# Topic: Forecasting – Time Series

**Instructions**

Please share your answers filled in-line in the word document. Submit code separately wherever applicable.

Please ensure you update all the details:

**Name: Anandakrishnan k v**

**Batch ID:** 19042021

**Topic: Forecasting – Time Series**

**Grading Guidelines:**

**1. An assignment submission is considered complete only when correct and executable code(s) are submitted along with the documentation explaining the method and results. Failing to submit either of those will be considered an invalid submission and will not be considered for evaluation.**

**2. Assignments submitted after the deadline will affect your grades.**

**Grading:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ans** | **Date** |  |  | **Ans** | **Date** |
| Correct | On time | A | 100 |  |  |
| 80% & above | On time | B | 85 | Correct | Late |
| 50% & above | On time | C | 75 | 80% & above | Late |
| 50% & below | On time | D | 65 | 50% & above | Late |
|  |  | E | 55 | 50% & below |  |
| Copied/No Submission |  | F | 45 |  |  |

* **Grade A: (>= 90):** When all assignments are submitted on or before the given deadline.
* **Grade B: (>= 80 and < 90):** 
  + When assignments are submitted on time but less than 80% of problems are completed.

(OR)

* + All assignments are submitted after the deadline.
* **Grade C: (>= 70 and < 80):** 
  + When assignments are submitted on time but less than 50% of the problems are completed.

(OR)

* + Less than 80% of problems in the assignments are submitted after the deadline.
* **Grade D: (>= 60 and < 70):**
  + Assignments submitted after the deadline and with 50% or less problems.
* **Grade E: (>= 50 and < 60):** 
  + Less than 30% of problems in the assignments are submitted after the deadline.

(OR)

* + Less than 30% of problems in the assignments are submitted before the deadline.
* **Grade F: (< 50):** No submission (or) malpractice.

**Hints:**

1. **Business Problem**
   1. **What is the business objective?**
   2. **Are there any constraints?**

**2. Work on each feature of the dataset to create a data dictionary as displayed in the below image:**



**2.1 Make a table as shown above and provide information about the features such as its data type and its relevance to the model building. And if not relevant, provide reasons and a description of the feature**

1. **Data Pre-processing**

**3.1 Data Cleaning, Feature Engineering, etc.**

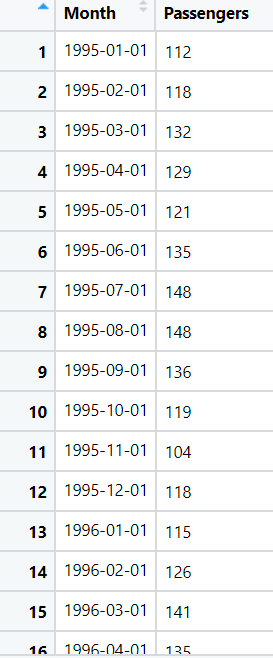
**3.2 Outlier Treatment**

1. **Exploratory Data Analysis (EDA):**
   1. **Summary**
   2. **Identify the trend**
   3. **Identify seasonality**
2. **Model Building:**
   1. **Perform Forecasting on the given datasets (both data driven and moving averages)**
   2. **Apply techniques like exponential smoothing, model-based approach, and ARIMA**
   3. **Briefly explain the output in the documentation for each step (as explained in the class)**
3. **Write about the benefits/impact of the solution - in what way does the business (client) benefit from the solution provided**

**Problem Statement: -**

1. The dataset consists of monthly totals of international airline passengers from 1995 to 2002. Our main aim is to predict the number of passengers for the next five years using time series forecasting. Prepare a document for each model explaining how many dummy variables you have created and also include the RMSE value for each model.

## File: - Airlines.xlsx



**Solution:-**

* **What is the business objective?**

**To predict the number of passengers for the next five years using time series forecasting**

* **Are there any constraints?**

**Maximize: Accuracy of the model**

**Minimize: Complexity of the model**

**Minimize: Time lag for operation**

**Python Code:**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from statsmodels.tsa.seasonal import seasonal\_decompose

from statsmodels.tsa.holtwinters import SimpleExpSmoothing # SES

from statsmodels.tsa.holtwinters import Holt # Holts Exponential Smoothing

from statsmodels.tsa.holtwinters import ExponentialSmoothing #

# from datetime import datetime

cocacola = pd.read\_excel("C://Users//user//Downloads//forecasting//CocaCola\_Sales\_Rawdata.xlsx")

cocacola.Sales.plot() # time series plot

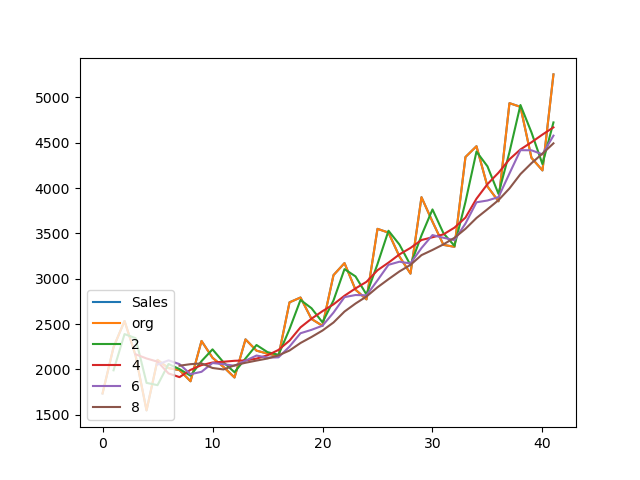
# Centering moving average for the time series

cocacola.Sales.plot(label = "org")

for i in range(2, 9, 2):

cocacola["Sales"].rolling(i).mean().plot(label = str(i))

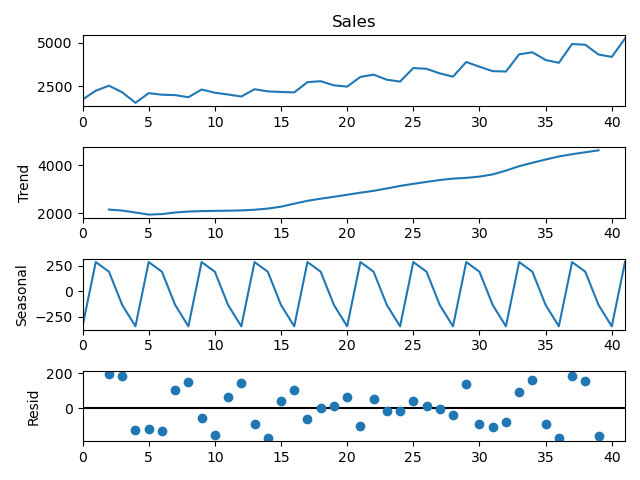
plt.legend(loc = 3)



# Time series decomposition plot

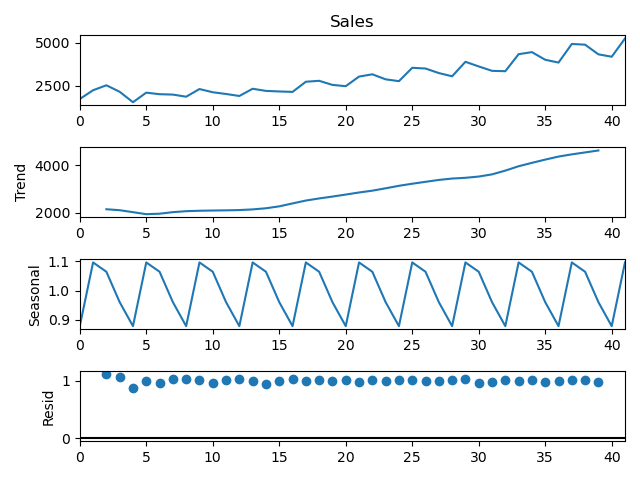
decompose\_ts\_add = seasonal\_decompose(cocacola.Sales, model = "additive", period = 4)

decompose\_ts\_add.plot()



decompose\_ts\_mul = seasonal\_decompose(cocacola.Sales, model = "multiplicative", period = 4)

decompose\_ts\_mul.plot()

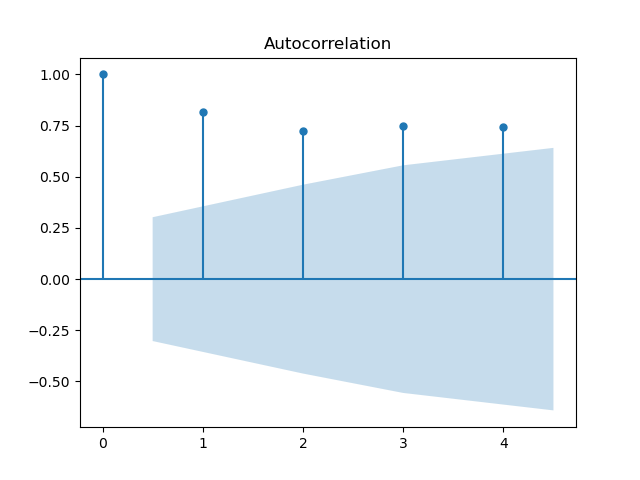


# ACF plot on Original data sets

import statsmodels.graphics.tsaplots as tsa\_plots

tsa\_plots.plot\_acf(cocacola.Sales, lags = 4) # lag 1 time series showing better correlation

# tsa\_plots.plot\_pacf(cocacola.Sales, lags=4)



# splitting the data into Train and Test data

# Recent 4 time period values are Test data

Train = cocacola.head(38)

Test = cocacola.tail(4)

# to change the index value in pandas data frame

Test.set\_index(np.arange(1,5),inplace=True)

# Creating a function to calculate the MAPE value for test data

def MAPE(pred,org):

temp = np.abs((pred-org)/org)\*100

return np.mean(temp)

# Simple Exponential Method

ses\_model = SimpleExpSmoothing(Train["Sales"]).fit()

pred\_ses = ses\_model.predict(start = Test.index[0], end = Test.index[-1])

MAPE(pred\_ses, Test.Sales) # 54.651178181121196

# Holt method

hw\_model = Holt(Train["Sales"]).fit()

pred\_hw = hw\_model.predict(start = Test.index[0], end = Test.index[-1])

MAPE(pred\_hw, Test.Sales) # 50.16565398885506

# Holts winter exponential smoothing with additive seasonality and additive trend

hwe\_model\_add\_add = ExponentialSmoothing(Train["Sales"], seasonal = "add", trend = "add", seasonal\_periods = 4).fit()

pred\_hwe\_add\_add = hwe\_model\_add\_add.predict(start = Test.index[0], end = Test.index[-1])

MAPE(pred\_hwe\_add\_add, Test.Sales) # 53.263950563988786

# Holts winter exponential smoothing with multiplicative seasonality and additive trend

hwe\_model\_mul\_add = ExponentialSmoothing(Train["Sales"], seasonal = "mul", trend = "add", seasonal\_periods = 4).fit()

pred\_hwe\_mul\_add = hwe\_model\_mul\_add.predict(start = Test.index[0], end = Test.index[-1])

MAPE(pred\_hwe\_mul\_add, Test.Sales) # 53.23120487169642

# Final Model on 100% Data

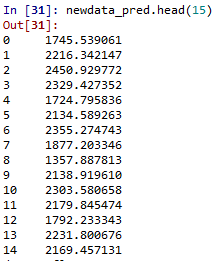
hwe\_model\_add\_add = ExponentialSmoothing(cocacola["Sales"], seasonal = "add", trend = "add", seasonal\_periods = 4).fit()

# Load the new data which includes the entry for future 4 values

new\_data = pd.read\_excel("C://Users//user//Downloads//forecasting//Newdata\_CocaCola\_Sales.xlsx")

newdata\_pred = hwe\_model\_add\_add.predict(start = new\_data.index[0], end = new\_data.index[-1])

newdata\_pred



**Problem Statement: -**

1. The dataset consists of quarterly sales data of Coca-Cola from 1986 to 1996. Predict sales for the next two years by using time series forecasting and prepare a document for each model explaining how many dummy variables you have created and also include the RMSE value for each model.

**File:- CocaCola\_Sales\_RawData.xlsx**



**Solution:-**

**What is the business objective?**

Predict sales for the next two years by using time series forecasting

**Are there any constraints?**

Maximize: Accuracy of the model

Minimize: Complexity of the model

Minimize: Time lag for operation

**Python Code:**

############## airlines ##########

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from statsmodels.tsa.seasonal import seasonal\_decompose

%matplotlib inline

import statsmodels.api as sm

import warnings

from statsmodels.tsa.stattools import adfuller

from statsmodels.tsa.arima\_model import ARIMA

#pip install pystan

#conda install -c conda -forge fbprophet

# Import the AirPassengers dataset

passengers\_p = pd.read\_excel('C://Users//user//Downloads//forecasting//Airlines Data.xlsx')

passengers = passengers\_p.copy(deep= True)

passengers.head()

# converting date column to date time format if it is in string format

passengers.info() # date column ds is already in date tme format

dates = pd.date\_range(start='1995-01-01', freq='MS',periods=len(passengers)) # converting to date time format

#split to month and year column

passengers['Month'] = dates.month

passengers['Year'] = dates.year

passengers.head()

#To get the names of the month

passengers.dtypes

passengers.head()

import calendar

passengers['Month'] = passengers['Month'].apply(lambda x: calendar.month\_abbr[x])

passengers.rename({'#Passengers':'Passengers'},axis=1,inplace=True)

passengers = passengers[['Month','Year','Passengers']]

passengers.head()

# adding date column

passengers['Date'] = dates

passengers.set\_index('Date',inplace=True)

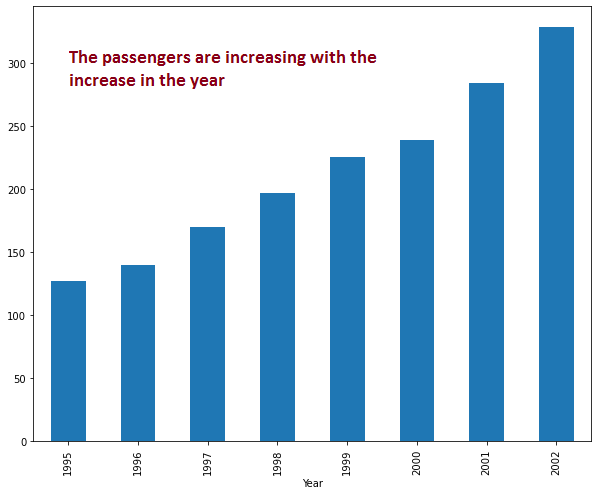
passengers.head()

# Exploratory Data Analysis

plt.figure(figsize=(10,8))

passengers.groupby('Year')['Passengers'].mean().plot(kind='bar')

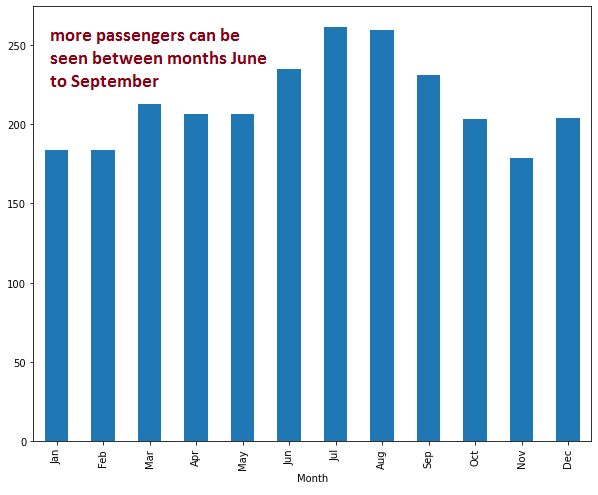
plt.show()



plt.figure(figsize=(10,8))

passengers.groupby('Month')['Passengers'].mean().reindex(index=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']).plot(kind='bar')

plt.show()



#Lets plot the data to see the trend and seasonality

passengers\_count = passengers['Passengers']

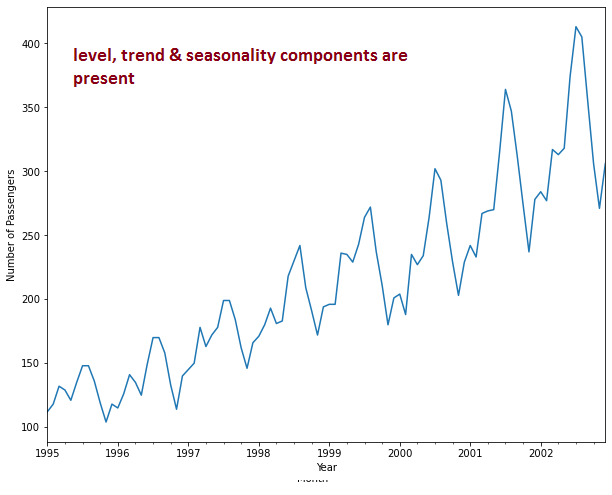
plt.figure(figsize=(10,8))

passengers\_count.plot()

plt.xlabel('Year')

plt.ylabel('Number of Passengers')

plt.show()



#Now we start with time series decomposition of this data to understand underlying patterns such as trend, seasonality, cycle and irregular remainder

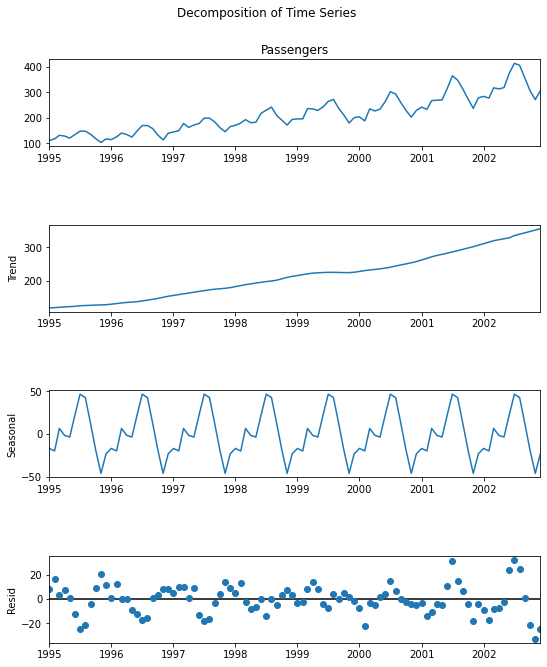
decompose = sm.tsa.seasonal\_decompose(passengers\_count,model='additive',extrapolate\_trend=8)

fig = decompose.plot()

fig.set\_figheight(10)

fig.set\_figwidth(8)

fig.suptitle('Decomposition of Time Series')



#Trend

#Time Series Decomposition: To begin with let's try to decipher trends embedded in the above tractor sales time series. It is clearly evident that there is an overall increasing trend in the data along with some seasonal variations. However, it might not always be possible to make such visual inferences.

#So, more formally, we can check stationarity using the following: Plotting Rolling Statistics: We can plot the moving average or moving variance and see if it varies with time. By moving average/variance we mean that at any instant 't', we'll take the average/variance of the last year, i.e. last 12 months. But again this is more of a visual technique.

#Now, let’s try to remove wrinkles from our time series using moving average. We will take moving average of different time periods i.e. 4,6,8, and 12 months as shown below. Here, moving average is shown in orange and actual series in blue.

# Centering moving average for the time series

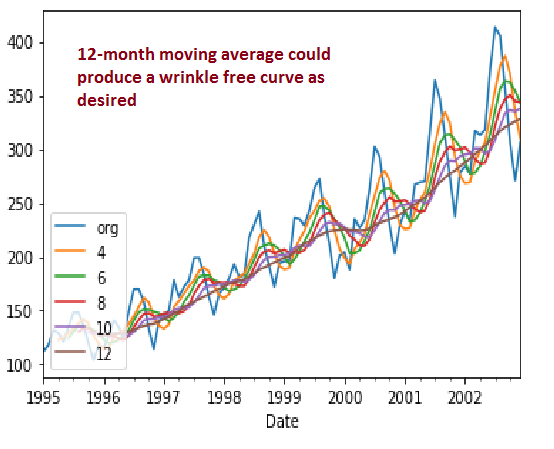
passengers.Passengers.plot(label = "org")

for i in range(4, 13, 2):

passengers['Passengers'].rolling(i).mean().plot(label = str(i))

plt.legend(loc = 3)

# As we could see in the above plots, 12-month moving average could produce a wrinkle free curve as desired. This on some level is expected since we are using month-wise data for our analysis and there is expected monthly-seasonal effect in our data.



#Seasonality

#Let us see how many passengers travelled in flights on a month on month basis. We will plot a stacked annual plot to observe seasonality in our data.

passengers.head()

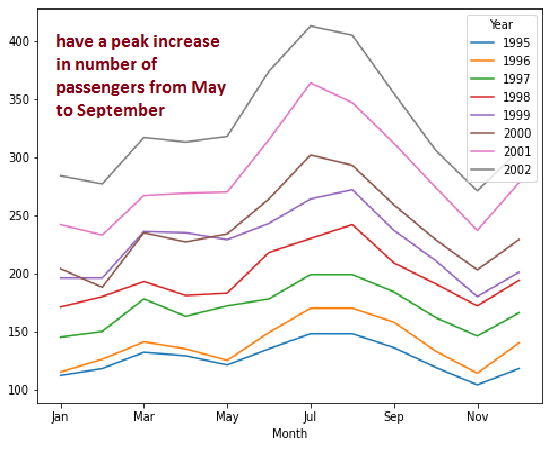
monthly = pd.pivot\_table(data=passengers,values='Passengers',index='Month',columns='Year')

monthly = monthly.reindex(index=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec'])

monthly

monthly.plot(figsize=(8,6))

plt.show()



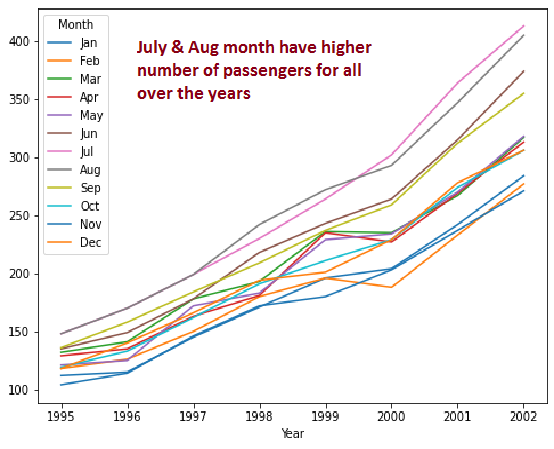
yearly = pd.pivot\_table(data=passengers,values='Passengers',index='Year',columns='Month')

yearly = yearly[['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']]

yearly

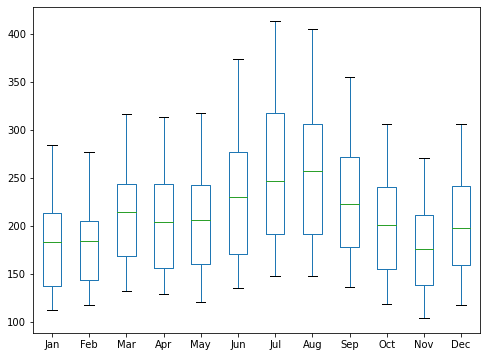
yearly.plot(figsize=(8,6))

plt.show()



yearly.plot(kind='box',figsize=(8,6))

plt.show()



# Important Inferences

# The passengers are increasing without fail every year.

# July and August are the peak months for passengers.

# We can see a seasonal cycle of 12 months where the mean value of each month starts with a increasing trend in the beginning of the year and drops down towards the end of the year. We can see a seasonal effect with a cycle of 12 months.

############# ARIMA Modelling ########

# The most important assumption of auto regressive method is that the TS data should be stationary.

# There are two primary way to determine whether a given time series is stationary.

# 1.Rolling Statistics

# 2.Augmented Dickey-Fuller Test

# Rolling Statistics

# Plot the rolling mean and rolling standard deviation. The time series is stationary if they remain constant with time (with the naked eye look to see if the lines are straight and parallel to the x-axis).

df = passengers\_p

#eliminating index column

df.reset\_index()

df.set\_index('Month',inplace=True)

df.head()

#checking for bad data rows

df.tail()

rolling\_mean = df.rolling(window = 12).mean()

rolling\_std = df.rolling(window = 12).std()

plt.plot(df, color = 'blue', label = 'Original')

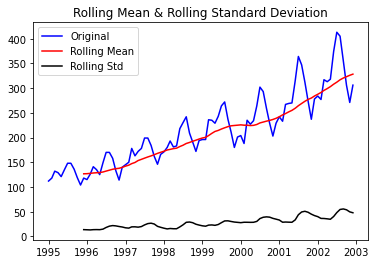
plt.plot(rolling\_mean, color = 'red', label = 'Rolling Mean')

plt.plot(rolling\_std, color = 'black', label = 'Rolling Std')

plt.legend(loc = 'best')

plt.title('Rolling Mean & Rolling Standard Deviation')

plt.show()



# As you can see, the rolling mean and rolling standard deviation increase with time. Therefore, we can conclude that the time series is not stationary.

# Dickey-Fuller Test

# The time series is considered stationary if the p-value is low (according to the null hypothesis) and the critical values at 1%, 5%, 10% confidence intervals are as close as possible to the ADF Statistics

# Perform Dickey-Fuller test:

from statsmodels.tsa.stattools import adfuller

result = adfuller(df['Passengers'])

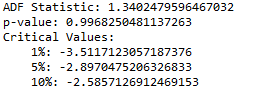
print('ADF Statistic: {}'.format(result[0]))

print('p-value: {}'.format(result[1]))

print('Critical Values:')

for key, value in result[4].items():

print('\t{}: {}'.format(key, value))

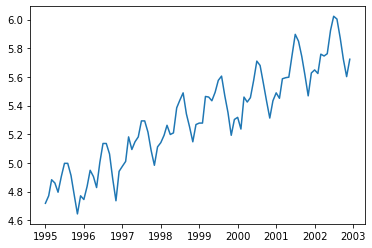


#The ADF Statistic is far from the critical values and the p-value is greater than the threshold (0.05). Thus, we can conclude that the time series is not stationary.

#Taking the log of the dependent variable is as simple way of lowering the rate at which rolling mean increases.

df\_log = np.log(df)

plt.plot(df\_log)



#Let’s create a function to run the two tests which determine whether a given time series is stationary.

def get\_stationarity(timeseries):

# rolling statistics

rolling\_mean = timeseries.rolling(window=12).mean()

rolling\_std = timeseries.rolling(window=12).std()

# rolling statistics plot

original = plt.plot(timeseries, color='blue', label='Original')

mean = plt.plot(rolling\_mean, color='red', label='Rolling Mean')

std = plt.plot(rolling\_std, color='black', label='Rolling Std')

plt.legend(loc='best')

plt.title('Rolling Mean & Standard Deviation')

plt.show(block=False)

# Dickey–Fuller test:

result = adfuller(timeseries['Passengers'])

print('ADF Statistic: {}'.format(result[0]))

print('p-value: {}'.format(result[1]))

print('Critical Values:')

for key, value in result[4].items():

print('\t{}: {}'.format(key, value))

#There are multiple transformations that we can apply to a time series to render it stationary. For instance, we subtract the rolling mean.

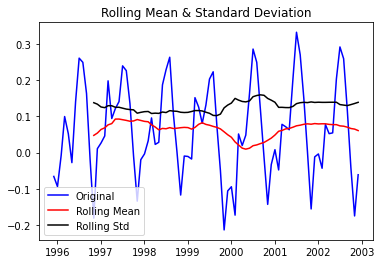
rolling\_mean = df\_log.rolling(window=12).mean()

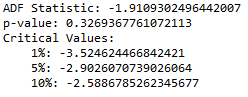
df\_log\_minus\_mean = df\_log - rolling\_mean

df\_log\_minus\_mean.dropna(inplace=True)

get\_stationarity(df\_log\_minus\_mean)

#As we can see, after subtracting the mean, the rolling mean and standard deviation are approximately horizontal. The p-value is below the threshold of 0.05 and the ADF Statistic is close to the critical values. Therefore, the time series is stationary.





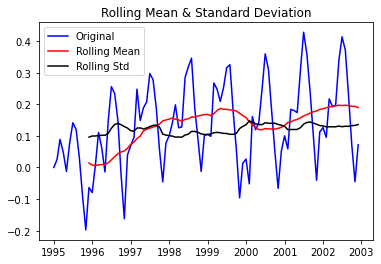
#Applying exponential decay is another way of transforming a time series such that it is stationary.

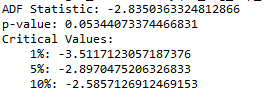
rolling\_mean\_exp\_decay = df\_log.ewm(halflife=12, min\_periods=0, adjust=True).mean()

df\_log\_exp\_decay = df\_log - rolling\_mean\_exp\_decay

df\_log\_exp\_decay.dropna(inplace=True)

get\_stationarity(df\_log\_exp\_decay)



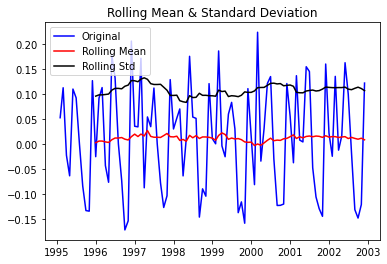


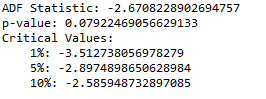
# applying time shifting, we subtract every the point by the one that preceded it(1 difference log method).

df\_log\_shift = df\_log - df\_log.shift()

df\_log\_shift.dropna(inplace=True)

get\_stationarity(df\_log\_shift)





# applying time shifting, we subtract every the point by the six that preceded it.

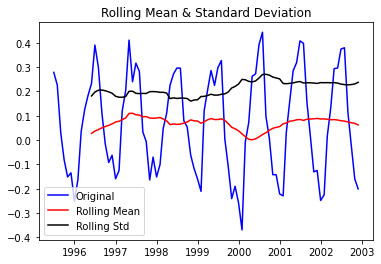
df\_log\_shift\_6 = df\_log - df\_log.shift(6)

df\_log\_shift\_6.dropna(inplace=True)

get\_stationarity(df\_log\_shift\_6)

df\_log\_shift\_6.dropna(inplace=True)

get\_stationarity(df\_log\_shift\_6)



# from all the above rolling statistics plot of 1 difference log method showing better performace. so choosing it

#ARIMA is a combination of 3 parts i.e. AR (AutoRegressive), I (Integrated), and

#MA (Moving Average). A convenient notation for ARIMA model is ARIMA(p,d,q). Here p,d, and q

#are the levels for each of the AR, I, and MA parts. Each of these three parts is an effort

# to make the final residuals display a white noise pattern (or no pattern at all). In each

#step of ARIMA modeling, time series data is passed through these 3 parts

#### Identification of best fit ARIMA model from acf & pacf analysis ###

#Identification of an AR model is often best done with the PACF.

#For an AR model, the theoretical PACF “shuts off” past the order of the model. The phrase “shuts off” means that in theory the partial autocorrelations are equal to 0 beyond that point. Put another way, the number of non-zero partial autocorrelations gives the order of the AR model. By the “order of the model” we mean the most extreme lag of x that is used as a predictor.

#Identification of an MA model is often best done with the ACF rather than the PACF.

#For an MA model, the theoretical PACF does not shut off, but instead tapers toward 0 in some manner. A clearer pattern for an MA model is in the ACF. The ACF will have non-zero autocorrelations only at lags involved in the model.

#p,d,q p AR model lags d differencing q MA lags

# Time series decomposition plot

from statsmodels.graphics.tsaplots import plot\_acf,plot\_pacf

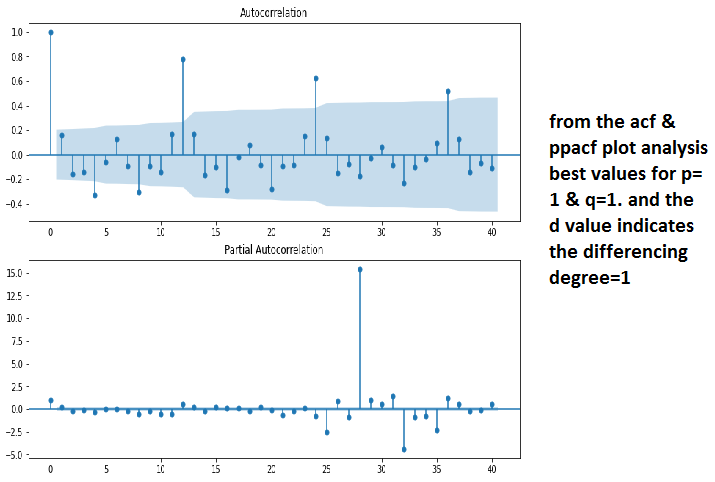
fig = plt.figure(figsize=(12,8))

ax1 = fig.add\_subplot(211)

fig = sm.graphics.tsa.plot\_acf(df\_log\_shift.iloc[2:],lags=40,ax=ax1)

ax2 = fig.add\_subplot(212)

fig = sm.graphics.tsa.plot\_pacf(df\_log\_shift.iloc[2:],lags=40,ax=ax2)



# from the acf & ppacf plot analysis best values for p=1 & q=1. and the d value indicates the differencing degree=1

# we are gonna cross check the p,q & d values through iterative process

#### Identification of best fit ARIMA model from Iterative process ###

#Iterate the process to find the best values for p, d, q

p=1

q=0

d=1

pdq=[]

aic=[]

for q in range(12):

try:

model = ARIMA(df\_log\_shift, order = (p, d, q)).fit(disp = 0)

x=model.aic

x1= p,d,q

aic.append(x)

pdq.append(x1)

except:

pass

keys = pdq

values = aic

d = dict(zip(keys, values))

print (d)

C:\Users\user\Documents\matplot\Untitled.png

#Best SARIMAX(1,1,7) model - AIC:-173.58099220149512 The best fit model is selected based on Akaike Information Criterion (AIC) , and Bayesian Information Criterion (BIC) values. The idea is to choose a model with minimum AIC and BIC values.

#Predict sales on in-sample date using the best fit ARIMA model

#The next step is to predict passengers for in-sample data and find out how close is the model prediction on the in-sample data to the actual truth.

# checking both the pdq values set got from previous 2 analysis

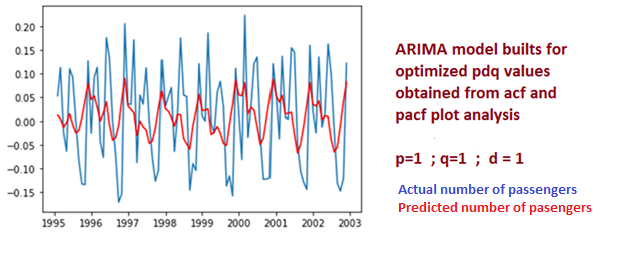
# compares the ARIMA model buits for pdq values from acf & pacf analysis to the original time series

model = ARIMA(df\_log, order=(1,1,1))

results\_acf = model.fit(disp=-1)

plt.plot(df\_log\_shift)

plt.plot(results\_acf.fittedvalues, color='red')



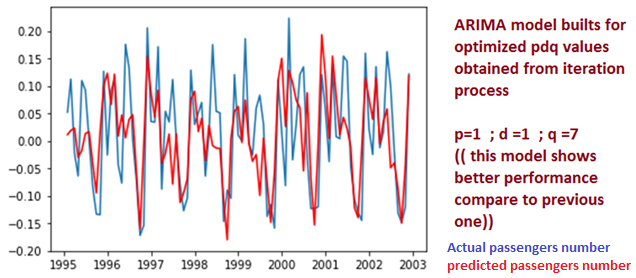
# compares the ARIMA model buits for pdq values from iteration to the original time series

model = ARIMA(df\_log, order=(1,1,7))

results\_iter = model.fit(disp=-1)

plt.plot(df\_log\_shift)

plt.plot(results\_iter.fittedvalues, color='red')



# the ARIMA model buits from iteration suits more with thoriginal time series. so take it as final model

# create and fit an ARIMA model with AR of order 1, differencing of order 1 and MA of order 7.

# the original output data response of the ARIMA model(eliminating log & differencing effect)

predictions\_ARIMA\_diff = pd.Series(results\_iter.fittedvalues, copy=True)

predictions\_ARIMA\_diff\_cumsum = predictions\_ARIMA\_diff.cumsum()

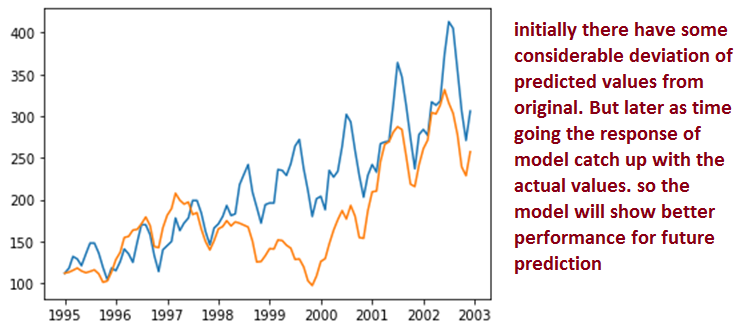
predictions\_ARIMA\_log = pd.Series(df\_log['Passengers'].iloc[0], index=df\_log.index)

predictions\_ARIMA\_log = predictions\_ARIMA\_log.add(predictions\_ARIMA\_diff\_cumsum, fill\_value=0)

predictions\_ARIMA = np.exp(predictions\_ARIMA\_log)

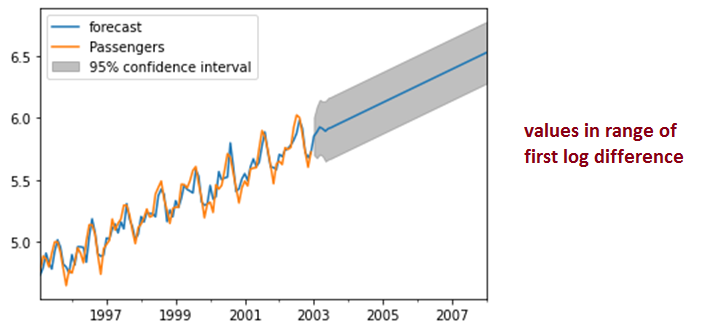
plt.plot(df)

plt.plot(predictions\_ARIMA)



# Given that we have data going for every month going back 12 years and want to forecast the number of passengers for the next 5 years, we use (12 x 8)+ (12 x 5) = 156.

results\_iter.plot\_predict(1,156) # in log fisrt difference value range



**Problem Statement: -**

A plastics manufacturing plant has recorded their monthly sales data from 1949 to 1953. Perform forecasting on the data and bring out insights from it and forecast the sale for the next year.

Plastic Sales.csv

A picture containing table

Description automatically generated

**Solution:-**

**What is the business objective?**

Perform forecasting on the plastics manufacturing plant monthly sales data and bring out insights from it and forecast the sale for the next year

**Are there any constraints?**

Maximize: Accuracy of the model

Minimize: Complexity of the model

Minimize: Time lag for operation

**Python Code:**

############## plastic ##########

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from statsmodels.tsa.seasonal import seasonal\_decompose

%matplotlib inline

import statsmodels.api as sm

import warnings

from statsmodels.tsa.stattools import adfuller

from statsmodels.tsa.arima\_model import ARIMA

#pip install pystan

#conda install -c conda -forge fbprophet

# Import the AirPassengers dataset

plastic\_p = pd.read\_csv('C://Users//user//Downloads//forecasting//PlasticSales.csv')

plastic = plastic\_p.copy(deep= True)

plastic.head()

# converting date column to date time format if it is in string format

plastic.info() # date column ds is already in date tme format

dates = pd.date\_range(start='1949-01-01', freq='MS',periods=len(plastic)) # converting to date time format

#split to month and year column

plastic['Month'] = dates.month

plastic['Year'] = dates.year

plastic.head()

#To get the names of the month

plastic.dtypes

plastic.head()

import calendar

plastic['Month'] = plastic['Month'].apply(lambda x: calendar.month\_abbr[x])

plastic.rename({'#Passengers':'Passengers'},axis=1,inplace=True)

plastic = plastic[['Month','Year','Sales']]

plastic.head()

# adding date column

plastic['Date'] = dates

plastic.set\_index('Date',inplace=True)

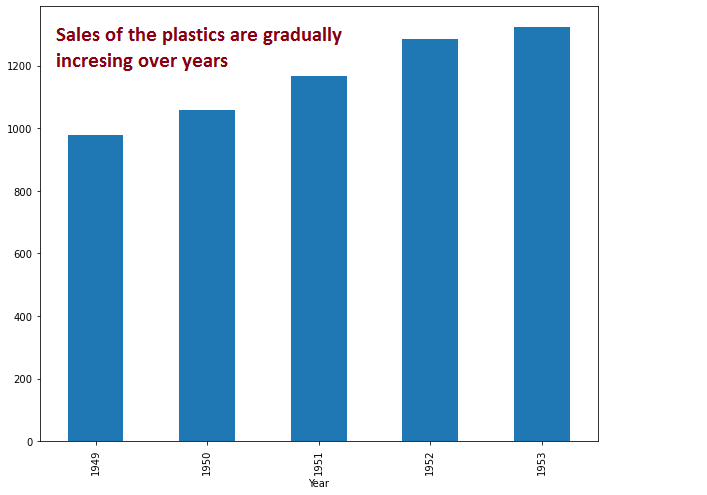
plastic.head()

# Exploratory Data Analysis

plt.figure(figsize=(10,8))

plastic.groupby('Year')['Sales'].mean().plot(kind='bar')

plt.show()



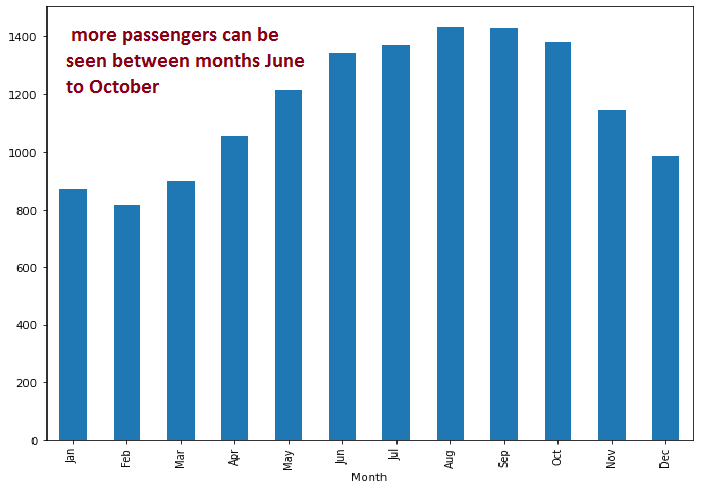
# From the above figure we can see that Sales are increasing with the increase in the year

plt.figure(figsize=(10,8))

plastic.groupby('Month')['Sales'].mean().reindex(index=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']).plot(kind='bar')

plt.show()

# From the above figure we can see that more passengers can be seen between months June to October

****

#Lets plot the data to see the trend and seasonality

plastic\_count = plastic['Sales']

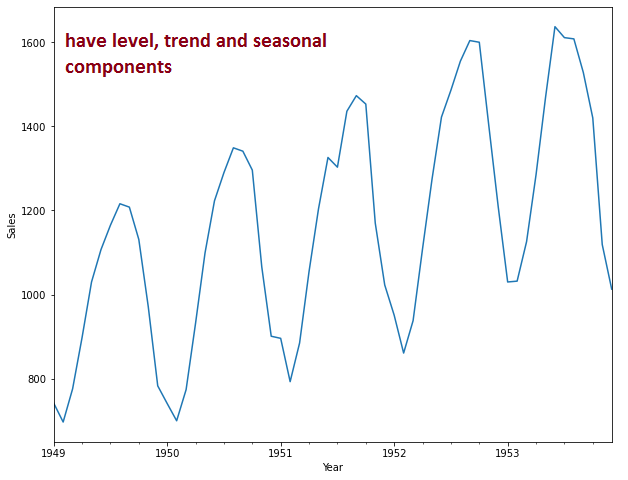
plt.figure(figsize=(10,8))

plastic\_count.plot()

plt.xlabel('Year')

plt.ylabel('Sales')

plt.show()



#Now we start with time series decomposition of this data to understand underlying patterns such as trend, seasonality, cycle and irregular remainder

decompose = sm.tsa.seasonal\_decompose(plastic\_count,model='additive',extrapolate\_trend=8)

fig = decompose.plot()

fig.set\_figheight(10)

fig.set\_figwidth(8)

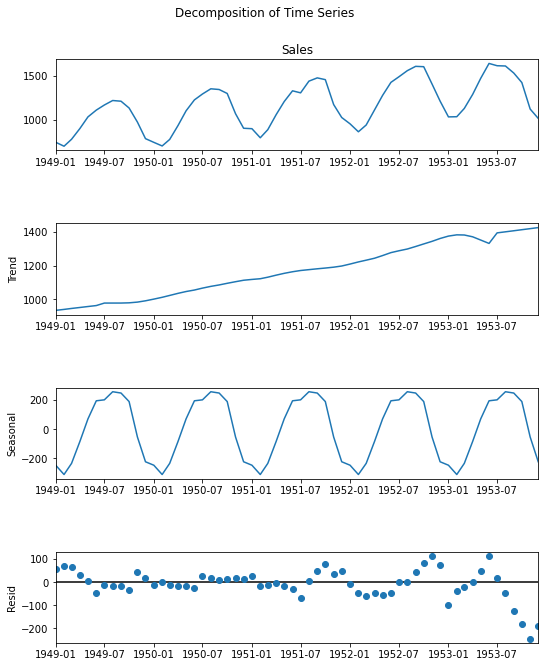
fig.suptitle('Decomposition of Time Series')

#Trend

#Time Series Decomposition: To begin with let's try to decipher trends embedded in the above tractor sales time series. It is clearly evident that there is an overall increasing trend in the data along with some seasonal variations. However, it might not always be possible to make such visual inferences.

#So, more formally, we can check stationarity using the following: Plotting Rolling Statistics: We can plot the moving average or moving variance and see if it varies with time. By moving average/variance we mean that at any instant 't', we'll take the average/variance of the last year, i.e. last 12 months. But again this is more of a visual technique.

#Now, let’s try to remove wrinkles from our time series using moving average. We will take moving average of different time periods i.e. 4,6,8, and 12 months as shown below. Here, moving average is shown in orange and actual series in blue.



# Centering moving average for the time series

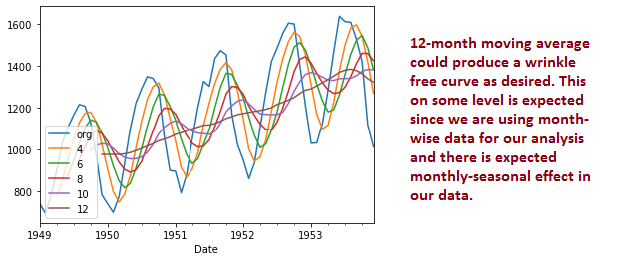
plastic.Sales.plot(label = "org")

for i in range(4, 13, 2):

plastic['Sales'].rolling(i).mean().plot(label = str(i))

plt.legend(loc = 3)

# As we could see in the above plots, 12-month moving average could produce a wrinkle free curve as desired. This on some level is expected since we are using month-wise data for our analysis and there is expected monthly-seasonal effect in our data.



#Seasonality

#Let us see how many passengers travelled in flights on a month on month basis. We will plot a stacked annual plot to observe seasonality in our data.

plastic.head()

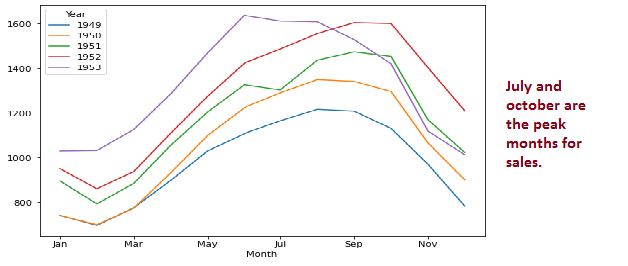
monthly = pd.pivot\_table(data=plastic,values='Sales',index='Month',columns='Year')

monthly = monthly.reindex(index=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec'])

monthly

monthly.plot(figsize=(8,6))

plt.show()



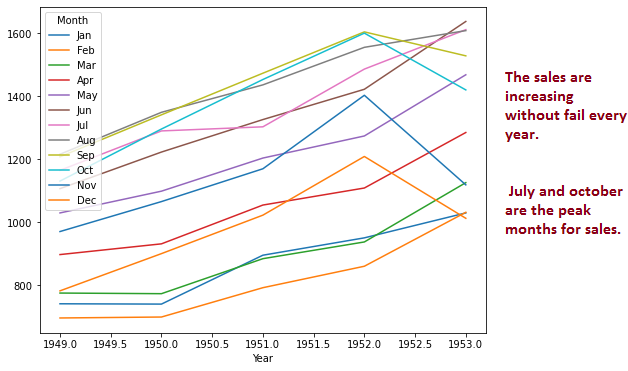
yearly = pd.pivot\_table(data=plastic,values='Sales',index='Year',columns='Month')

yearly = yearly[['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']]

yearly

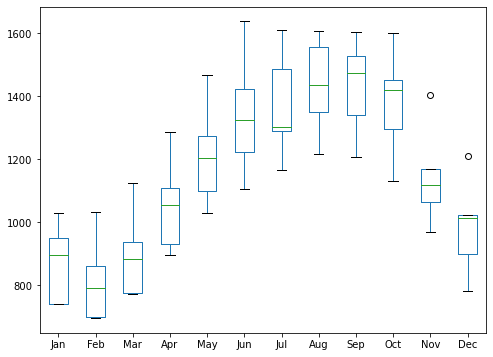
yearly.plot(figsize=(8,6))

plt.show()



yearly.plot(kind='box',figsize=(8,6))

plt.show()



# Important Inferences

# The sales are increasing without fail every year.

# July and october are the peak months for sales.

# We can see a seasonal cycle of 12 months where the mean value of each month starts with a increasing trend in the beginning of the year and drops down towards the end of the year. We can see a seasonal effect with a cycle of 12 months.

############# ARIMA Modelling ########

# The most important assumption of auto regressive method is that the TS data should be stationary.

# There are two primary way to determine whether a given time series is stationary.

# 1.Rolling Statistics

# 2.Augmented Dickey-Fuller Test

# Rolling Statistics

# Plot the rolling mean and rolling standard deviation. The time series is stationary if they remain constant with time (with the naked eye look to see if the lines are straight and parallel to the x-axis).

df = plastic[["Sales"]]

#checking for bad data rows

df.tail()

rolling\_mean = df.rolling(window = 12).mean()

rolling\_std = df.rolling(window = 12).std()

plt.plot(df, color = 'blue', label = 'Original')

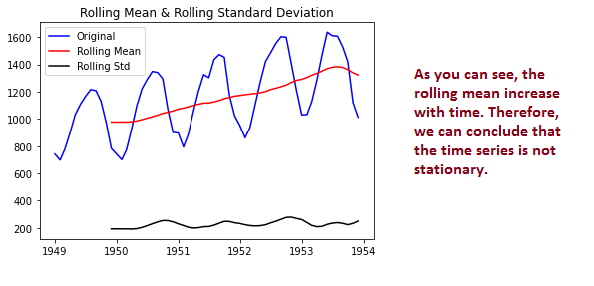
plt.plot(rolling\_mean, color = 'red', label = 'Rolling Mean')

plt.plot(rolling\_std, color = 'black', label = 'Rolling Std')

plt.legend(loc = 'best')

plt.title('Rolling Mean & Rolling Standard Deviation')

plt.show()



# Dickey-Fuller Test

# The time series is considered stationary if the p-value is low (according to the null hypothesis) and the critical values at 1%, 5%, 10% confidence intervals are as close as possible to the ADF Statistics

# Perform Dickey-Fuller test:

from statsmodels.tsa.stattools import adfuller

result = adfuller(df['Sales'])

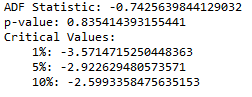
print('ADF Statistic: {}'.format(result[0]))

print('p-value: {}'.format(result[1]))

print('Critical Values:')

for key, value in result[4].items():

print('\t{}: {}'.format(key, value))



#The ADF Statistic is far from the critical values and the p-value is greater than the threshold (0.05). Thus, we can conclude that the time series is not stationary.

#Taking the log of the dependent variable is as simple way of lowering the rate at which rolling mean increases.

df\_log = np.log(df)

plt.plot(df\_log)

#Taking the log of the dependent variable is as simple way of lowering the rate at which rolling mean increases.

df\_log = np.log(df)

plt.plot(df\_log)

#Let’s create a function to run the two tests which determine whether a given time series is stationary.

def get\_stationarity(timeseries):

# rolling statistics

rolling\_mean = timeseries.rolling(window=12).mean()

rolling\_std = timeseries.rolling(window=12).std()

# rolling statistics plot

original = plt.plot(timeseries, color='blue', label='Original')

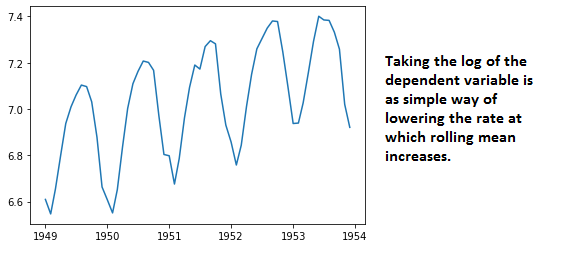
mean = plt.plot(rolling\_mean, color='red', label='Rolling Mean')

std = plt.plot(rolling\_std, color='black', label='Rolling Std')

plt.legend(loc='best')

plt.title('Rolling Mean & Standard Deviation')

plt.show(block=False)



# Dickey–Fuller test:

result = adfuller(timeseries['Sales'])

print('ADF Statistic: {}'.format(result[0]))

print('p-value: {}'.format(result[1]))

print('Critical Values:')

for key, value in result[4].items():

print('\t{}: {}'.format(key, value))

#There are multiple transformations that we can apply to a time series to render it stationary. For instance, we subtract the rolling mean.

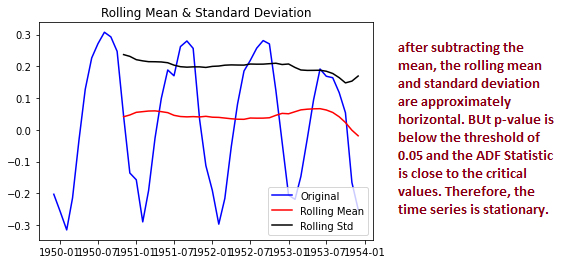
rolling\_mean = df\_log.rolling(window=12).mean()

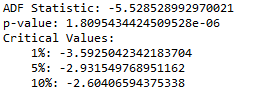
df\_log\_minus\_mean = df\_log - rolling\_mean

df\_log\_minus\_mean.dropna(inplace=True)

get\_stationarity(df\_log\_minus\_mean)

#As we can see, after subtracting the mean, the rolling mean and standard deviation are approximately horizontal. The p-value is below the threshold of 0.05 and the ADF Statistic is close to the critical values. Therefore, the time series is stationary





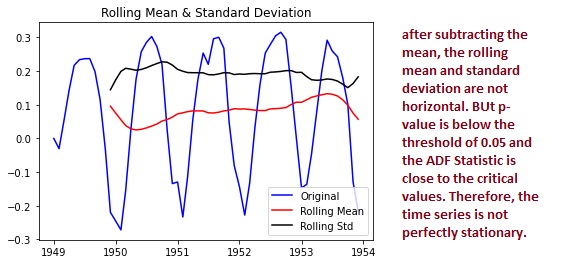
#Applying exponential decay is another way of transforming a time series such that it is stationary.

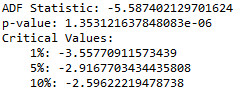
rolling\_mean\_exp\_decay = df\_log.ewm(halflife=12, min\_periods=0, adjust=True).mean()

df\_log\_exp\_decay = df\_log - rolling\_mean\_exp\_decay

df\_log\_exp\_decay.dropna(inplace=True)

get\_stationarity(df\_log\_exp\_decay)



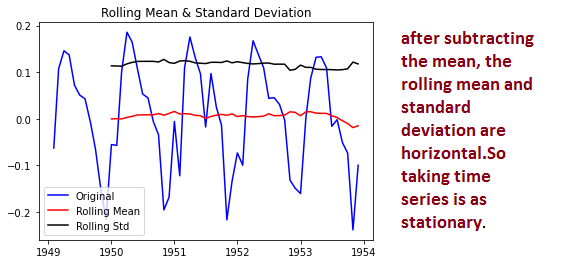


# applying time shifting, we subtract every the point by the one that preceded it(1 difference log method).

df\_log\_shift = df\_log - df\_log.shift()

df\_log\_shift.dropna(inplace=True)

get\_stationarity(df\_log\_shift)



# from all the above rolling statistics plot of 1 difference log method showing better performace. so choosing it

#ARIMA is a combination of 3 parts i.e. AR (AutoRegressive), I (Integrated), and

#MA (Moving Average). A convenient notation for ARIMA model is ARIMA(p,d,q). Here p,d, and q

#are the levels for each of the AR, I, and MA parts. Each of these three parts is an effort

# to make the final residuals display a white noise pattern (or no pattern at all). In each

#step of ARIMA modeling, time series data is passed through these 3 parts

#### Identification of best fit ARIMA model from acf & pacf analysis ###

#Identification of an AR model is often best done with the PACF.

#For an AR model, the theoretical PACF “shuts off” past the order of the model. The phrase “shuts off” means that in theory the partial autocorrelations are equal to 0 beyond that point. Put another way, the number of non-zero partial autocorrelations gives the order of the AR model. By the “order of the model” we mean the most extreme lag of x that is used as a predictor.

#Identification of an MA model is often best done with the ACF rather than the PACF.

#For an MA model, the theoretical PACF does not shut off, but instead tapers toward 0 in some manner. A clearer pattern for an MA model is in the ACF. The ACF will have non-zero autocorrelations only at lags involved in the model.

#p,d,q p AR model lags d differencing q MA lags

# Time series decomposition plot

from statsmodels.graphics.tsaplots import plot\_acf,plot\_pacf

fig = plt.figure(figsize=(12,8))

ax1 = fig.add\_subplot(211)

fig = sm.graphics.tsa.plot\_acf(df\_log\_shift.iloc[2:],lags=40,ax=ax1)

ax2 = fig.add\_subplot(212)

fig = sm.graphics.tsa.plot\_pacf(df\_log\_shift.iloc[2:],lags=25,ax=ax2)

# from the acf & ppacf plot analysis best values for p=1 & q=1. and the d value indicates the differencing degree=1

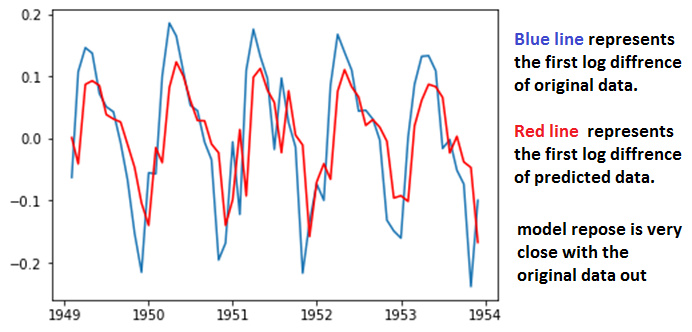
# ARIMA model builts for pdq values from acf & pacf analysis to the original time series

model = ARIMA(df\_log, order=(1,1,1))

results = model.fit()

plt.plot(df\_log\_shift)

plt.plot(results.fittedvalues, color='red')



# the original output data response of the ARIMA model(eliminating log & differencing effect)

predictions\_ARIMA\_diff = pd.Series(results.fittedvalues, copy=True)

predictions\_ARIMA\_diff\_cumsum = predictions\_ARIMA\_diff.cumsum()

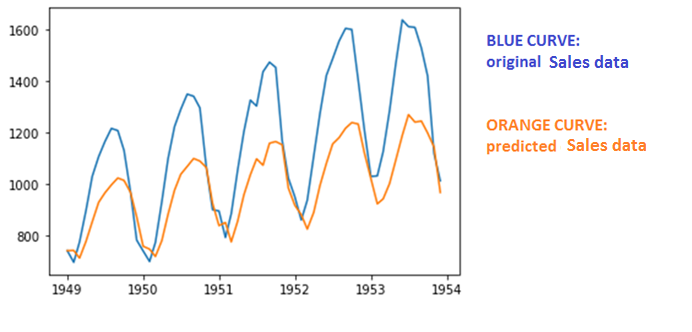
predictions\_ARIMA\_log = pd.Series(df\_log['Sales'].iloc[0], index=df\_log.index)

predictions\_ARIMA\_log = predictions\_ARIMA\_log.add(predictions\_ARIMA\_diff\_cumsum, fill\_value=0)

predictions\_ARIMA = np.exp(predictions\_ARIMA\_log)

plt.plot(df)

plt.plot(predictions\_ARIMA)



# Given that we have data going for every month going back 12 years and want to forecast the number of passengers for the next 1 years, we use (12 x 5)+ (12 x 1) = 72.

results.plot\_predict(1,72) # in log fisrt difference value range

