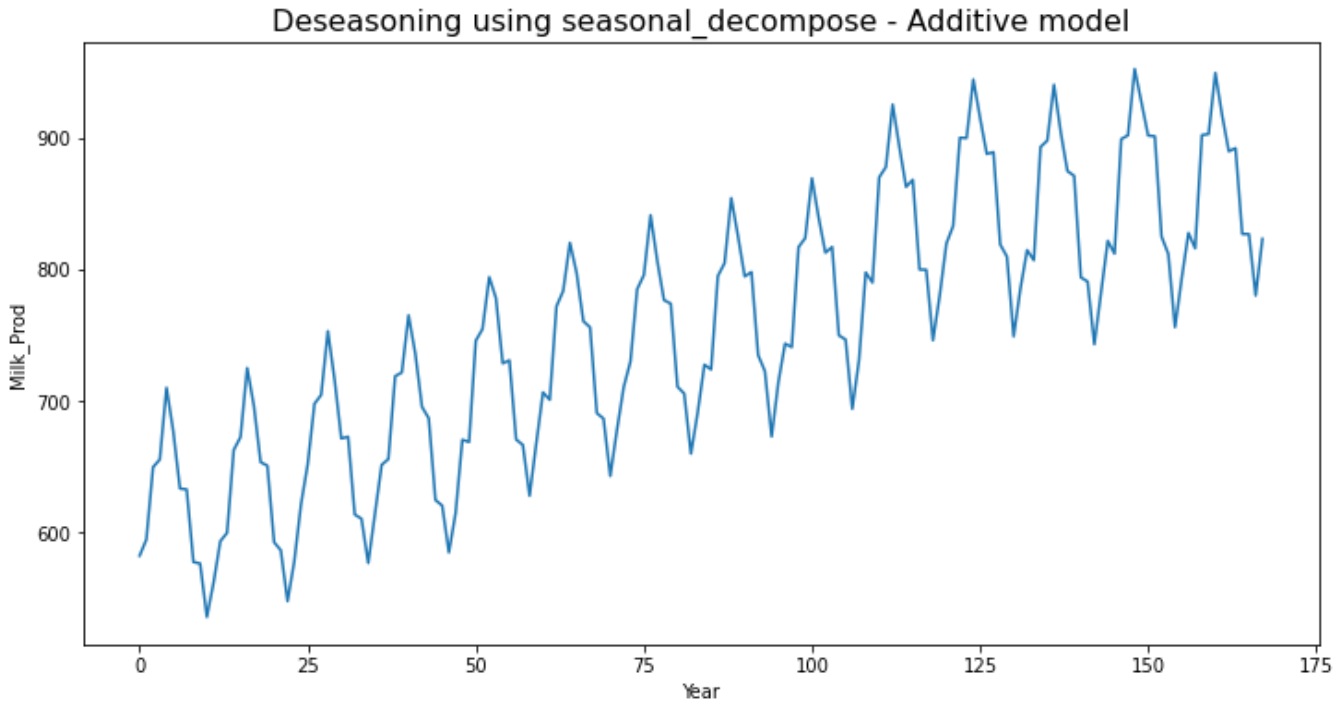


3) Deseasoning the time series

a) Additive model

In [ ]:

```
result_mul = seasonal_decompose(df['Milk_Prod'], model='additive', freq = 6)
deseason = df['Milk_Prod'] - result_mul.seasonal
plt.figure(figsize=(12,6))
plt.plot(deseason)
plt.title('Deseasoning using seasonal_decompose - Additive model', fontsize=16)
plt.xlabel('Year')
plt.ylabel('Milk_Prod')
plt.show()
```

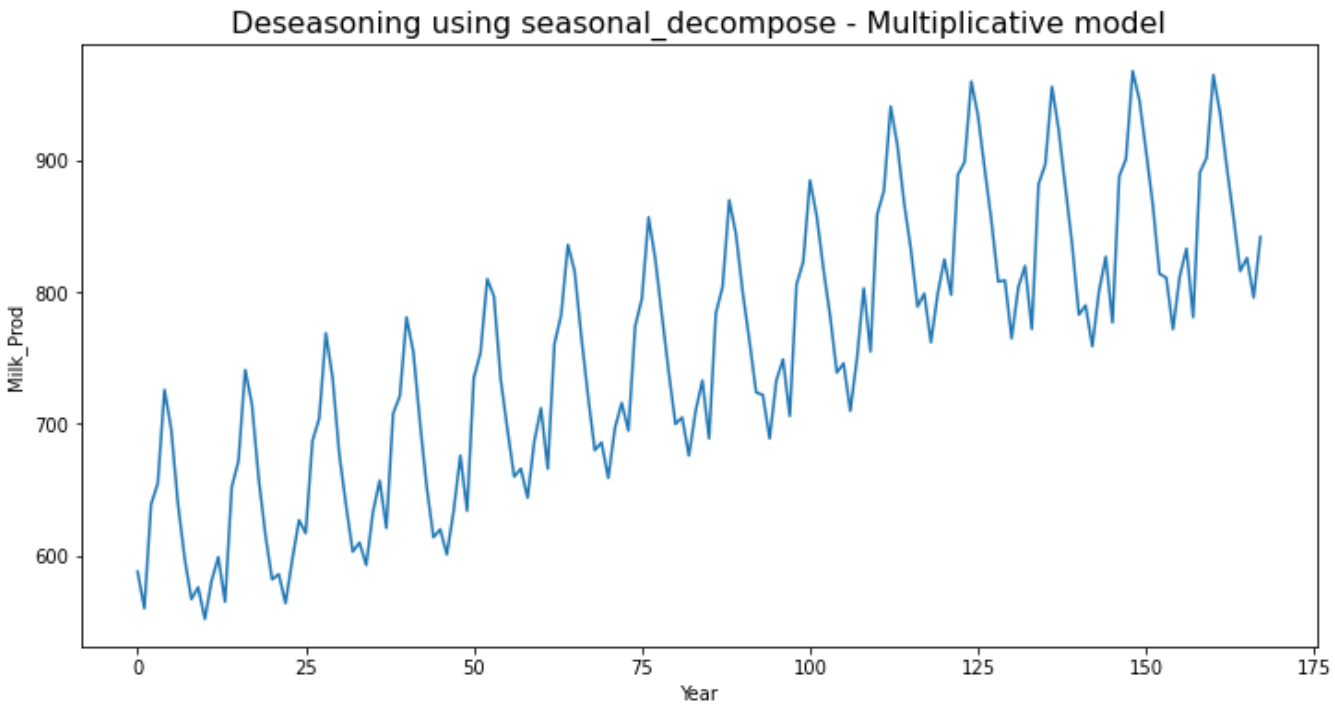


b) Multiplicative model

In [ ]:

```
result_mul = seasonal_decompose(df['Milk_Prod'], model='multiplicative', freq = 6)
deseason = df['Milk_Prod'] - result_mul.seasonal
plt.figure(figsize=(12,6))
plt.plot(deseason)
plt.title('Deseasoning using seasonal_decompose - Multiplicative model', fontsize=16)
plt.xlabel('Year')
plt.ylabel('Milk_Prod')
```

```
plt.show()
```



4) Seasonal Decomposition

a) From Scratch

In [ ]:

```
dtemp = df["Milk_Prod"]
dtemp.head()
```

Out[ ]:

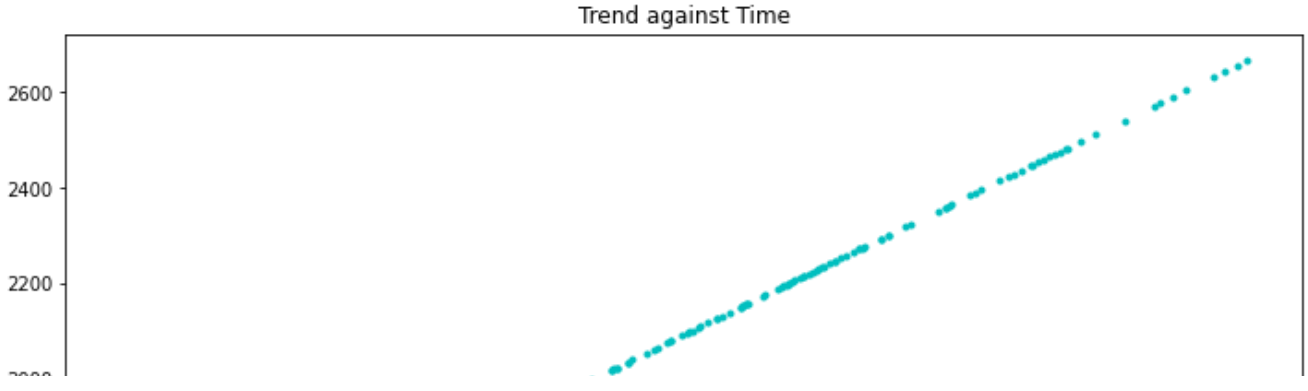
```
0    589.0
1    561.0
2    640.0
3    656.0
4    727.0
Name: Milk_Prod, dtype: float64
```

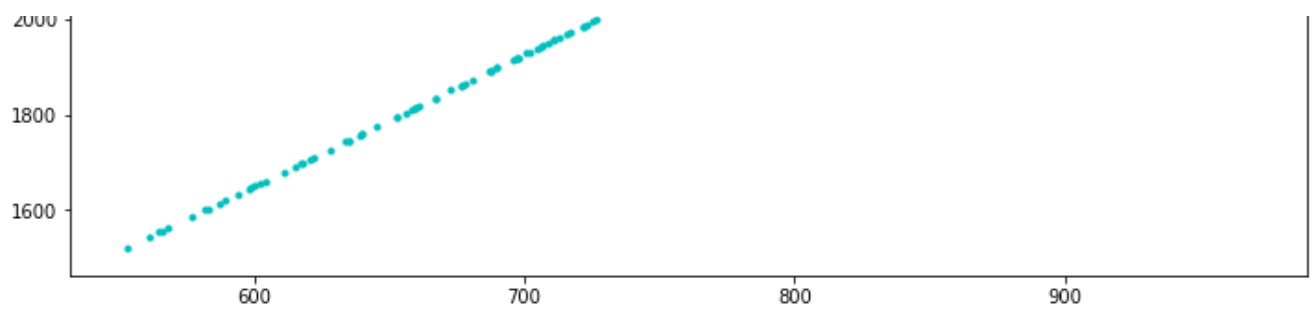
In [ ]:

```
#create the trend component
T_Series = dtemp
Trend = T_Series * 2.75

#plot the trend
plt.plot(T_Series, Trend, 'c.')
plt.title("Trend against Time")
```

Out[ ]:

```
Text(0.5, 1.0, 'Trend against Time')
```



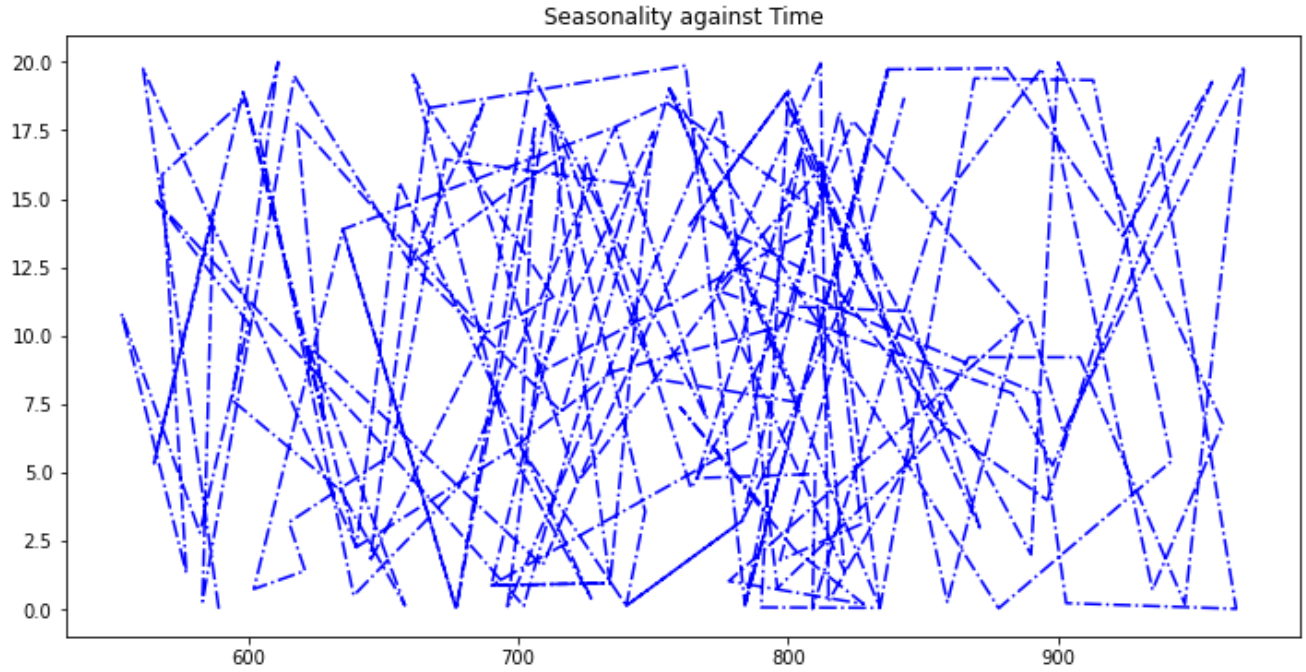


In [ ]:

```
# creating the seasonality component
seasonality = 10 + np.sin(T_Series) * 10

#plot the seasonality trend
plt.plot(T_Series,  seasonality, 'b-.')
plt.title("Seasonality against Time")
```

Out[ ]:

```
Text(0.5, 1.0, 'Seasonality against Time')
```

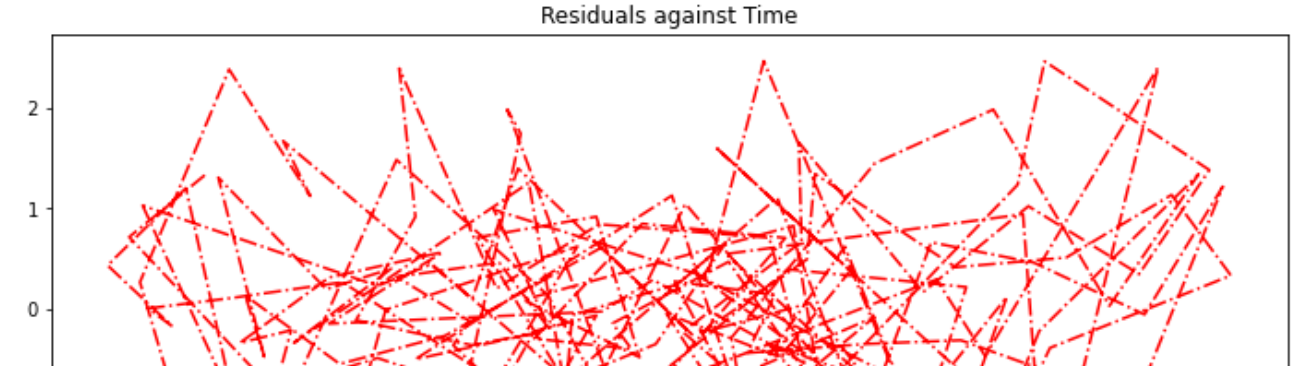


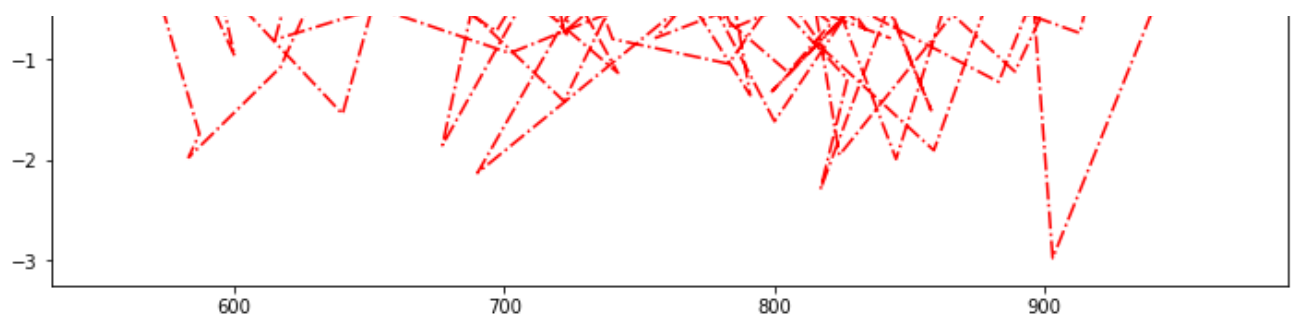
In [ ]:

```
#creating the residual component
np.random.seed(10)  # for result reproducibility
residual = np.random.normal(loc=0.0, scale=1, size=len(T_Series))

#plot the residual component
plt.plot(T_Series, residual, 'r-.')
plt.title("Residuals against Time")
```

Out[ ]:

```
Text(0.5, 1.0, 'Residuals against Time')
```



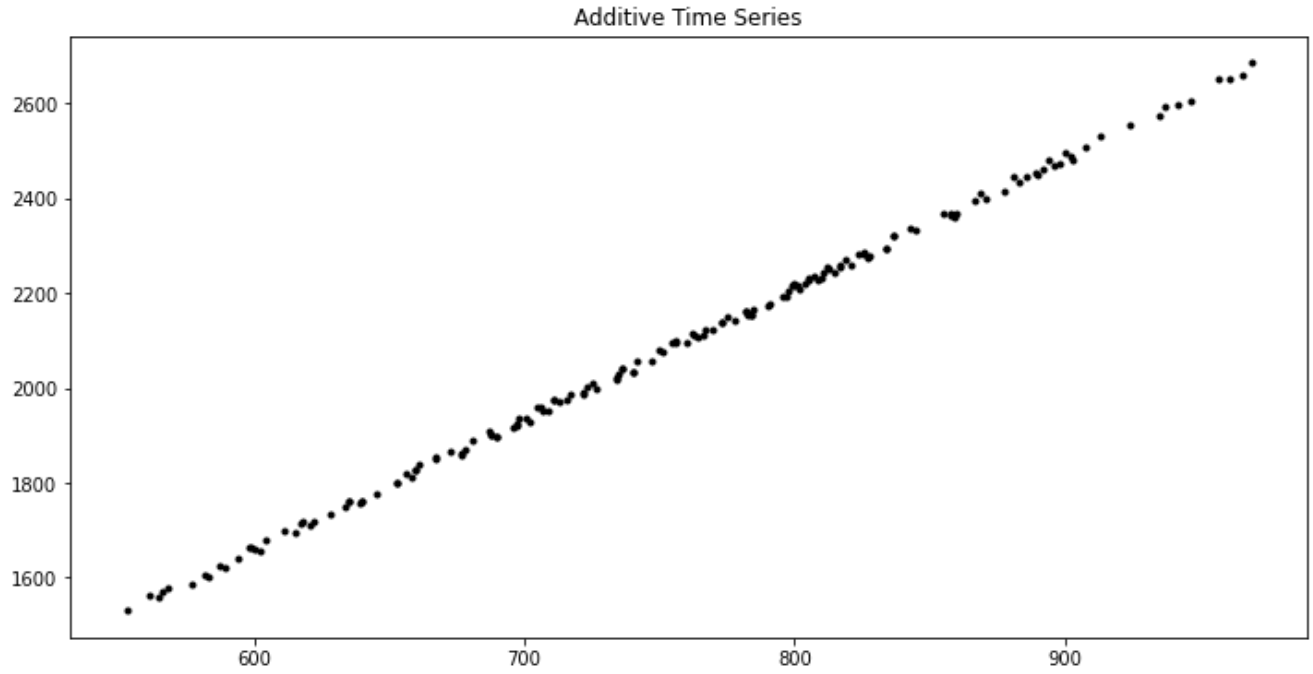


In [ ]:

```
#create an additive model of our time series
additive_Tmodel = Trend + seasonality + residual

#Plot the additive model
plt.plot(T_Series, additive_Tmodel, 'k.')
plt.title("Additive Time Series")
```

Out[ ]:

```
Text(0.5, 1.0, 'Additive Time Series')
```

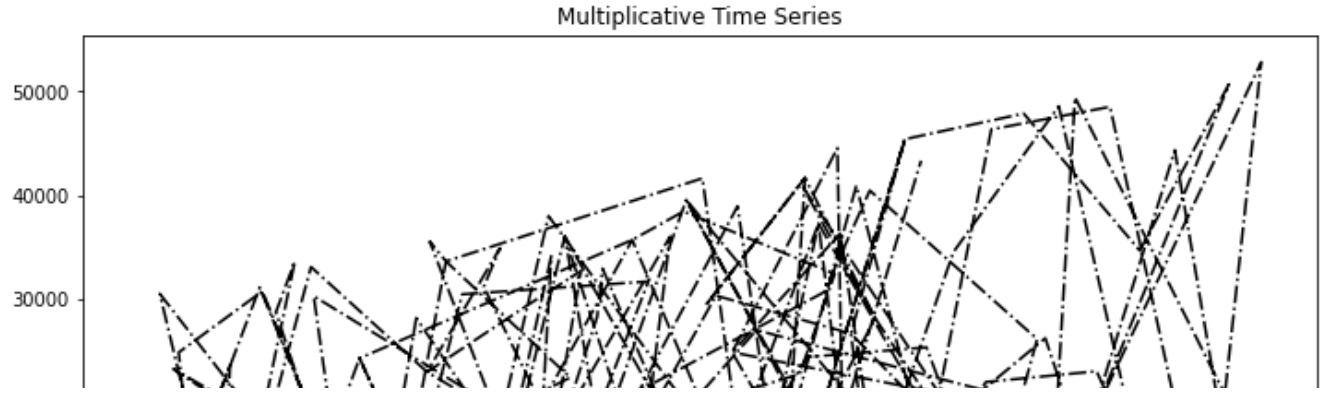


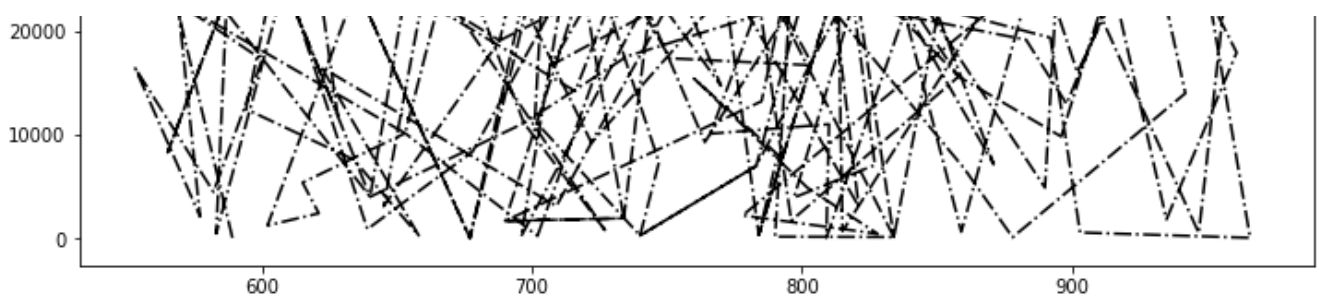
In [ ]:

```
ignored_residual = np.ones_like(residual)

multiplicative_Tmodel = Trend * seasonality * ignored_residual

plt.plot(T_Series, multiplicative_Tmodel, 'k-.')
plt.title("Multiplicative Time Series")
```

Out[ ]:

```
Text(0.5, 1.0, 'Multiplicative Time Series')
```



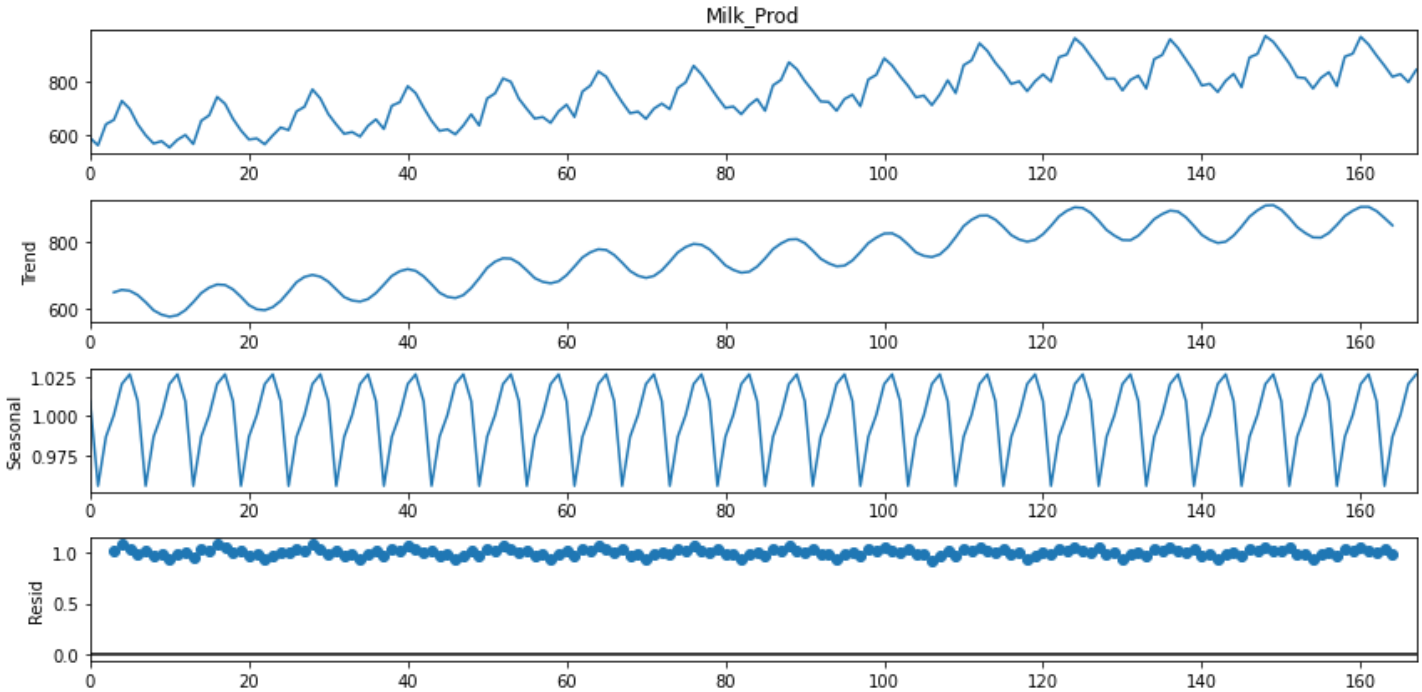


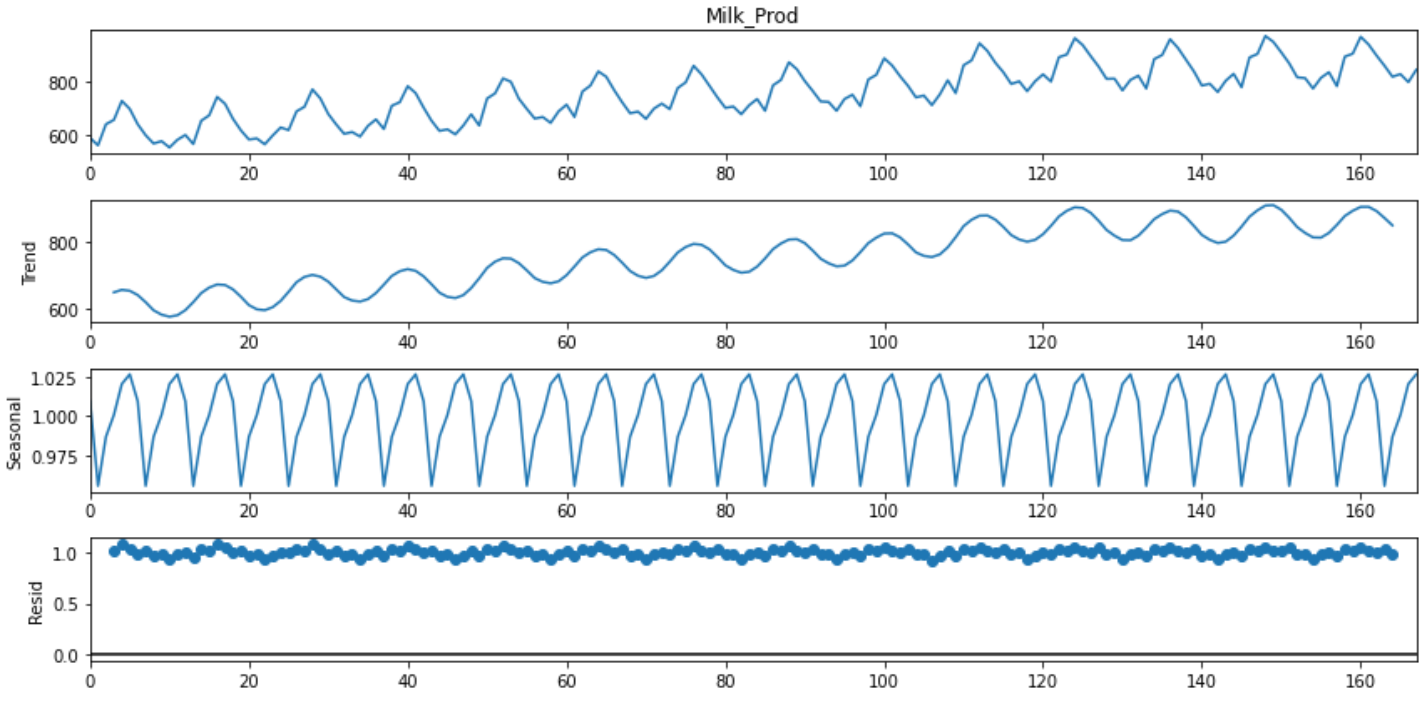
b) Using libraries

In [ ]:

```
result = seasonal_decompose(df['Milk_Prod'], model='mul',freq =6)
result.plot()
```

Out[ ]:

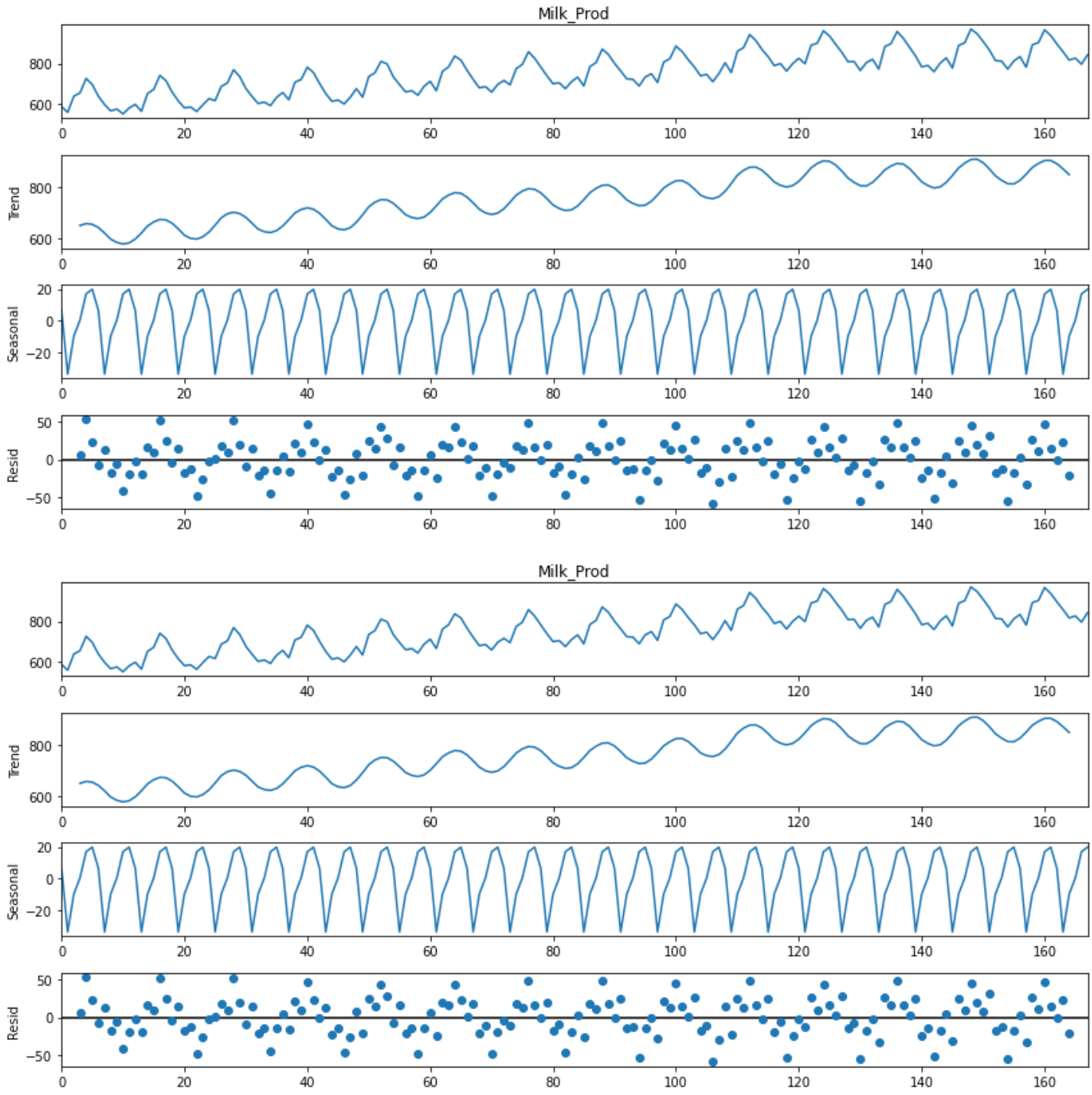




In [ ]:

```
result = seasonal_decompose(df['Milk_Prod'], model='add', freq =6)
result.plot()
```

Out[ ]:



Chapter 6: Smoothing Methods

https://colab.research.google.com/drive/1odoatMSnHKpJQVfAHqMHlJwv6vfdW3pT?usp=sharing

Chapter 7: Testing Stationarity

In [ ]:

```
milk = pd.read_csv(r'/content/drive/MyDrive/Time Series Analysis/monthly-milk-production-pounds.csv', header=0, index_col=0)
milk.head()
```

Out[ ]:

Month	Monthly milk production: pounds per cow

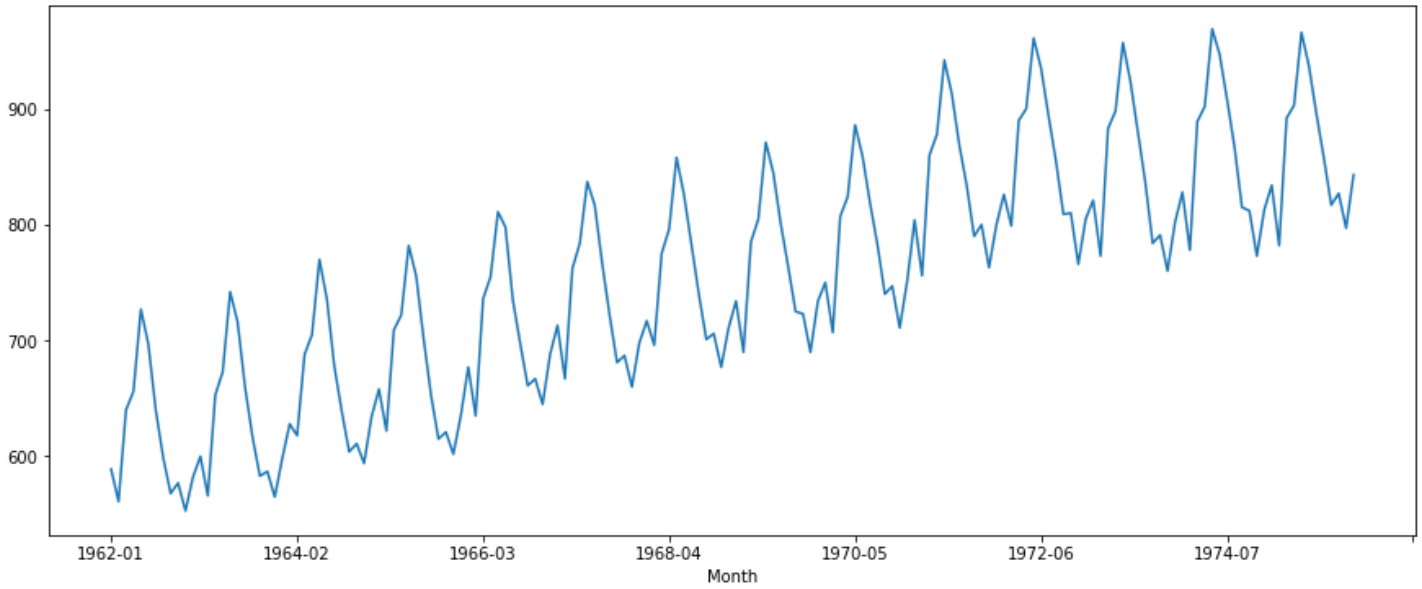
	Monthly milk production: pounds per cow
1962-01	589.0
1962-02	561.0
Month	
1962-03	640.0
1962-04	656.0
1962-05	727.0

In [ ]:

```
milk.rename(columns = {'Monthly milk production: pounds per cow':'Milk_Production'}, inp
lace = True)
milk.dropna(inplace = True)
```

In [ ]:

```
milk['Milk_Production'].plot(figsize=(15,6))
plt.show()
```



In [ ]:

```
from statsmodels.tsa.stattools import adfuller
def Augmented_Dickey_Fuller_Test_func(series , column_name):
    print (f'Results of Dickey-Fuller Test for column: {column_name}')
    dftest = adfuller(series, autolag='aic') #AIC: Method to use when automatically dete
rmining the lag length among the values 0, 1, …, maxlag
    #The t-value measures the size of the difference relative to the variation in your sa
mple data.
    #T is simply the calculated difference represented in units of standard error.
    #The greater the magnitude of T, the greater the evidence against the null hypothesis
.

    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','No Lags Used','
No of Observations Used'])
    for key,value in dftest[4].items():
       dfoutput['Critical Value (%s)'%key] = value #Critical values for the ADF test for
1%, 5%, and 10% significance levels with the constant model are -3.43, -2.87, and -2.57,
respectively.
    print (dfoutput)
    if dftest[1] <= 0.05:
        print("Conclusion:====>")
        print("Reject the null hypothesis")
        print("Data is stationary")
    else:
        print("Conclusion:====>")
        print("Fail to reject the null hypothesis")
        print("Data is non-stationary")
```

In [ ]:

```
Augmented_Dickey_Fuller_Test_func(milk['Milk_Production'],'Milk_Production')
```

```
Results of Dickey-Fuller Test for column: Milk_Production
Test Statistic                 -1.303812
p-value                         0.627427
No Lags Used                   13.000000
No of Observations Used       154.000000
Critical Value (1%)            -3.473543
Critical Value (5%)            -2.880498
Critical Value (10%)           -2.576878
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary
```

a. Calculate the following:

> **i. Test Statistic -1.303812**
>
> **ii. p-value 0.627427**
>
> **iii. No Lags Used 13.000000**
>
> **iv. Number of Observations Used 154.000000**
>
> **v. Critical Value**
>
> - **1% : -3.473543**
> - **5% : -2.880498**
> - **10% : -2.576878**

b. Conclude if the time series data contains unit roots or not.

> **The time series data contains unit roots.**

c. Also, infer if the data is stationary or not.

> **Data is not stationary.**

d. Apply differencing if the data is not stationary.

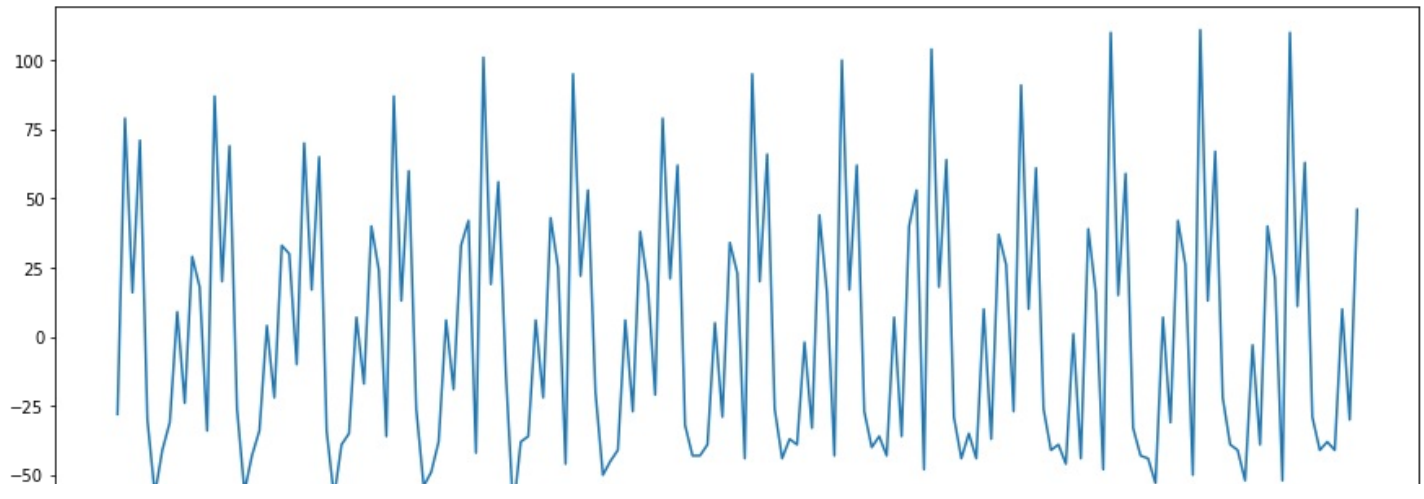
> **We apply first differencing as follows:**

In [ ]:

```
milk['Milk_Production'].diff().plot(figsize=(15,6))
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0e872dcaf0>
```



In [ ]:

```
milk_firstdiff = milk['Milk_Production'].diff()
Augmented_Dickey_Fuller_Test_func(milk_firstdiff.dropna(),'Milk Production')
```

```
Results of Dickey-Fuller Test for column: Milk Production
Test Statistic               -3.054996
p-value                       0.030068
No Lags Used                 14.000000
No of Observations Used     152.000000
Critical Value (1%)          -3.474121
Critical Value (5%)          -2.880750
Critical Value (10%)         -2.577013
dtype: float64
Conclusion:====>
Reject the null hypothesis
Data is stationary
```

a. Calculate the following:

> **i. Test Statistic -3.054996**
>
> **ii. p-value 0.030068**
>
> **iii. No Lags Used 14.000000**
>
> **iv. Number of Observations Used 152.000000**
>
> **v. Critical Value**
>
> - **1% : -3.474121**
> - **5% : -2.880750**
> - **10% : -2.577013**

b. Conclude if the time series data contains unit roots or not.

> **The time series data does not contain unit roots anymore.**

c. Also, infer if the data is stationary or not.

> **Data is now stationary.**

d. Apply differencing if the data is not stationary.

> **Now the data is stationary.**

Chapter 9: Implementation and interpretation for forecast

Model 1: ARIMA

In [ ]:

```
!pip install pmdarima
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publi
c/simple/
Requirement already satisfied: pmdarima in /usr/local/lib/python3.8/dist-packages (2.0.2)
Requirement already satisfied: setuptools!=50.0.0,>=38.6.0 in /usr/local/lib/python3.8/di
```

```
st-packages (from pmdarima) (57.4.0)
Requirement already satisfied: Cython!=0.29.18,!=0.29.31,>=0.29 in /usr/local/lib/python3
.8/dist-packages (from pmdarima) (0.29.32)
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/lib/python3.8/dist-packag
es (from pmdarima) (1.0.2)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.8/dist-packages (fr
om pmdarima) (1.7.3)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.8/dist-packages (from pm
darima) (1.24.3)
Requirement already satisfied: pandas>=0.19 in /usr/local/lib/python3.8/dist-packages (fr
om pmdarima) (1.3.5)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.8/dist-packages (f
rom pmdarima) (1.21.6)
Requirement already satisfied: statsmodels>=0.13.2 in /usr/local/lib/python3.8/dist-packa
ges (from pmdarima) (0.13.5)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.8/dist-packages (fr
om pmdarima) (1.2.0)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.8/dist-pa
ckages (from pandas>=0.19->pmdarima) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.8/dist-packages (fr
om pandas>=0.19->pmdarima) (2022.6)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-packages (from p
ython-dateutil>=2.7.3->pandas>=0.19->pmdarima) (1.15.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.8/dist-pack
ages (from scikit-learn>=0.22->pmdarima) (3.1.0)
Requirement already satisfied: patsy>=0.5.2 in /usr/local/lib/python3.8/dist-packages (fr
om statsmodels>=0.13.2->pmdarima) (0.5.3)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.8/dist-packages
(from statsmodels>=0.13.2->pmdarima) (21.3)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.8/dist-
packages (from packaging>=21.3->statsmodels>=0.13.2->pmdarima) (3.0.9)
```

In [ ]:

```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import metrics
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller

warnings.filterwarnings("ignore")
```

In [ ]:

```
df = pd.read_csv(r'/content/drive/MyDrive/Time Series Analysis/monthly-milk-production-po
unds.csv', parse_dates = True)
```

In [ ]:

```
df.head(5)
```

Out[ ]:

	Month	Monthly milk production: pounds per cow
0	1962-01	589.0
1	1962-02	561.0
2	1962-03	640.0
3	1962-04	656.0
4	1962-05	727.0

In [ ]:

```
df.dropna(inplace=True)
df.rename(columns = {'Monthly milk production: pounds per cow':'Milk_Prod'}, inplace = T
rue)
```

```
df.head(5)
```

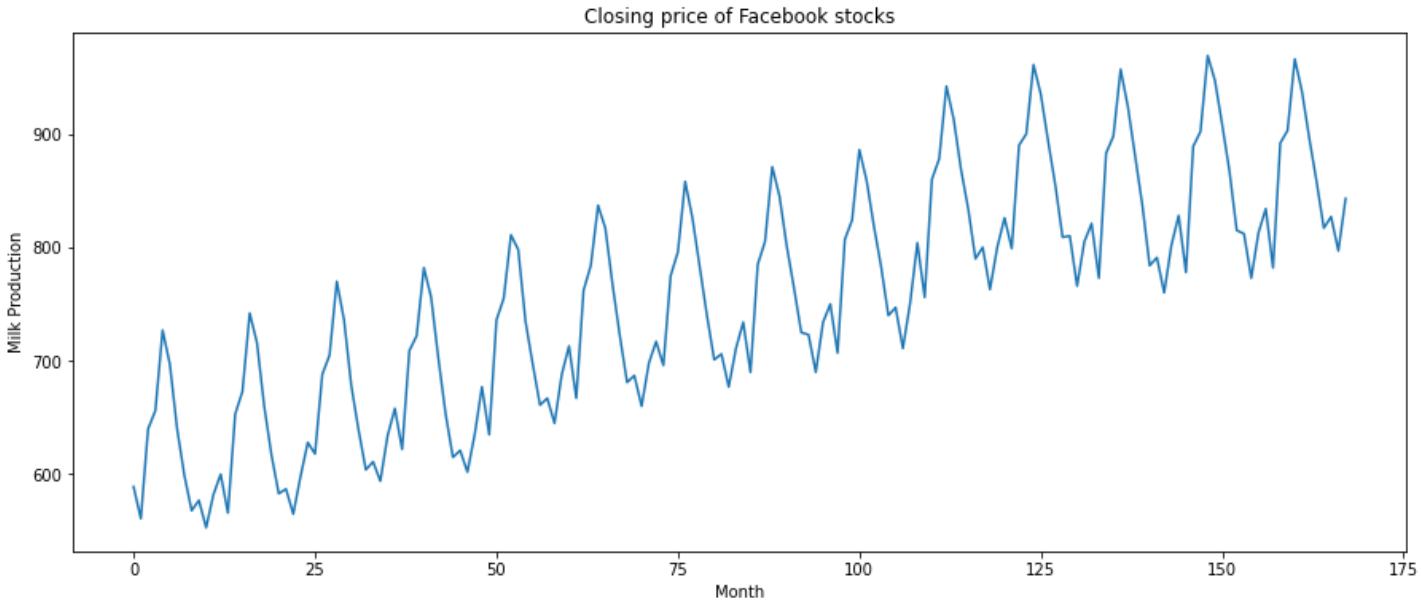
Out[ ]:

	Month	Milk_Prod
0	1962-01	589.0
1	1962-02	561.0
2	1962-03	640.0
3	1962-04	656.0
4	1962-05	727.0

a. Plot a histogram and compare the values with N (0,1).

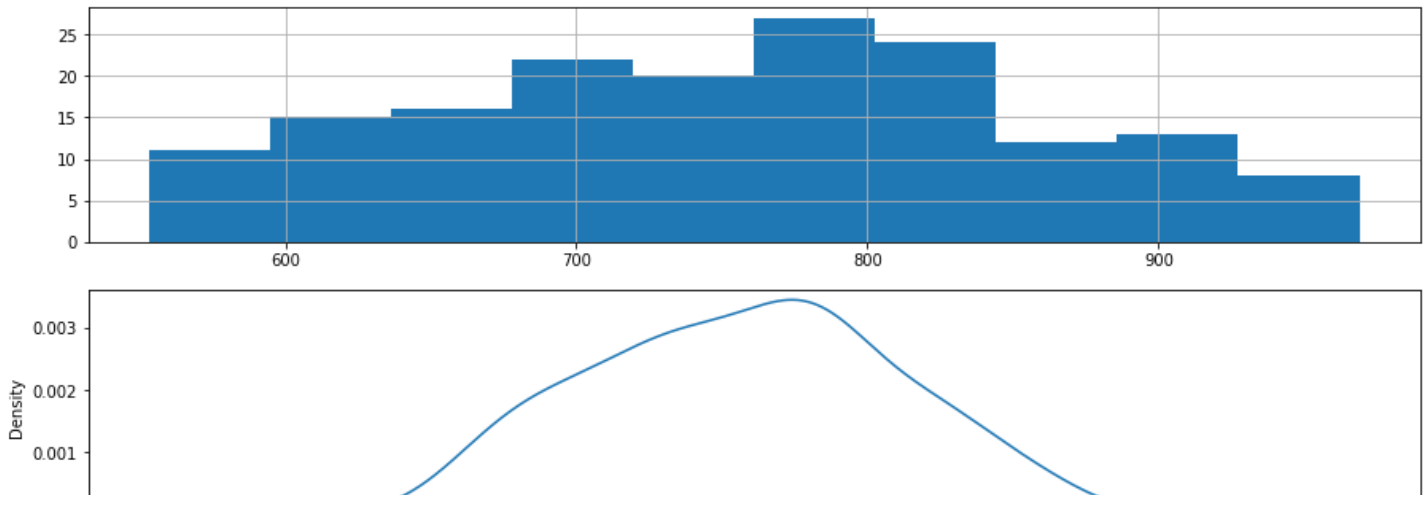
In [ ]:

```
df["Milk_Prod"].plot(figsize=(15, 6))
plt.xlabel("Month")
plt.ylabel("Milk Production")
plt.title("Closing price of Facebook stocks")
plt.show()
```



In [ ]:

```
plt.figure(1, figsize=(15,6))
plt.subplot(211)
df["Milk_Prod"].hist()
plt.subplot(212)
df["Milk_Prod"].plot(kind='kde')
plt.show()
```



b. Check for Stationarity.

In [ ]:

```
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true, y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true, y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true, y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end='\n\n')
```

In [ ]:

```
def Augmented_Dickey_Fuller_Test_func(series , column_name):
    print (f'Results of Dickey-Fuller Test for column: {column_name}')
    dftest = adfuller(series, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','No Lags Used','
Number of Observations Used'])
    for key,value in dftest[4].items():
       dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput)
    if dftest[1] <= 0.05:
        print("Conclusion:====>")
        print("Reject the null hypothesis")
        print("Data is stationary")
    else:
        print("Conclusion:====>")
        print("Fail to reject the null hypothesis")
        print("Data is non-stationary")
```

In [ ]:

```
Augmented_Dickey_Fuller_Test_func(df['Milk_Prod' ],'Milk_Prod')
```

```
Results of Dickey-Fuller Test for column: Milk_Prod
Test Statistic                  -1.303812
p-value                          0.627427
No Lags Used                    13.000000
Number of Observations Used    154.000000
Critical Value (1%)             -3.473543
Critical Value (5%)             -2.880498
Critical Value (10%)            -2.576878
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary
```

Close is non-stationary and auto-arima handles this internally.

c. State the coefficients which must be used for the appropriate model selected.

In [ ]:

```
X = df[['Milk_Prod']]
train, test = X[0:-30], X[-30:]
```

In [ ]:

```
stepwise_model = auto_arima(train,start_p=1, start_q=1,
```

```
    max_p=7, max_q=7, seasonal=True,
    d=None, trace=True,error_action='ignore',suppress_warnings=True, stepwise=True)
print(stepwise_model)
```

```
Performing stepwise search to minimize aic
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=1438.162, Time=0.25 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=1434.648, Time=0.02 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=1436.513, Time=0.09 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=1436.558, Time=0.14 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=1433.055, Time=0.02 sec

Best model:  ARIMA(0,1,0)(0,0,0)[0]
Total fit time: 0.543 seconds
 ARIMA(0,1,0)(0,0,0)[0]
```

In [ ]:

```
stepwise_model.summary()
```

Out[ ]:

SARIMAX Results

Dep. Variable:	**y**	**No. Observations:**	**138**
Model:	**SARIMAX(0, 1, 0)**	**Log Likelihood**	**-715.528**
Date:	**Fri, 02 Dec 2022**	**AIC**	**1433.055**
Time:	**14:30:47**	**BIC**	**1435.975**
Sample:	**0**	**HQIC**	**1434.242**
	- 138		
Covariance Type:	**opg**		

	coef	std err	z	P>|z|	[0.025	0.975]
sigma2	2013.8117	292.256	6.891	0.000	1441.000	2586.624

Ljung-Box (L1) (Q):	**0.14**	**Jarque-Bera (JB):**	**11.40**
Prob(Q):	**0.71**	**Prob(JB):**	**0.00**
Heteroskedasticity (H):	**1.20**	**Skew:**	**0.60**
Prob(H) (two-sided):	**0.54**	**Kurtosis:**	**2.25**

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

Auto-ARIMA says ARIMA(0,1,0) is the optimal selection for the dataset.

Forecast both results and the confidence for the next 30 months and store it in a DataFrame.

Confidence is the range of values you expect your estimate to fall between if you redo your test, within a certain level of confidence.

return_conf_int=True: Whether to get the confidence intervals of the forecasts.

d. Calculate the evaluation metrics (MSE, RMSE, MAPE and R2).

In [ ]:

```
forecast,conf_int = stepwise_model.predict(n_periods=30,return_conf_int=True)
forecast = pd.DataFrame(forecast,columns=['milk_prod_pred'])
```

In [ ]:

```
df_conf = pd.DataFrame(conf_int,columns= ['Upper_bound','Lower_bound'])
```

```
df_conf["new_index"] = range(138, 168)
df_conf = df_conf.set_index("new_index")
```

In [ ]:

```
timeseries_evaluation_metrics_func(test, forecast)
```

```
Evaluation metric results:-
MSE is : 8889.466666666667
MAE is : 81.93333333333334
RMSE is : 94.28396823780099
MAPE is : 10.034072890491244
R2 is : -1.5745211058394069
```

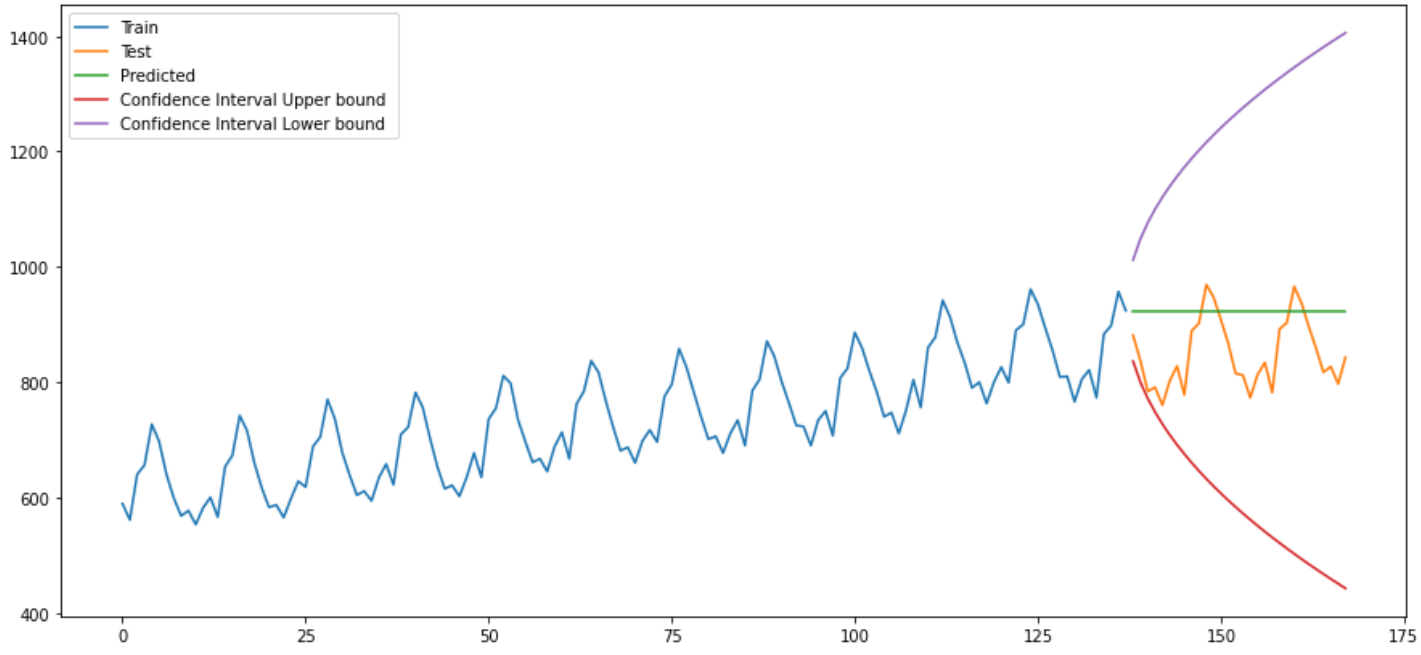
e. Forecast the future values and plot Confidence Interval Upper bound and Confidence Interval Lower bound with respect to train, test and predicted.

In [ ]:

```
forecast["new_index"] = range(138, 168)
forecast = forecast.set_index("new_index")
```

In [ ]:

```
plt.rcParams["figure.figsize"] = [15,7]
plt.plot( train, label='Train ')
plt.plot(test, label='Test ')
plt.plot(forecast, label='Predicted ')
plt.plot(df_conf['Upper_bound'], label='Confidence Interval Upper bound ')
plt.plot(df_conf['Lower_bound'], label='Confidence Interval Lower bound ')
plt.legend(loc='best')
plt.show()
```



f. Analyse the actual data with predicted based on the plots:

i. Standardize Residual

ii. Histogram plus estimated density

iii. Normal Q-Q

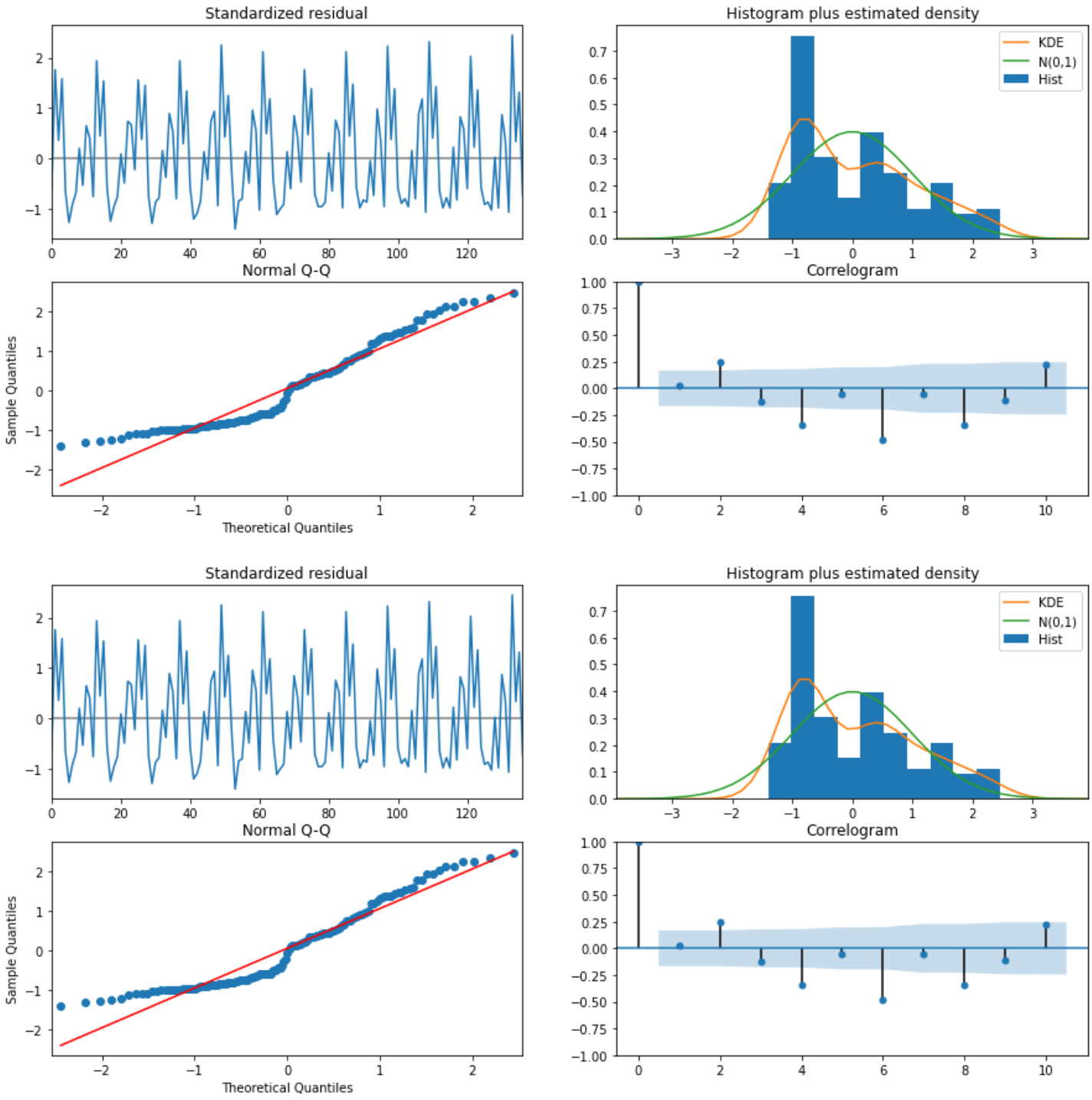
iv. Correlogram

In [ ]:

```
stepwise_model.plot_diagnostics()
```

```
stepwise_model.plot_diagnostics()
```

Out[ ]:



In [ ]:

```
forecast
```

Out[ ]:

new_index	milk_prod_pred
138	924.0
139	924.0
140	924.0
141	924.0
142	924.0
143	924.0
144	924.0
145	924.0

new_index	milk_prod_pred
145	924.0
146	924.0
147	924.0
148	924.0
149	924.0
150	924.0
151	924.0
152	924.0
153	924.0
154	924.0
155	924.0
156	924.0
157	924.0
158	924.0
159	924.0
160	924.0
161	924.0
162	924.0
163	924.0
164	924.0
165	924.0
166	924.0
167	924.0

Model 2: CNN

a. Import all the libraries from Keras for neural network architectures.

In [ ]:

```
from numpy import array
import pandas as pd
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
```

b. Define a function that extracts features (lagged values)

In [ ]:

```
def split_sequence(sequence,steps):
  X,y=[],[]
  for start in range(len(sequence)):
    end_index = start+steps
    if end_index>len(sequence)-1:
      break
    sequence_x,sequence_y = sequence[start:end_index],sequence[end_index]
    X.append(sequence_x)
    y.append(sequence_y)
  return(array(X),array(y))
```

In [ ]:

```
df = pd.read_csv('/content/drive/MyDrive/Time Series Analysis/monthly-milk-production-pounds.csv')
df = df.dropna()
df.head()
```

Out[ ]:

	Month	Monthly milk production: pounds per cow
0	1962-01	589.0
1	1962-02	561.0
2	1962-03	640.0
3	1962-04	656.0
4	1962-05	727.0

c. Initializing Sequence, steps, and reshaping the output to input it to our CNN model.

In [ ]:

```
raw_sequence = df["Monthly milk production: pounds per cow"]

steps=3
X,y = split_sequence(raw_sequence,steps)

print("The input raw sequence on which we will test our CNN model: ",raw_sequence)
print("The features (X), lagged values of time series, for our CNN model: ",X)
print("The respectice observed values (y)of the sequence for training our CNN model: ",y)
```

```
The input raw sequence on which we will test our CNN model:  0      589.0
1      561.0
2      640.0
3      656.0
4      727.0
       ...
163    858.0
164    817.0
165    827.0
166    797.0
167    843.0
Name: Monthly milk production: pounds per cow, Length: 168, dtype: float64
The features (X), lagged values of time series, for our CNN model:  [[589. 561. 640.]
 [561. 640. 656.]
 [640. 656. 727.]
 [656. 727. 697.]
 [727. 697. 640.]
 [697. 640. 599.]
 [640. 599. 568.]
 [599. 568. 577.]
 [568. 577. 553.]
 [577. 553. 582.]
 [553. 582. 600.]
 [582. 600. 566.]
 [600. 566. 653.]
 [566. 653. 673.]
 [653. 673. 742.]
 [673. 742. 716.]
 [742. 716. 660.]
 [716. 660. 617.]
 [660. 617. 583.]
 [617. 583. 587.]
 [583. 587. 565.]
 [587. 565. 598.]
 [565. 598. 628.]
 [598. 628. 618.]
 [628. 618. 688.]
 [618. 688. 705.]
 [688. 705. 770.]
 [705. 770. 736.]
```

```
[770. 736. 678.]
[736. 678. 639.]
[678. 639. 604.]
[639. 604. 611.]
[604. 611. 594.]
[611. 594. 634.]
[594. 634. 658.]
[634. 658. 622.]
[658. 622. 709.]
[622. 709. 722.]
[709. 722. 782.]
[722. 782. 756.]
[782. 756. 702.]
[756. 702. 653.]
[702. 653. 615.]
[653. 615. 621.]
[615. 621. 602.]
[621. 602. 635.]
[602. 635. 677.]
[635. 677. 635.]
[677. 635. 736.]
[635. 736. 755.]
[736. 755. 811.]
[755. 811. 798.]
[811. 798. 735.]
[798. 735. 697.]
[735. 697. 661.]
[697. 661. 667.]
[661. 667. 645.]
[667. 645. 688.]
[645. 688. 713.]
[688. 713. 667.]
[713. 667. 762.]
[667. 762. 784.]
[762. 784. 837.]
[784. 837. 817.]
[837. 817. 767.]
[817. 767. 722.]
[767. 722. 681.]
[722. 681. 687.]
[681. 687. 660.]
[687. 660. 698.]
[660. 698. 717.]
[698. 717. 696.]
[717. 696. 775.]
[696. 775. 796.]
[775. 796. 858.]
[796. 858. 826.]
[858. 826. 783.]
[826. 783. 740.]
[783. 740. 701.]
[740. 701. 706.]
[701. 706. 677.]
[706. 677. 711.]
[677. 711. 734.]
[711. 734. 690.]
[734. 690. 785.]
[690. 785. 805.]
[785. 805. 871.]
[805. 871. 845.]
[871. 845. 801.]
[845. 801. 764.]
[801. 764. 725.]
[764. 725. 723.]
[725. 723. 690.]
[723. 690. 734.]
[690. 734. 750.]
[734. 750. 707.]
[750. 707. 807.]
[707. 807. 824.]
[807. 824. 886.]
[824. 886. 859.]
```

[886. 859. 819.]
[859. 819. 783.]
[819. 783. 740.]
[783. 740. 747.]
[740. 747. 711.]
[747. 711. 751.]
[711. 751. 804.]
[751. 804. 756.]
[804. 756. 860.]
[756. 860. 878.]
[860. 878. 942.]
[878. 942. 913.]
[942. 913. 869.]
[913. 869. 834.]
[869. 834. 790.]
[834. 790. 800.]
[790. 800. 763.]
[800. 763. 800.]
[763. 800. 826.]
[800. 826. 799.]
[826. 799. 890.]
[799. 890. 900.]
[890. 900. 961.]
[900. 961. 935.]
[961. 935. 894.]
[935. 894. 855.]
[894. 855. 809.]
[855. 809. 810.]
[809. 810. 766.]
[810. 766. 805.]
[766. 805. 821.]
[805. 821. 773.]
[821. 773. 883.]
[773. 883. 898.]
[883. 898. 957.]
[898. 957. 924.]
[957. 924. 881.]
[924. 881. 837.]
[881. 837. 784.]
[837. 784. 791.]
[784. 791. 760.]
[791. 760. 802.]
[760. 802. 828.]
[802. 828. 778.]
[828. 778. 889.]
[778. 889. 902.]
[889. 902. 969.]
[902. 969. 947.]
[969. 947. 908.]
[947. 908. 867.]
[908. 867. 815.]
[867. 815. 812.]
[815. 812. 773.]
[812. 773. 813.]
[773. 813. 834.]
[813. 834. 782.]
[834. 782. 892.]
[782. 892. 903.]
[892. 903. 966.]
[903. 966. 937.]
[966. 937. 896.]
[937. 896. 858.]
[896. 858. 817.]
[858. 817. 827.]
[817. 827. 797.]]
The respectice observed values (y)of the sequence for training our CNN model: [656. 727. 697. 640. 599. 568. 577. 553. 582. 600. 566. 653. 673. 742.
716. 660. 617. 583. 587. 565. 598. 628. 618. 688. 705. 770. 736. 678.
639. 604. 611. 594. 634. 658. 622. 709. 722. 782. 756. 702. 653. 615.
621. 602. 635. 677. 635. 736. 755. 811. 798. 735. 697. 661. 667. 645.
688. 713. 667. 762. 784. 837. 817. 767. 722. 681. 687. 660. 698. 717.
696. 775. 796. 858. 826. 783. 740. 701. 706. 677. 711. 734. 690. 785.

```
 805. 871. 845. 801. 764. 725. 723. 690. 734. 750. 707. 807. 824. 886.
 859. 819. 783. 740. 747. 711. 751. 804. 756. 860. 878. 942. 913. 869.
 834. 790. 800. 763. 800. 826. 799. 890. 900. 961. 935. 894. 855. 809.
 810. 766. 805. 821. 773. 883. 898. 957. 924. 881. 837. 784. 791. 760.
 802. 828. 778. 889. 902. 969. 947. 908. 867. 815. 812. 773. 813. 834.
 782. 892. 903. 966. 937. 896. 858. 817. 827. 797. 843.]
```

d. Reshaping the X matrices

In [ ]:

```
features = 1
X = X.reshape((X.shape[0],X.shape[1],features))

print("After reshaping, the shape of input X",X.shape)
print("Final form of input feature matrix X",X)
print("Feature matrix X is ready for input to CNN model. We have used feature engineering
to convert a sequence to matrix with image matrix shape to find patterns in sequence.")
```

```
After reshaping, the shape of input X (165, 3, 1)
Final form of input feature matrix X [[[589.]
  [561.]
  [640.]]

 [[561.]
  [640.]
  [656.]]

 [[640.]
  [656.]
  [727.]]

 [[656.]
  [727.]
  [697.]]

 [[727.]
  [697.]
  [640.]]

 [[697.]
  [640.]
  [599.]]

 [[640.]
  [599.]
  [568.]]

 [[599.]
  [568.]
  [577.]]

 [[568.]
  [577.]
  [553.]]

 [[577.]
  [553.]
  [582.]]

 [[553.]
  [582.]
  [600.]]

 [[582.]
  [600.]
  [566.]]

 [[600.]
  [566.]
  [653.]]
```

```
[[566.]
 [653.]
 [673.]]

[[653.]
 [673.]
 [742.]]

[[673.]
 [742.]
 [716.]]

[[742.]
 [716.]
 [660.]]

[[716.]
 [660.]
 [617.]]

[[660.]
 [617.]
 [583.]]

[[617.]
 [583.]
 [587.]]

[[583.]
 [587.]
 [565.]]

[[587.]
 [565.]
 [598.]]

[[565.]
 [598.]
 [628.]]

[[598.]
 [628.]
 [618.]]

[[628.]
 [618.]
 [688.]]

[[618.]
 [688.]
 [705.]]

[[688.]
 [705.]
 [770.]]

[[705.]
 [770.]
 [736.]]

[[770.]
 [736.]
 [678.]]

[[736.]
 [678.]
 [639.]]

[[678.]
 [639.]
 [604.]]
```

```
[[639.]
 [604.]
 [611.]]

[[604.]
 [611.]
 [594.]]

[[611.]
 [594.]
 [634.]]

[[594.]
 [634.]
 [658.]]

[[634.]
 [658.]
 [622.]]

[[658.]
 [622.]
 [709.]]

[[622.]
 [709.]
 [722.]]

[[709.]
 [722.]
 [782.]]

[[722.]
 [782.]
 [756.]]

[[782.]
 [756.]
 [702.]]

[[756.]
 [702.]
 [653.]]

[[702.]
 [653.]
 [615.]]

[[653.]
 [615.]
 [621.]]

[[615.]
 [621.]
 [602.]]

[[621.]
 [602.]
 [635.]]

[[602.]
 [635.]
 [677.]]

[[635.]
 [677.]
 [635.]]

[[677.]
 [635.]
 [736.]]
```

```
[[635.]
 [736.]
 [755.]]

[[736.]
 [755.]
 [811.]]

[[755.]
 [811.]
 [798.]]

[[811.]
 [798.]
 [735.]]

[[798.]
 [735.]
 [697.]]

[[735.]
 [697.]
 [661.]]

[[697.]
 [661.]
 [667.]]

[[661.]
 [667.]
 [645.]]

[[667.]
 [645.]
 [688.]]

[[645.]
 [688.]
 [713.]]

[[688.]
 [713.]
 [667.]]

[[713.]
 [667.]
 [762.]]

[[667.]
 [762.]
 [784.]]

[[762.]
 [784.]
 [837.]]

[[784.]
 [837.]
 [817.]]

[[837.]
 [817.]
 [767.]]

[[817.]
 [767.]
 [722.]]

[[767.]
 [722.]
 [681.]]
```

```
[[722.]
 [681.]
 [687.]]

[[681.]
 [687.]
 [660.]]

[[687.]
 [660.]
 [698.]]

[[660.]
 [698.]
 [717.]]

[[698.]
 [717.]
 [696.]]

[[717.]
 [696.]
 [775.]]

[[696.]
 [775.]
 [796.]]

[[775.]
 [796.]
 [858.]]

[[796.]
 [858.]
 [826.]]

[[858.]
 [826.]
 [783.]]

[[826.]
 [783.]
 [740.]]

[[783.]
 [740.]
 [701.]]

[[740.]
 [701.]
 [706.]]

[[701.]
 [706.]
 [677.]]

[[706.]
 [677.]
 [711.]]

[[677.]
 [711.]
 [734.]]

[[711.]
 [734.]
 [690.]]

[[734.]
 [690.]
 [785.]]
```

```
[[690.]
 [785.]
 [805.]]

[[785.]
 [805.]
 [871.]]

[[805.]
 [871.]
 [845.]]

[[871.]
 [845.]
 [801.]]

[[845.]
 [801.]
 [764.]]

[[801.]
 [764.]
 [725.]]

[[764.]
 [725.]
 [723.]]

[[725.]
 [723.]
 [690.]]

[[723.]
 [690.]
 [734.]]

[[690.]
 [734.]
 [750.]]

[[734.]
 [750.]
 [707.]]

[[750.]
 [707.]
 [807.]]

[[707.]
 [807.]
 [824.]]

[[807.]
 [824.]
 [886.]]

[[824.]
 [886.]
 [859.]]

[[886.]
 [859.]
 [819.]]

[[859.]
 [819.]
 [783.]]

[[819.]
 [783.]
 [740.]]
```

```
[[783.]
 [740.]
 [747.]]

[[740.]
 [747.]
 [711.]]

[[747.]
 [711.]
 [751.]]

[[711.]
 [751.]
 [804.]]

[[751.]
 [804.]
 [756.]]

[[804.]
 [756.]
 [860.]]

[[756.]
 [860.]
 [878.]]

[[860.]
 [878.]
 [942.]]

[[878.]
 [942.]
 [913.]]

[[942.]
 [913.]
 [869.]]

[[913.]
 [869.]
 [834.]]

[[869.]
 [834.]
 [790.]]

[[834.]
 [790.]
 [800.]]

[[790.]
 [800.]
 [763.]]

[[800.]
 [763.]
 [800.]]

[[763.]
 [800.]
 [826.]]

[[800.]
 [826.]
 [799.]]

[[826.]
 [799.]
 [890.]]
```

```
[[799.]
 [890.]
 [900.]]

[[890.]
 [900.]
 [961.]]

[[900.]
 [961.]
 [935.]]

[[961.]
 [935.]
 [894.]]

[[935.]
 [894.]
 [855.]]

[[894.]
 [855.]
 [809.]]

[[855.]
 [809.]
 [810.]]

[[809.]
 [810.]
 [766.]]

[[810.]
 [766.]
 [805.]]

[[766.]
 [805.]
 [821.]]

[[805.]
 [821.]
 [773.]]

[[821.]
 [773.]
 [883.]]

[[773.]
 [883.]
 [898.]]

[[883.]
 [898.]
 [957.]]

[[898.]
 [957.]
 [924.]]

[[957.]
 [924.]
 [881.]]

[[924.]
 [881.]
 [837.]]

[[881.]
 [837.]
 [784.]]
```

```
[[837.]
 [784.]
 [791.]]

[[784.]
 [791.]
 [760.]]

[[791.]
 [760.]
 [802.]]

[[760.]
 [802.]
 [828.]]

[[802.]
 [828.]
 [778.]]

[[828.]
 [778.]
 [889.]]

[[778.]
 [889.]
 [902.]]

[[889.]
 [902.]
 [969.]]

[[902.]
 [969.]
 [947.]]

[[969.]
 [947.]
 [908.]]

[[947.]
 [908.]
 [867.]]

[[908.]
 [867.]
 [815.]]

[[867.]
 [815.]
 [812.]]

[[815.]
 [812.]
 [773.]]

[[812.]
 [773.]
 [813.]]

[[773.]
 [813.]
 [834.]]

[[813.]
 [834.]
 [782.]]

[[834.]
 [782.]
 [892.]]
```

```
 [[782.]
  [892.]
  [903.]]

 [[892.]
  [903.]
  [966.]]

 [[903.]
  [966.]
  [937.]]

 [[966.]
  [937.]
  [896.]]

 [[937.]
  [896.]
  [858.]]

 [[896.]
  [858.]
  [817.]]

 [[858.]
  [817.]
  [827.]]

 [[817.]
  [827.]
  [797.]]]
Feature matrix X is ready for input to CNN model. We have used feature engineering to con
vert a sequence to matrix with image matrix shape to find patterns in sequence.
```

e. Define the CNN model.

In [ ]:

```
model = Sequential()
model.add(Conv1D(filters=64,kernel_size=2,activation="relu",input_shape=(steps,features)
))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(100,activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam',loss='mse')
```

f. Implement CNN Model Fitting.

In [ ]:

```
model.fit(X,y,epochs=1000,verbose=0)
```

Out[ ]:

```
<keras.callbacks.History at 0x7f0e2e6221c0>
```

g. Predict the future values.

In [ ]:

```
y_pred=[]
for i in range(138,168):
  x_input = array(raw_sequence[i-3:i])
  x_input = x_input.reshape((1,steps,features))
  mp = model.predict(x_input,verbose=0)[0]
  yp = y_pred.append(mp[0])

print(f"The predicted values for the sequence {raw_sequence} for data points between 138
```

```
and 167")
print("are",y_pred)
```

```
The predicted values for the sequence 0      589.0
1      561.0
2      640.0
3      656.0
4      727.0
       ...
163    858.0
164    817.0
165    827.0
166    797.0
167    843.0
Name: Monthly milk production: pounds per cow, Length: 168, dtype: float64 for data point
s between 138 and 167
are [936.11865, 851.8923, 806.4495, 764.918, 759.4829, 751.2463, 782.10986, 804.2042, 798
.2645, 905.5056, 933.19794, 949.1377, 952.9298, 879.4255, 833.7303, 794.5124, 779.2638, 7
65.1379, 791.50665, 809.9722, 801.0512, 907.78906, 934.03955, 944.1784, 946.4997, 866.209
4, 821.7299, 787.6525, 789.60724, 786.5853]
```

h. Plot the graph the predicted value.

In [ ]:

```
import pandas as pd
df = pd.read_csv('/content/drive/MyDrive/Time Series Analysis/monthly-milk-production-pou
nds.csv')
df = df.dropna()
raw_sequence = df["Monthly milk production: pounds per cow"][-30:]
pred= {"Actual Values": raw_sequence, "Predicted values":[936.11865, 851.8923, 806.4495,
764.918, 759.4829, 751.2463, 782.10986, 804.2042, 798.2645, 905.5056, 933.19794, 949.1377
, 952.9298, 879.4255, 833.7303, 794.5124, 779.2638, 765.1379, 791.50665, 809.9722, 801.05
12, 907.78906, 934.03955, 944.1784, 946.4997, 866.2094, 821.7299, 787.6525, 789.60724, 78
6.5853]}
df = pd.DataFrame(pred)
df
```

Out[ ]:

	Actual Values	Predicted values
138	**881.0**	**936.11865**
139	**837.0**	**851.89230**
140	**784.0**	**806.44950**
141	**791.0**	**764.91800**
142	**760.0**	**759.48290**
143	**802.0**	**751.24630**
144	**828.0**	**782.10986**
145	**778.0**	**804.20420**
146	**889.0**	**798.26450**
147	**902.0**	**905.50560**
148	**969.0**	**933.19794**
149	**947.0**	**949.13770**
150	**908.0**	**952.92980**
151	**867.0**	**879.42550**
152	**815.0**	**833.73030**
153	**812.0**	**794.51240**
154	**773.0**	**779.26380**
155	**813.0**	**765.13790**
156	**834.0**	**791.50665**

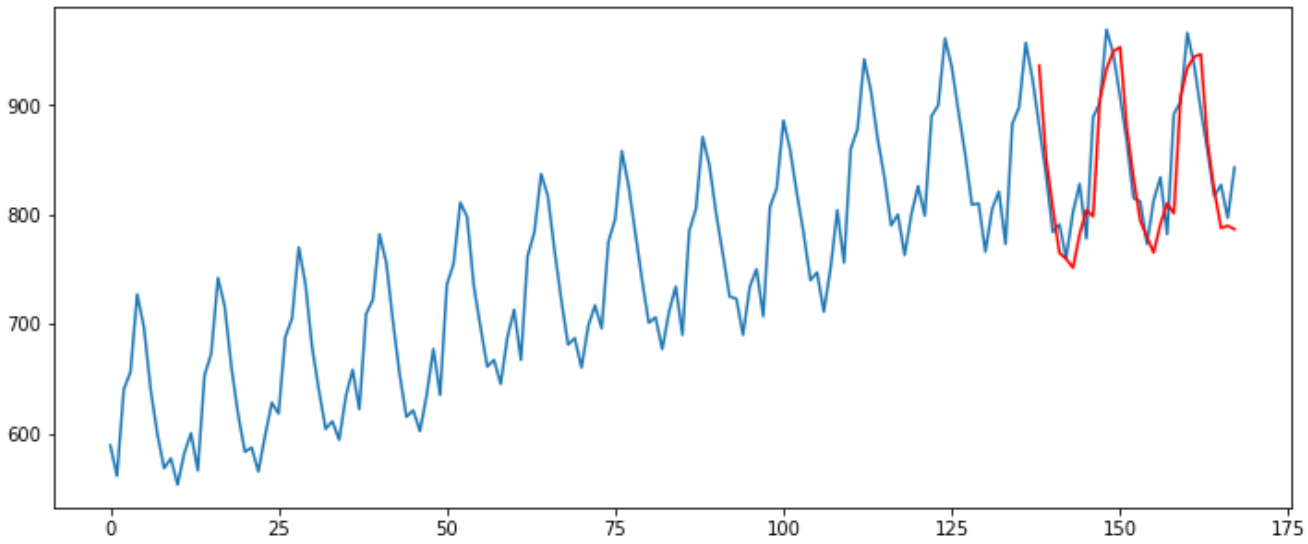
	Actual Values	Predicted values
157	782.0	809.97220
158	892.0	801.05120
159	903.0	907.78906
160	966.0	934.03955
161	937.0	944.17840
162	896.0	946.49970
163	858.0	866.20940
164	817.0	821.72990
165	827.0	787.65250
166	797.0	789.60724
167	843.0	786.58530

In [ ]:

```
df["Monthly milk production: pounds per cow"].plot(figsize=(12,5))
plt.plot([i for i in range(138,168)], y_pred, color='r')
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x7f0e2e66f2e0>]
```



In [ ]:

```
mprod = df['Monthly milk production: pounds per cow']
mprod = list(mprod[138:168])
timeseries_evaluation_metrics_func(y_pred, mprod)
```

```
Evaluation metric results:-
MSE is : 1462.8468736467262
MAE is : 29.790791829427082
RMSE is : 38.24718125099844
MAPE is : 3.602589316922293
R2 is : 0.679653367397139
```

Using CNN:

Evaluation metric results:-

> **MSE is : 8889.466666666667**
>
> **MAE is : 81.93333333333334**
>
> **RMSE is : 94.28396823780099**

MAPE is : 10.034072890491244

R2 is : -1.5745211058394069

Using CNN:

Evaluation metric results:-

MSE is : 1462.8468736467262

MAE is : 29.790791829427082

RMSE is : 38.24718125099844

MAPE is : 3.602589316922293

R2 is : 0.679653367397139

Model 3: GARCH

In [ ]:

```
!pip install arch
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publi
c/simple/
Requirement already satisfied: arch in /usr/local/lib/python3.8/dist-packages (5.3.1)
Requirement already satisfied: scipy>=1.3 in /usr/local/lib/python3.8/dist-packages (from
arch) (1.7.3)
Requirement already satisfied: property-cached>=1.6.4 in /usr/local/lib/python3.8/dist-pa
ckages (from arch) (1.6.4)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.8/dist-packages (fro
m arch) (1.21.6)
Requirement already satisfied: pandas>=1.0 in /usr/local/lib/python3.8/dist-packages (fro
m arch) (1.3.5)
Requirement already satisfied: statsmodels>=0.11 in /usr/local/lib/python3.8/dist-package
s (from arch) (0.13.5)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.8/dist-packages (fr
om pandas>=1.0->arch) (2022.6)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.8/dist-pa
ckages (from pandas>=1.0->arch) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-packages (from p
ython-dateutil>=2.7.3->pandas>=1.0->arch) (1.15.0)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.8/dist-packages
(from statsmodels>=0.11->arch) (21.3)
Requirement already satisfied: patsy>=0.5.2 in /usr/local/lib/python3.8/dist-packages (fr
om statsmodels>=0.11->arch) (0.5.3)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.8/dist-
packages (from packaging>=21.3->statsmodels>=0.11->arch) (3.0.9)
```

In [ ]:

```
from random import gauss
from arch import arch_model
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
```

In [ ]:

```
mp = pd.read_csv('/content/drive/MyDrive/Time Series Analysis/monthly-milk-production-pou
nds.csv',parse_dates= True)
mp.head()
```

Out[ ]:

	Month	Monthly milk production: pounds per cow

	Month	Monthly milk production: pounds per cow
0	1962-01	589.0
1	1962-02	561.0
2	1962-03	640.0
3	1962-04	656.0
4	1962-05	727.0

In [ ]:

```
mp.dropna(inplace=True)
mp.rename(columns={"Monthly milk production: pounds per cow": "Monthly_Prod"},inplace=True)
mp.head()
```

Out[ ]:

	Month	Monthly_Prod
0	1962-01	589.0
1	1962-02	561.0
2	1962-03	640.0
3	1962-04	656.0
4	1962-05	727.0

In [ ]:

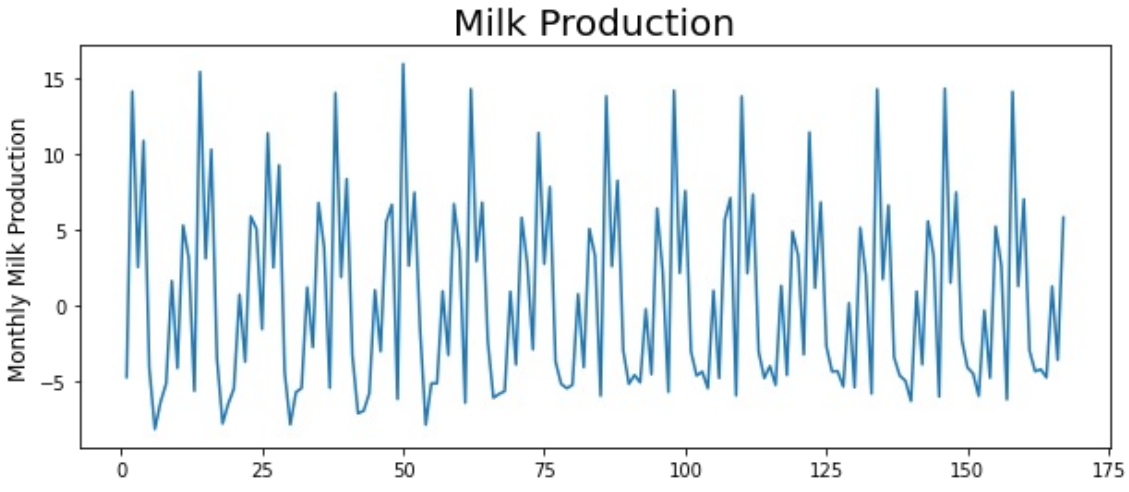
```
returns = 100*mp.Monthly_Prod.pct_change().dropna()
```

In [ ]:

```
plt.figure(figsize=(10,4))
plt.plot(returns)
plt.ylabel('Monthly Milk Production',fontsize=12)
plt.title('Milk Production',fontsize=20)
```

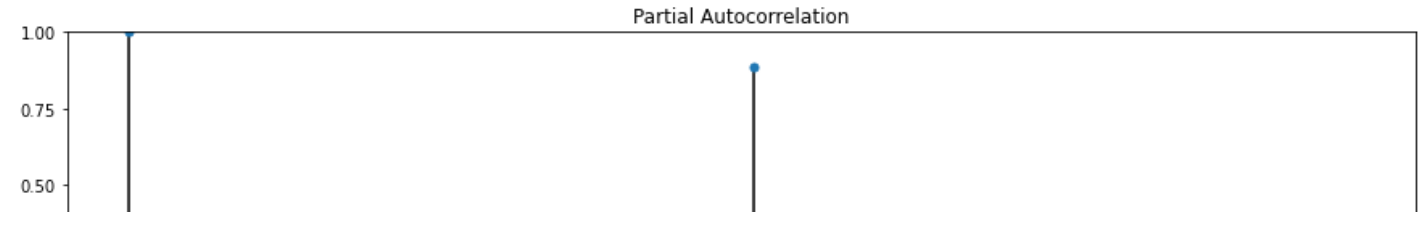
Out[ ]:

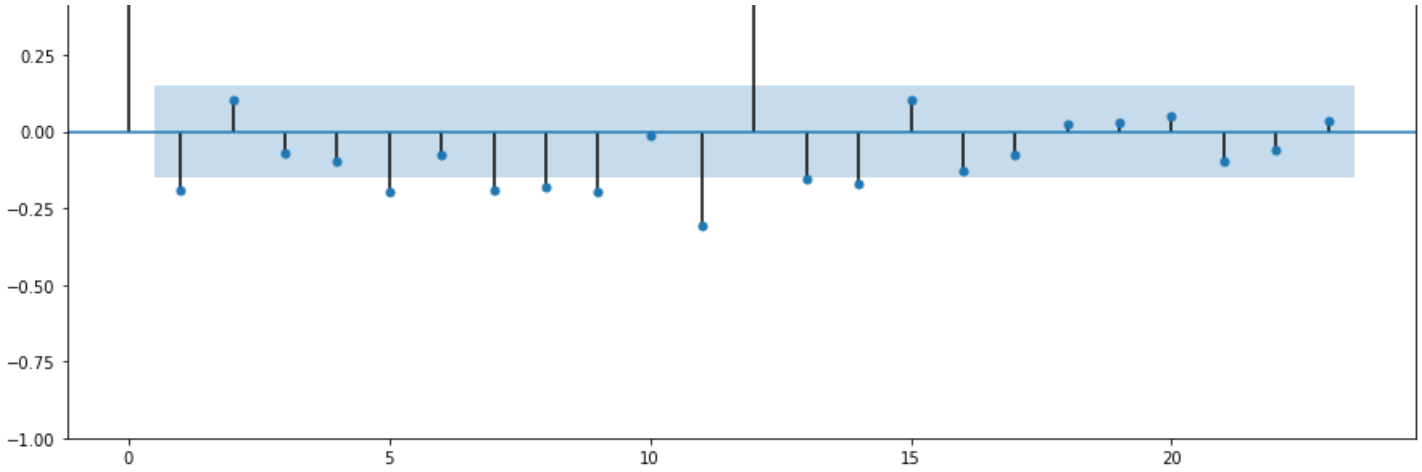
```
Text(0.5, 1.0, 'Milk Production')
```



In [ ]:

```
plot_pacf(returns**2)
plt.show()
```





Our model does not have any volatilty hence, the above plot shows no PACF value above the threshold. Hence, we can say that GARCH is not a suitable model for out dataset.

In [ ]:

```
model = arch_model(returns,p=1,q=1)
model_fit = model.fit()
model_fit.summary()
```

```
Iteration:      1,   Func. Count:      6,   Neg. LLF: 1202.9208846663814
Iteration:      2,   Func. Count:     14,   Neg. LLF: 540.9095886668101
Iteration:      3,   Func. Count:     19,   Neg. LLF: 540.8783201528573
Iteration:      4,   Func. Count:     24,   Neg. LLF: 540.8732188976921
Iteration:      5,   Func. Count:     29,   Neg. LLF: 540.8308154548415
Iteration:      6,   Func. Count:     34,   Neg. LLF: 542.2548509531562
Iteration:      7,   Func. Count:     40,   Neg. LLF: 541.2846486735282
Iteration:      8,   Func. Count:     46,   Neg. LLF: 541.0363190706321
Iteration:      9,   Func. Count:     52,   Neg. LLF: 541.2907611979283
Iteration:     10,   Func. Count:     58,   Neg. LLF: 4985.12373328337
Iteration:     11,   Func. Count:     66,   Neg. LLF: 540.7738193714922
Iteration:     12,   Func. Count:     72,   Neg. LLF: 540.5895752325059
Iteration:     13,   Func. Count:     77,   Neg. LLF: 540.8246805998508
Iteration:     14,   Func. Count:     83,   Neg. LLF: 540.6356009118631
Iteration:     15,   Func. Count:     89,   Neg. LLF: 540.5226272722762
Iteration:     16,   Func. Count:     94,   Neg. LLF: 540.5214248440084
Iteration:     17,   Func. Count:     99,   Neg. LLF: 540.5215031994494
Iteration:     18,   Func. Count:    105,   Neg. LLF: 540.5212430209192
Iteration:     19,   Func. Count:    110,   Neg. LLF: 540.5212410974377
Iteration:     20,   Func. Count:    114,   Neg. LLF: 540.5212410977607
Optimization terminated successfully    (Exit mode 0)
            Current function value: 540.5212410974377
            Iterations: 21
            Function evaluations: 114
            Gradient evaluations: 20
```

Out[ ]:

Constant Mean - GARCH Model Results

Dep. Variable:	**Monthly_Prod**	**R-squared:**	**0.000**
Mean Model:	**Constant Mean**	**Adj. R-squared:**	**0.000**
Vol Model:	**GARCH**	**Log-Likelihood:**	**-540.521**
Distribution:	**Normal**	**AIC:**	**1089.04**
Method:	**Maximum Likelihood**	**BIC:**	**1101.51**
		No. Observations:	**167**
Date:	**Fri, Dec 02 2022**	**Df Residuals:**	**166**
Time:	**14:31:38**	**Df Model:**	**1**

Mean Model

	coef	std err	t	P>|t|	95.0% Conf. Int.
mu	0.3836	0.390	0.985	0.325	[-0.380, 1.147]

Volatility Model

	coef	std err	t	P>|t|	95.0% Conf. Int.
omega	0.4778	1.050	0.455	0.649	[-1.581, 2.537]
alpha[1]	0.0000	5.070e-02	0.000	1.000	[-9.937e-02,9.937e-02]
beta[1]	0.9846	5.557e-02	17.717	3.082e-70	[0.876, 1.094]

Covariance estimator: robust

In [ ]:

```
rolling_pred = []
test_size=365
for i in range(test_size):
  train = returns[:-(test_size-i)]
  model = arch_model(train,p=1,q=0)
  model_fit = model.fit(disp='off')
  pred = model_fit.forecast(horizon=1)
  rolling_pred.append(np.sqrt(pred.variance.values[-1,:][0]))
```

In [ ]:

```
rolling_pred = pd.Series(rolling_pred,index = returns.index[-365:])
```

In [ ]:

```
plt.figure(figsize=(10,4))
true, = plt.plot(returns[-365:])
preds, = plt.plot(rolling_pred)
plt.title("Volatility Prediction - Rolling Forecast",fontsize = 20)
plt.legend(['True Returns','Predicted Volatilty'],fontsize=12,loc='lower left')
```

Out[ ]:

```
<matplotlib.legend.Legend at 0x7fa45d590cd0>
```