**Tutorial – 01**

**TIME SERIES ANALYSIS USING PYTHON**

**(Airline passenger time series data)**

Name: Neel Dissanayake

**CONTENT**

1. Time series analysis

2. Python

3. Data set

4. Data analysis

1. **Time series analysis**

In mathematics, a time series is a series of data points indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time. Thus it is a sequence of discrete-time data. Examples of time series are heights of ocean tides, counts of sunspots, and the daily closing value of the Dow Jones Industrial Average.

A time series is very frequently plotted via a run chart (which is a temporal line chart). Time series are used in statistics, signal processing, pattern recognition, econometrics, mathematical finance, weather forecasting, earthquake prediction, electroencephalography, control engineering, astronomy, communications engineering, and largely in any domain of applied science and engineering which involves temporal measurements.

Time series analysis comprises methods for analyzing time series data in order to extract meaningful statistics and other characteristics of the data. Time series forecasting is the use of a model to predict future values based on previously observed values. While regression analysis is often employed in such a way as to test relationships between one or more different time series, this type of analysis is not usually called "time series analysis", which refers in particular to relationships between different points in time within a single series. Interrupted time series analysis is used to detect changes in the evolution of a time series from before to after some intervention which may affect the underlying variable.

Time series data have a natural temporal ordering. This makes time series analysis distinct from cross-sectional studies, in which there is no natural ordering of the observations (e.g. explaining people's wages by reference to their respective education levels, where the individuals' data could be entered in any order). Time series analysis is also distinct from spatial data analysis where the observations typically relate to geographical locations (e.g. accounting for house prices by the location as well as the intrinsic characteristics of the houses). A stochastic model for a time series will generally reflect the fact that observations close together in time will be more closely related than observations further apart. In addition, time series models will often make use of the natural one-way ordering of time so that values for a given period will be expressed as deriving in some way from past values, rather than from future values.

Time series analysis can be applied to real-valued, continuous data, discrete numeric data, or discrete symbolic data.

1. **PYTHON PROGRAMMING**

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

Python is Interpreted − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

Python is Interactive − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Python is Object-Oriented − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

Python is a Beginner's Language − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

**History of Python**

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

**Python Features**

Python's features include − Easy-to-learn − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.

Easy-to-read − Python code is more clearly defined and visible to the eyes.

Easy-to-maintain − Python's source code is fairly easy-to-maintain.

A broad standard library − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.

Interactive Mode − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.

Portable − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

Extendable − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.

Databases − Python provides interfaces to all major commercial databases.

GUI Programming − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

Scalable − Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below –

It supports functional and structured programming methods as well as OOP.

It can be used as a scripting language or can be compiled to byte-code for building large applications.

It provides very high-level dynamic data types and supports dynamic type checking.

It supports automatic garbage collection.

It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

1. **DATA SET**

Across industries, organizations commonly use time series data, which means any information collected over a regular interval of time, in their operations. Examples include daily stock prices, energy consumption rates, social media engagement metrics and retail demand, among others. Analyzing time series data yields insights like trends, seasonal patterns and forecasts into future events that can help generate profits. For example, by understanding the seasonal trends in demand for retail products, companies can plan promotions to maximize sales throughout the year.

When analyzing time series data, you should undertake a number of steps. First, you need to check for stationarity and autocorrelation. Stationarity is a way to measure if the data has structural patterns like seasonal trends. Autocorrelation occurs when future values in a time series linearly depend on past values. You need to check for both of these in time series data because they’re assumptions that are made by many widely used methods in time series analysis. For example, the autoregressive integrated moving average (ARIMA) method for forecasting time series assumes stationarity. Further, linear regression for time series forecasting assumes that the data has no autocorrelation. Before conducting these processes, then, you need to know if the data is viable for the analysis.

During a time series analysis in Python, you also need to perform trend decomposition and forecast future values. Decomposition allows you to visualize trends in your data, which is a great way to clearly explain their behavior. Finally, forecasting allows you to anticipate future events that can aid in decision making. You can use many different techniques for time series forecasting, but here, we will discuss the autoregressive integrated moving average (ARIMA).

We will be working with publicly available airline passenger time series data. There is a part of data table as follows.

|  |  |
| --- | --- |
| Month | #Passengers |
| 1949-01 | 112 |
| 1949-02 | 118 |
| 1949-03 | 132 |
| 1949-04 | 129 |
| 1949-05 | 121 |
| 1949-06 | 135 |
| 1949-07 | 148 |
| 1949-08 | 148 |
| 1949-09 | 136 |
| 1949-10 | 119 |
| 1949-11 | 104 |
| 1949-12 | 118 |
| 1950-01 | 115 |
| 1950-02 | 126 |
| 1950-03 | 141 |
| 1950-04 | 135 |
| 1950-05 | 125 |
| 1950-06 | 149 |
| 1950-07 | 170 |
| 1950-08 | 170 |
| 1950-09 | 158 |
| 1950-10 | 133 |
| 1950-11 | 114 |
| 1950-12 | 140 |
| 1951-01 | 145 |
| 1951-02 | 150 |
| 1951-03 | 178 |
| 1951-04 | 163 |
| 1951-05 | 172 |
| 1951-06 | 178 |
| 1951-07 | 199 |
| 1951-08 | 199 |
| …………. | …… |

1. **DATA ANALYSIS (Time series analysis)**

**Reading and Displaying Data**

To start, let’s import the Pandas library and read the airline passenger data into a data frame:

df = pd.read\_csv("C:\\Users\\neeld\Desktop\Study\Tutorial\_1\Time series analysis\_1\AirPassengers.csv")

Now, let’s display the Ten five rows of data using the data frame head() method:

print(df.head(10))

Month #Passengers

0 1949-01 112

1 1949-02 118

2 1949-03 132

3 1949-04 129

4 1949-05 121

5 1949-06 135

6 1949-07 148

7 1949-08 148

8 1949-09 136

9 1949-10 119

We can see that the data contains a column labeled “Month” that contains dates. In that column, the dates are formatted as year–month. We also see that the data starts in the year 1949.

The second column is labeled “#Passengers,” and it contains the number of passengers for the year–month. Let’s take a look at the last five records the data using the tail() method:

print(df.tail(10))

Month #Passengers

134 1960-03 419

135 1960-04 461

136 1960-05 472

137 1960-06 535

138 1960-07 622

139 1960-08 606

140 1960-09 508

141 1960-10 461

142 1960-11 390

143 1960-12 432

We see from this process that the data ends in 1960.

The next thing we will want to do is convert the month column into a datetime object. This will allow it to programmatically pull time values like the year or month for each record. To do this, we use the Pandas to\_datetime() method:

df['Month'] = pd.to\_datetime(df['Month'], format='%Y-%m')  
print(df.head())

Month #Passengers

0 1949-01-01 112

1 1949-02-01 118

2 1949-03-01 132

3 1949-04-01 129

4 1949-05-01 121

Note that this process automatically inserts the first day of each month, which is basically a dummy value since we have no daily passenger data.

The next thing we can do is convert the month column to an index. This will allow us to more easily work with some of the packages we will be covering later:

df.index = df['Month']  
del df['Month']  
print(df.head())

#Passengers

Month

1949-01-01 112

1949-02-01 118

1949-03-01 132

1949-04-01 129

1949-05-01 121

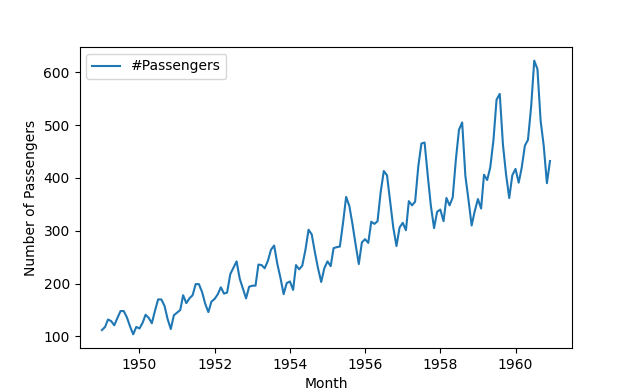
Next, let’s generate a time series plot using Seaborn and Matplotlib. This will allow us to visualize the time series data. First, let’s import Matplotlib and Seaborn:

import matplotlib.pyplot as plt  
import seaborn as sns

Next, let’s generate a line plot using Seaborn:  
  
sns.lineplot(df)

And label the y-axis with Matplotlib:

plt.ylabel("Number of Passengers")



**Stationarity**

Stationarity is a key part of time series analysis. Simply put, stationarity means that the manner in which time series data changes is constant. A stationary time series will not have any trends or seasonal patterns. You should check for stationarity because it not only makes modeling time series easier, but it is an underlying assumption in many time series methods. Specifically, stationarity is assumed for a wide variety of time series forecasting methods including autoregressive moving average (ARMA), ARIMA and Seasonal ARIMA (SARIMA).

We will use the Dickey Fuller test to check for stationarity in our data. This test will generate critical values and a p-value, which will allow us to accept or reject the null hypothesis that there is no stationarity. If we reject the null hypothesis, that means we accept the alternative, which states that there is stationarity.

These values allow us to test the degree to which present values change with past values. If there is no stationarity in the data set, a change in present values will not cause a significant change in past values.

Let’s test for stationarity in our airline passenger data. To start, let’s calculate a seven-month rolling mean:

rolling\_mean = df.rolling(7).mean()  
rolling\_std = df.rolling(7).std()

Next, let’s overlay our time series with the seven-month rolling mean and seven-month rolling standard deviation. First, let’s make a Matplotlib plot of our time series:

plt.plot(df, color="blue",label="Original Passenger Data")

Then the rolling mean:

plt.plot(rolling\_mean, color="red", label="Rolling Mean Passenger Number")

And finally, the rolling standard deviation:

plt.plot(rolling\_std, color="black", label = "Rolling Standard Deviation in Passenger Number")

Let’s then add a title:

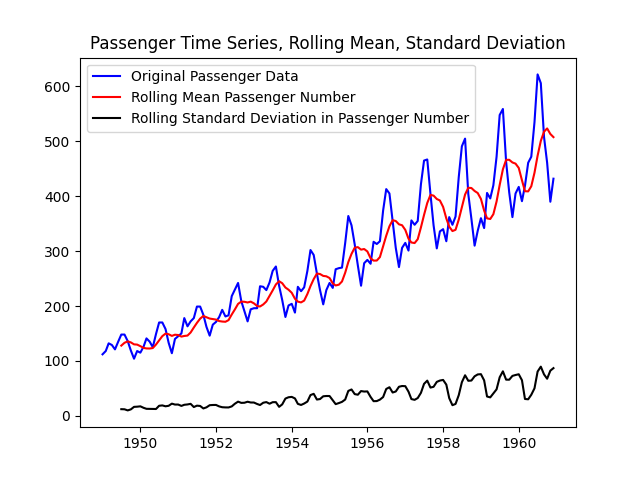
plt.title("Passenger Time Series, Rolling Mean, Standard Deviation")

And a legend:

plt.legend(loc="best")

Then show graph:

plt.show()



Next, let’s import the augmented Dickey-Fuller test from the statsmodels package. The documentation for the test can be found here.

from statsmodels.tsa.stattools import adfuller

Next, let’s pass our data frame into the adfuller method. Here, we specify the autolag parameter as “AIC,” which means that the lag is chosen to minimize the information criterion:

adft = adfuller(df,autolag="AIC")

Next, let’s store our results in a data frame display it:

output\_df = pd.DataFrame({"Values":[adft[0],adft[1],adft[2],adft[3], adft[4]['1%'], adft[4]['5%'], adft[4]['10%']] , "Metric":["Test Statistics","p-value","No. of lags used","Number of observations used", "critical value (1%)", "critical value (5%)", "critical value (10%)"]})  
print(output\_df)

Values Metric

0 0.815369 Test Statistics

1 0.991880 p-value

2 13.000000 No. of lags used

3 130.000000 Number of observations used

4 -3.481682 critical value (1%)

5 -2.884042 critical value (5%)

6 -2.578770 critical value (10%)

We can see that our data is not stationary from the fact that our p-value is greater than 5 percent and the test statistic is greater than the critical value. We can also draw these conclusions from inspecting the data, as we see a clear, increasing trend in the number of passengers.

**Autocorrelation**

Checking time series data for autocorrelation in Python is another important part of the analytic process. This is a measure of how correlated time series data is at a given point in time with past values, which has huge implications across many industries. For example, if our passenger data has strong autocorrelation, we can assume that high passenger numbers today suggest a strong likelihood that they will be high tomorrow as well.

The Pandas data frame has an autocorrelation method that we can use to calculate the autocorrelation in our passenger data. Let’s do this for a one-month lag:

autocorrelation\_lag1 = df['#Passengers'].autocorr(lag=1)  
print("One Month Lag: ", autocorrelation\_lag1)

Now, let’s try three, six and nine months:

autocorrelation\_lag3 = df['#Passengers'].autocorr(lag=3)  
print("Three Month Lag: ", autocorrelation\_lag3)  
  
autocorrelation\_lag6 = df['#Passengers'].autocorr(lag=6)  
print("Six Month Lag: ", autocorrelation\_lag6)  
  
autocorrelation\_lag9 = df['#Passengers'].autocorr(lag=9)  
print("Nine Month Lag: ", autocorrelation\_lag9)

One Month Lag: 0.9601946480498522

Three Month Lag: 0.837394765081794

Six Month Lag: 0.7839187959206183

Nine Month Lag: 0.8278519011167602

We see that, even with a nine-month lag, the data is highly autocorrelated. This is further illustration of the short- and long-term trends in the data.

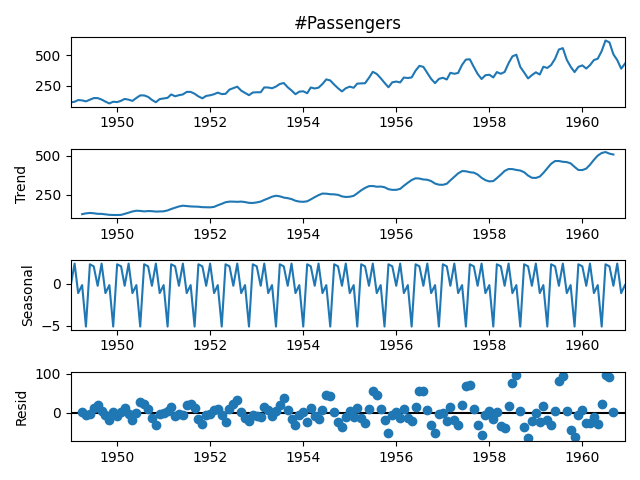
**Decomposition**

Trend decomposition is another useful way to visualize the trends in time series data. To proceed, let’s import seasonal\_decompose from the statsmodels package:

from statsmodels.tsa.seasonal import seasonal\_decompose

Next, let’s pass our data frame into the seasonal\_decompose method and plot the result:

decompose = seasonal\_decompose(df['#Passengers'],model='additive', period=7)  
decompose.plot()  
plt.show()



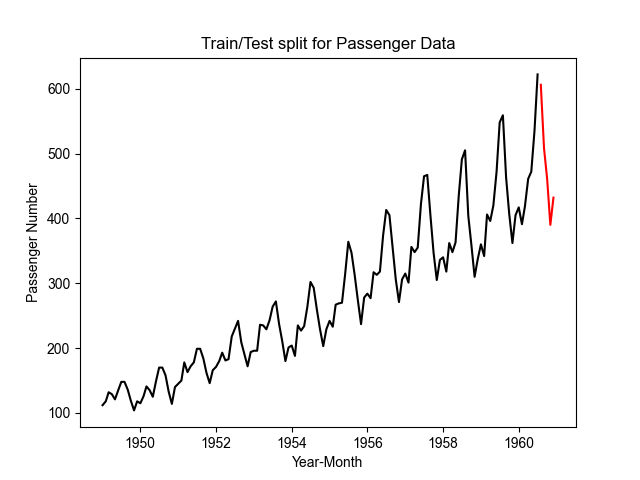
From this plot, we can clearly see the increasing trend in number of passengers and the seasonality patterns in the rise and fall in values each year.

**Forecasting**

Time series forecasting allows us to predict future values in a time series given current and past data. Here, we will use the ARIMA method to forecast the number of passengers, which allows us to forecast future values in terms of a linear combination of past values. We will use the auto\_arima package, which will allow us to forgo the time consuming process of hyperparameter tuning.

First, let’s split our data for training and testing and visualize the split:

df['Date'] = df.index  
train = df[df['Date'] < pd.to\_datetime("1960-08", format='%Y-%m')]  
train['train'] = train['#Passengers']  
del train['Date']  
del train['#Passengers']  
test = df[df['Date'] >= pd.to\_datetime("1960-08", format='%Y-%m')]  
del test['Date']  
test['test'] = test['#Passengers']  
del test['#Passengers']  
plt.plot(train, color = "black")  
plt.plot(test, color = "red")  
plt.title("Train/Test split for Passenger Data")  
plt.ylabel("Passenger Number")  
plt.xlabel('Year-Month')  
sns.set()



The black line corresponds to our training data and the red line corresponds to our test data.

Let’s import auto\_arima from the pdmarima package, train our model and generate predictions:

from pmdarima.arima import auto\_arima  
model = auto\_arima(train, trace=True, error\_action='ignore', suppress\_warnings=True)  
model.fit(train)  
forecast = model.predict(n\_periods=len(test))  
forecast = pd.DataFrame(forecast,index = test.index,columns=['Prediction'])

Below is a truncated sample of the output:

Performing stepwise search to minimize aic

ARIMA(2,1,2)(0,0,0)[0] intercept : AIC=inf, Time=1.25 sec

ARIMA(0,1,0)(0,0,0)[0] intercept : AIC=1352.593, Time=0.03 sec

ARIMA(1,1,0)(0,0,0)[0] intercept : AIC=1340.702, Time=0.09 sec

ARIMA(0,1,1)(0,0,0)[0] intercept : AIC=1336.259, Time=0.15 sec

ARIMA(0,1,0)(0,0,0)[0] : AIC=1352.415, Time=0.04 sec

ARIMA(1,1,1)(0,0,0)[0] intercept : AIC=1329.986, Time=0.24 sec

ARIMA(2,1,1)(0,0,0)[0] intercept : AIC=inf, Time=0.44 sec

ARIMA(1,1,2)(0,0,0)[0] intercept : AIC=inf, Time=0.49 sec

ARIMA(0,1,2)(0,0,0)[0] intercept : AIC=1335.098, Time=0.16 sec

ARIMA(2,1,0)(0,0,0)[0] intercept : AIC=1336.923, Time=0.09 sec

ARIMA(1,1,1)(0,0,0)[0] : AIC=1329.407, Time=0.13 sec

ARIMA(0,1,1)(0,0,0)[0] : AIC=1335.407, Time=0.09 sec

ARIMA(1,1,0)(0,0,0)[0] : AIC=1339.796, Time=0.03 sec

ARIMA(2,1,1)(0,0,0)[0] : AIC=1325.560, Time=0.11 sec

ARIMA(2,1,0)(0,0,0)[0] : AIC=1336.364, Time=0.06 sec

ARIMA(3,1,1)(0,0,0)[0] : AIC=1327.333, Time=0.18 sec

ARIMA(2,1,2)(0,0,0)[0] : AIC=inf, Time=0.53 sec

ARIMA(1,1,2)(0,0,0)[0] : AIC=1329.419, Time=0.10 sec

ARIMA(3,1,0)(0,0,0)[0] : AIC=1337.022, Time=0.08 sec

ARIMA(3,1,2)(0,0,0)[0] : AIC=1319.705, Time=0.26 sec

ARIMA(4,1,2)(0,0,0)[0] : AIC=1317.124, Time=0.48 sec

ARIMA(4,1,1)(0,0,0)[0] : AIC=1324.140, Time=0.28 sec

ARIMA(5,1,2)(0,0,0)[0] : AIC=1319.052, Time=0.39 sec

ARIMA(4,1,3)(0,0,0)[0] : AIC=1315.051, Time=0.64 sec

ARIMA(3,1,3)(0,0,0)[0] : AIC=inf, Time=0.79 sec

ARIMA(5,1,3)(0,0,0)[0] : AIC=1317.044, Time=0.88 sec

ARIMA(4,1,4)(0,0,0)[0] : AIC=inf, Time=0.85 sec

ARIMA(3,1,4)(0,0,0)[0] : AIC=inf, Time=0.83 sec

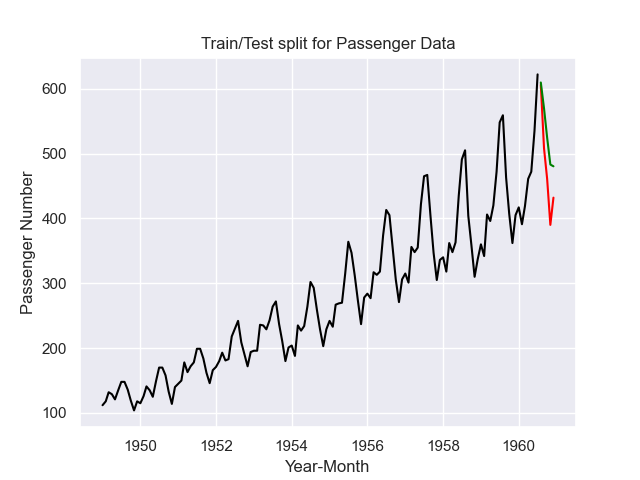
ARIMA(5,1,4)(0,0,0)[0] : AIC=inf, Time=0.76 sec

ARIMA(4,1,3)(0,0,0)[0] intercept : AIC=inf, Time=0.75 sec

Best model: ARIMA(4,1,3)(0,0,0)[0]

Total fit time: 11.252 seconds

Now, let’s display the output of our model:



Our predictions are shown in green and the actual values are shown in orange.

Finally, let’s calculate root mean squared error (RMSE):

from math import sqrt  
from sklearn.metrics import mean\_squared\_error  
rms = sqrt(mean\_squared\_error(test,forecast))  
print("RMSE: ", rms)

RMSE: 61.36447741110699

**Importance of Time Series Analysis in Python**

Conducting time series data analysis is a task that almost every data scientist will face in their career. Having a good understanding of the tools and methods for analysis can enable data scientists to uncover trends, anticipate events and consequently inform decision making. Understanding the seasonality patterns through stationarity, autocorrelation and trend decomposition can guide promotion planning throughout the year, which can improve profits for companies. Finally, time series forecasting is a powerful way to anticipate future events in your time series data, which can also significantly impact decision making. These types of analyses are invaluable to any data scientist or data science team that looks to bring value to their company with time series data.