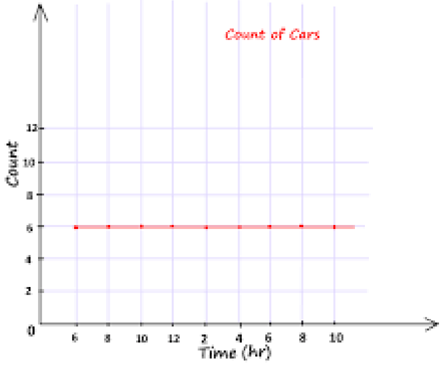
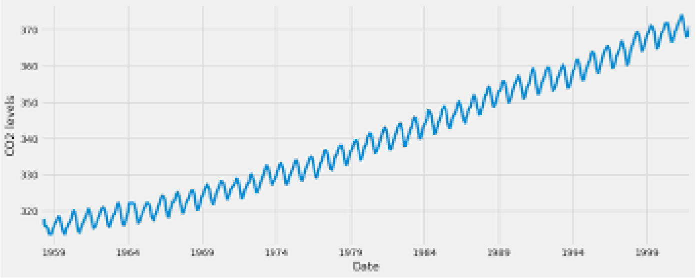
Analytics Vidhya

courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6132962-introduction-to-time-series

Introduction to Time Series

Which of the following do you think is an example of time series? Even if you don’t know, try making a guess.





Time Series is generally data which is collected over time and is dependent on it.

Here we see that the count of cars is independent of time, hence it is not a time series. While the CO2 level increases with respect to time, hence it is a time series.

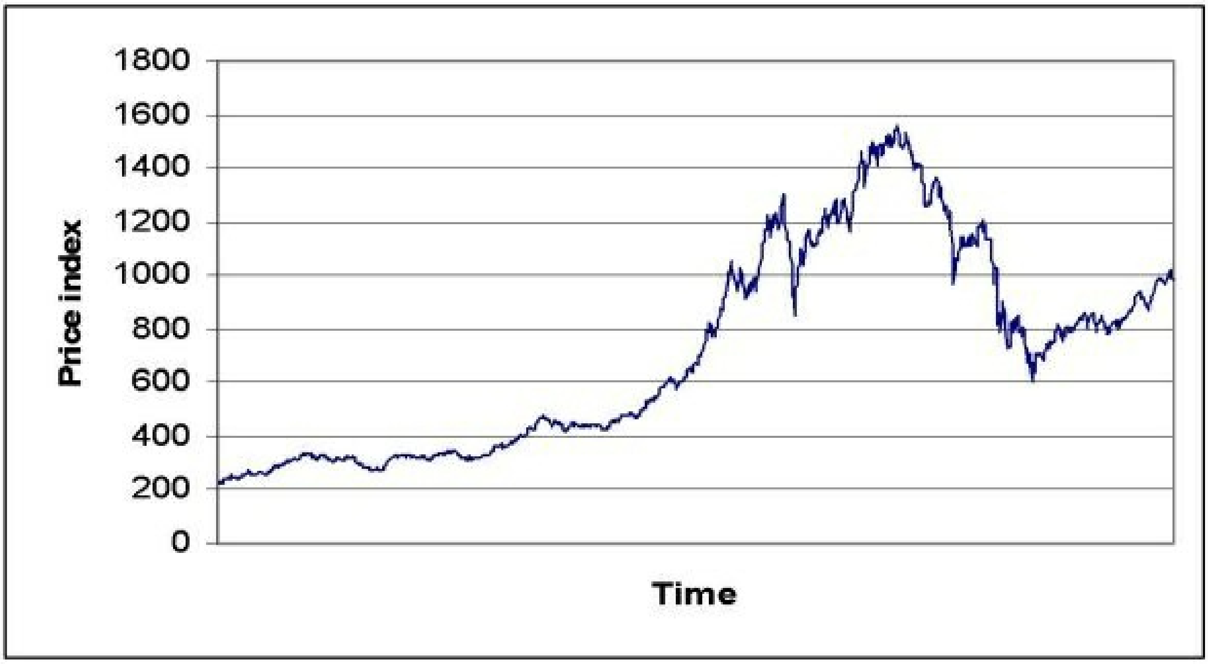
Let us now look at the formal definition of Time Series.

A series of data points collected in time order is known as a time series. Most of business houses work on time series data to analyze sales number for the next year, website traffic, count of traffic, number of calls received, etc. Data of a time series can be used for forecasting.

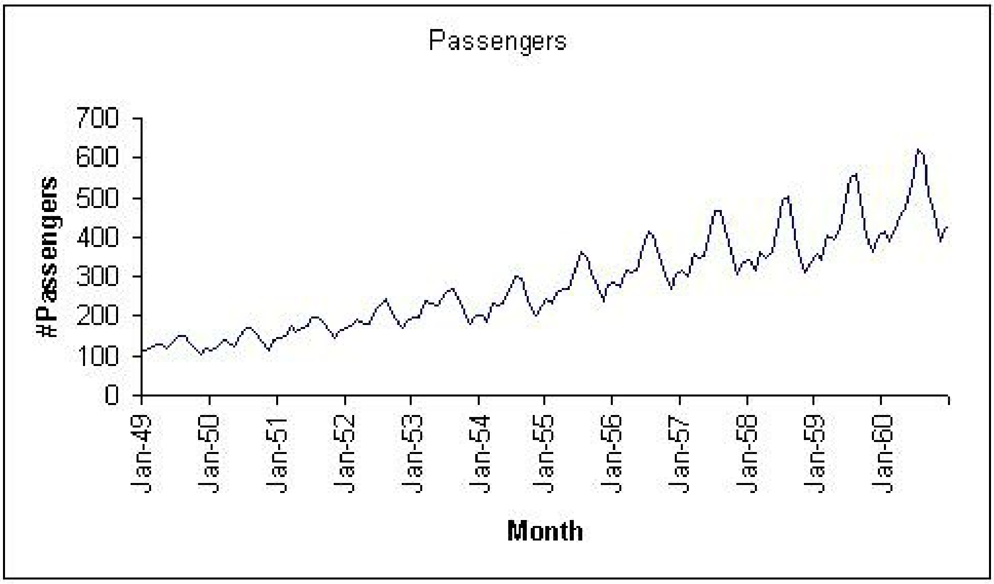
Not every data collected with respect to time represents a time series.

Some of the examples of time series are:

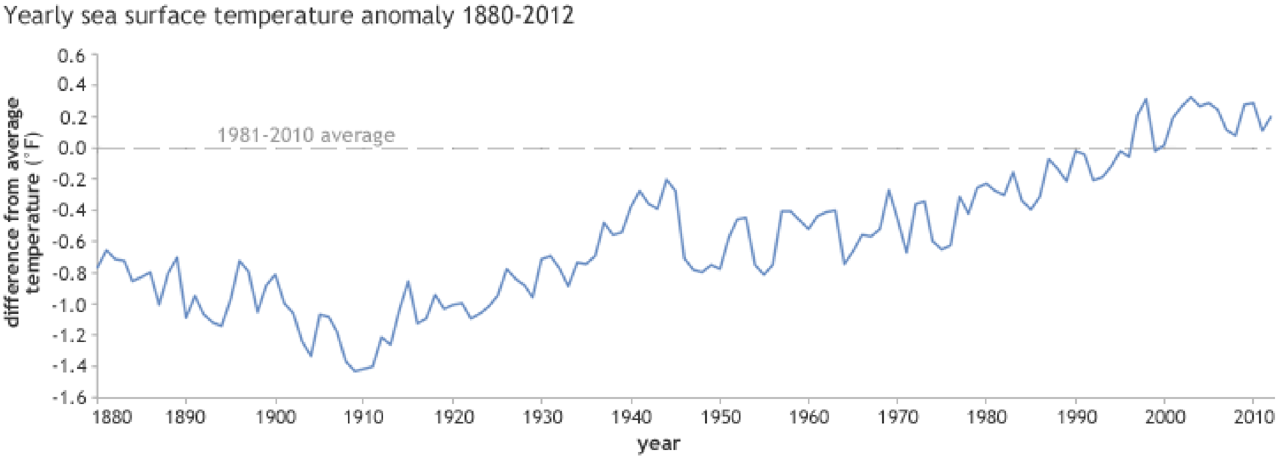
Stock Price :



Passenger Count of an airlines :

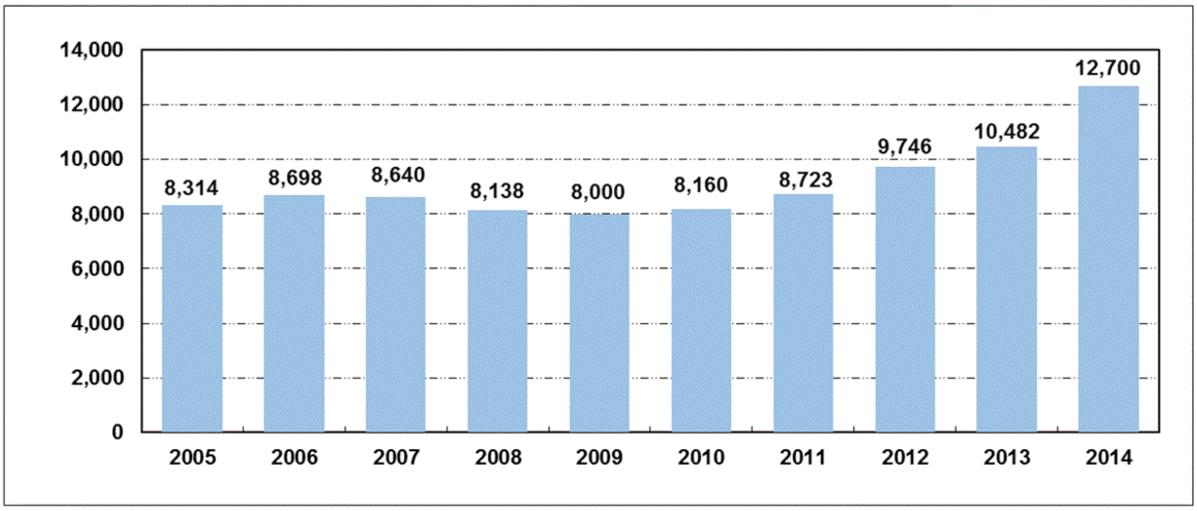


Temperature over time :



Number of visitors in a hotel :

Visitors to Universal Studios Japan **(Unit: million)**



Source: Universal Studios Japan

Now that we can differentiate between a Time Series and a non Time Series data, let us explore Time Series further.

Analytics Vidhya

courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6133008-components-of-a-time-series

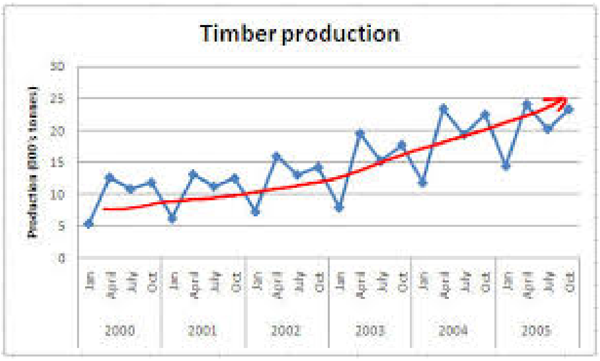
Now as we have an understanding of what a time series is and the difference between a time series and a non time series, let's now look at the components of a time series.

Components of a Time Series

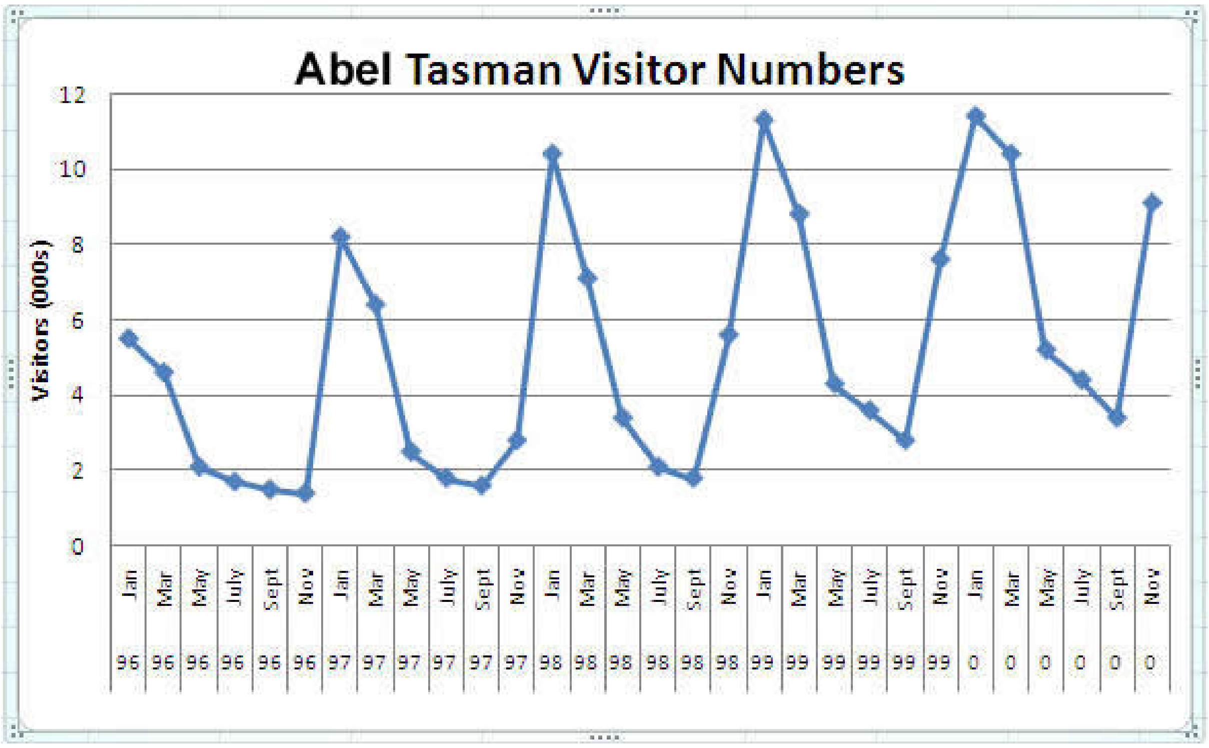
1. **Trend** : Trend is a general direction in which something is developing or changing. So we see an increasing trend in this time series. We can see that the passenger count is increasing with the number of years. Let's visualize the trend of a time series:

Example

Here the red line represents an increasing trend of the time series.



2. **Seasonality** : Another clear pattern can also be seen in the above time series, i.e., the pattern is repeating at regular time interval which is known as the seasonality. Any predictable change or pattern in a time series that recurs or repeats over a specific time period can be said to be seasonality. Let's visualize the seasonality of the time series:



Example

We can see that the time series is repeating its pattern after every 12 months i.e there is a peak every year during the month of january and a trough every year in the month of september, hence this time series has a seasonality of 12 months.

Difference between a time series and regression problem

Here you might think that as the target variable is numerical it can be predicted using regression techniques, but a time series problem is different from a regression problem in following ways:

- The main difference is that a time series is time dependent. So the basic assumption of a linear regression model that the observations are independent doesn’t hold in this case.
- Along with an increasing or decreasing trend, most Time Series have some form of seasonality trends,i.e. variations specific to a particular time frame.

So, predicting a time series using regression techniques is not a good approach.

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.

Analytics Vidhya

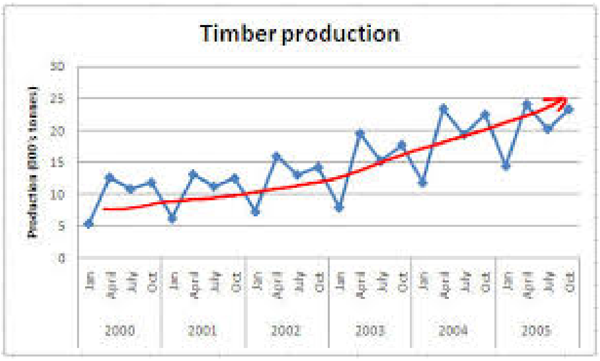
courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6133008-components-of-a-time-series

Now as we have an understanding of what a time series is and the difference between a time series and a non time series, let's now look at the components of a time series.

Components of a Time Series

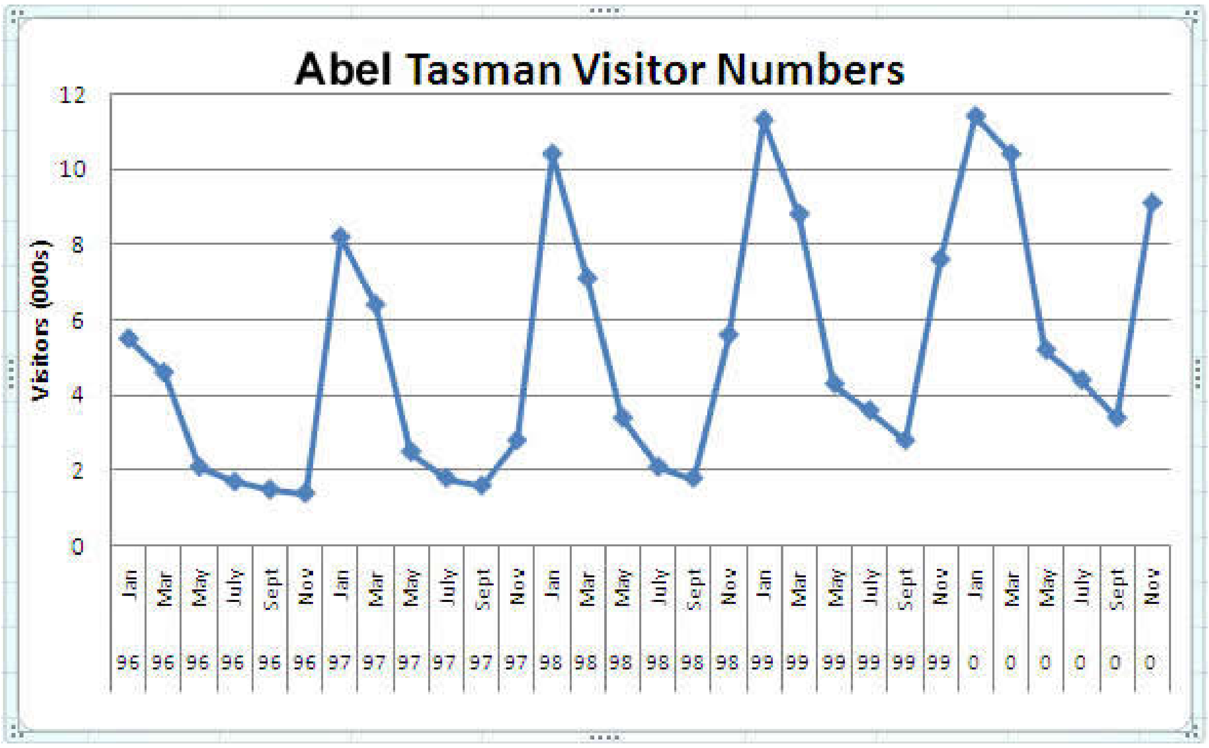
1. **Trend** : Trend is a general direction in which something is developing or changing. So we see an increasing trend in this time series. We can see that the passenger count is increasing with the number of years. Let's visualize the trend of a time series:

Example



Here the red line represents an increasing trend of the time series.

2. **Seasonality** : Another clear pattern can also be seen in the above time series, i.e., the pattern is repeating at regular time interval which is known as the seasonality. Any predictable change or pattern in a time series that recurs or repeats over a specific time period can be said to be seasonality. Let's visualize the seasonality of the time series:



Example

We can see that the time series is repeating its pattern after every 12 months i.e there is a peak every year during the month of january and a trough every year in the month of september, hence this time series has a seasonality of 12 months.

Difference between a time series and regression problem

Here you might think that as the target variable is numerical it can be predicted using regression techniques, but a time series problem is different from a regression problem in following ways:

- The main difference is that a time series is time dependent. So the basic assumption of a linear regression model that the observations are independent doesn’t hold in this case.
- Along with an increasing or decreasing trend, most Time Series have some form of seasonality trends,i.e. variations specific to a particular time frame.

So, predicting a time series using regression techniques is not a good approach.

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.

Analytics Vidhya

Now as we have understood various components of a time series, let’s look at the problem statement which we will be solving in this course.

Problem Statement

Unicorn Investors wants to make an investment in a new form of transportation - JetRail. JetRail uses Jet propulsion technology to run rails and move people at a high speed! The investment would only make sense, if they can get more than 1 Million monthly users with in next 18 months. In order to help Unicorn Ventures in their decision, you need to forecast the traffic on JetRail for the next 7 months. You are provided with traffic data of JetRail since inception in the test file.

You can get the dataset here https://datahack.analyticsvidhya.com/contest/practice-problem-time-series-2/

It is advised to look at the dataset after completing the hypothesis generation part.

Analytics Vidhya

courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6133010-table-of-contents

Now that we have the dataset as well as the understanding of the problem statement, let’s look at the steps that we will follow in this course to solve the problem at hand.

Table of Contents

Analytics Vidhya

We will start with the first step, i.e. Hypothesis Generation. Hypothesis Generation is the process of listing out all the possible factors that can affect the outcome.

Hypothesis generation is done before having a look at the data in order to avoid any bias that may result after the observation.

1) Hypothesis Generation

Hypothesis generation helps us to point out the factors which might affect our dependent variable. Below are some of the hypotheses which I think can affect the passenger count(dependent variable for this time series problem) on the JetRail:

1. There will be an increase in the traffic as the years pass by.

   Explanation - Population has a general upward trend with time, so I can expect more people to travel by JetRail. Also, generally companies expand their businesses over time leading to more customers travelling through JetRail.

2. The traffic will be high from May to October.

   Explanation - Tourist visits generally increases during this time perion.

3. Traffic on weekdays will be more as compared to weekends/holidays.

   Explanation - People will go to office on weekdays and hence the traffic will be more

4. Traffic during the peak hours will be high.

   Explanation - People will travel to work, college.

We will try to validate each of these hypothesis based on the dataset. Now let's have a look at the dataset.

Analytics Vidhya

courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6133032-getting-the-system-ready-and-loading-data

After making our hypothesis, we will try to validate them. Before that we will import all the necessary packages.

2) Getting the system ready and loading the data

Versions:

- Python = 3.7
- Pandas = 0.20.3
- sklearn = 0.19.1

Now we will import all the packages which will be used throughout the notebook.

```
import pandas as pd
import numpy as np          # For mathematical calculations
import matplotlib.pyplot as plt  # For plotting graphs
from datetime import datetime    # To access datetime
from pandas import Series        # To work on series
%matplotlib inline
import warnings                   # To ignore the warnings warnings.filterwarnings("ignore")
```

Now let’s read the train and test data

```
train=pd.read_csv("Train_SU63ISt.csv")
test=pd.read_csv("Test_0qrQsBZ.csv")
```

Let’s make a copy of train and test data so that even if we do changes in these dataset we do not lose the original dataset.

```
train_original=train.copy()
test_original=test.copy()
```

Analytics Vidhya

courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6133080-dataset-structure-and-content

After loading the data, let’s have a quick look at the dataset to know our data better.

3) Dataset Structure and Content

Let’s dive deeper and have a look at the dataset. First of all let’s have a look at the features in the train and test dataset.

```
train.columns, test.columns
```

```
(Index(['ID', 'Datetime', 'Count'], dtype='object'),
 Index(['ID', 'Datetime'], dtype='object'))
```

We have ID, Datetime and corresponding count of passengers in the train file. For test file we have ID and Datetime only so we have to predict the Count for test file.

Let’s understand each feature first:

- ID is the unique number given to each observation point.
- Datetime is the time of each observation.
- Count is the passenger count corresponding to each Datetime.

Let’s look at the data types of each feature.

```
train.dtypes, test.dtypes
```

```
(ID           int64
 Datetime    object
 Count        int64
 dtype: object, ID           int64
 Datetime    object
 dtype: object)
```

- ID and Count are in integer format while the Datetime is in object format for the train file.
- Id is in integer and Datetime is in object format for test file.

Now we will see the shape of the dataset.

```
train.shape, test.shape
```

```
((18288, 3), (5112, 2))
```

We have 18288 different records for the Count of passengers in train set and 5112 in test set.

Analytics Vidhya

courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6133138-feature-extraction

Now we will extract more features to validate our hypothesis.

4) Feature Extraction

We will extract the time and date from the Datetime. We have seen earlier that the data type of Datetime is object. So first of all we have to change the data type to datetime format otherwise we can not extract features from it.

```
train['Datetime'] = pd.to_datetime(train.Datetime,format='%d-%m-%Y %H:%M')
test['Datetime'] = pd.to_datetime(test.Datetime,format='%d-%m-%Y %H:%M')
test_original['Datetime'] = pd.to_datetime(test_original.Datetime,format='%d-%m-%Y %H:%M')
train_original['Datetime'] = pd.to_datetime(train_original.Datetime,format='%d-%m-%Y %H:%M')
```

We made some hypothesis for the effect of hour, day, month and year on the passenger count. So, let's extract the year, month, day and hour from the Datetime to validate our hypothesis.

```
for i in (train, test, test_original, train_original):
    i['year']=i.Datetime.dt.year
    i['month']=i.Datetime.dt.month
    i['day']=i.Datetime.dt.day
    i['Hour']=i.Datetime.dt.hour
```

We made a hypothesis for the traffic pattern on weekday and weekend as well. So, let's make a weekend variable to visualize the impact of weekend on traffic.

- We will first extract the day of week from Datetime and then based on the values we will assign whether the day is a weekend or not.
- Values of 5 and 6 represents that the days are weekend.

```
train['day of week']=train['Datetime'].dt.dayofweek
temp = train['Datetime']
```

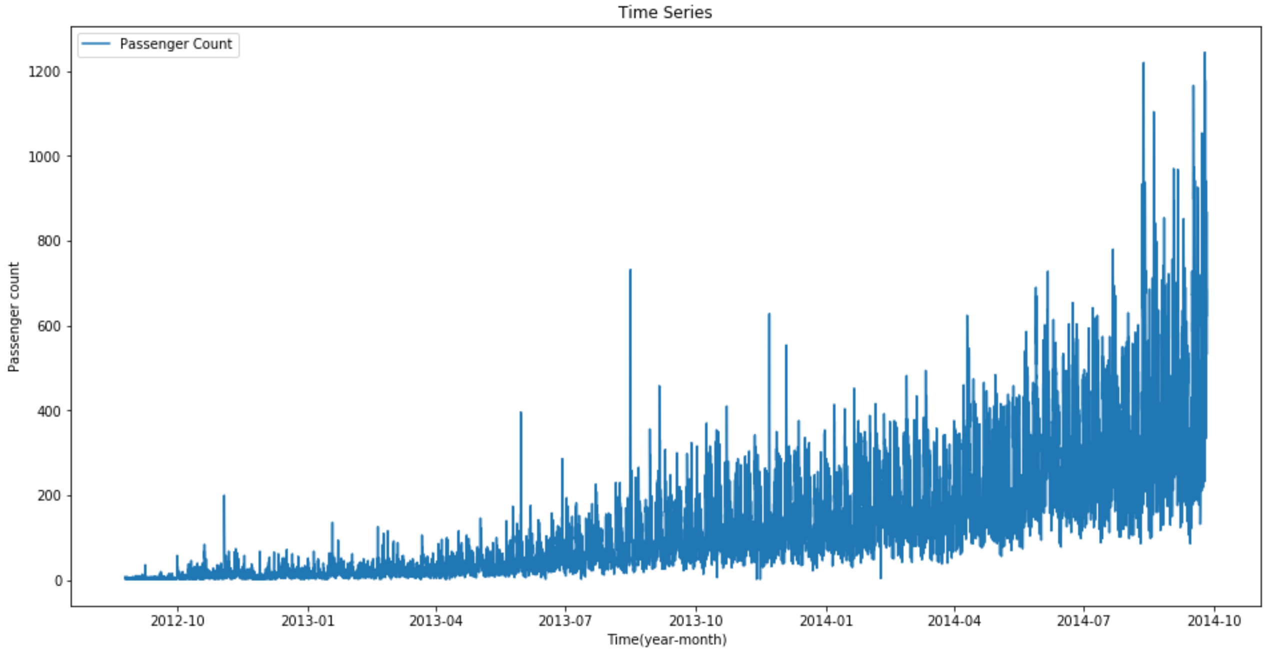
Let's assign 1 if the day of week is a weekend and 0 if the day of week in not a weekend.

```
def applyer(row):
    if row.dayofweek == 5 or row.dayofweek == 6:
        return 1
    else:
        return 0
temp2 = train['Datetime'].apply(applyer)
train['weekend']=temp2
```

Let's look at the time series.

```
train.index = train['Datetime'] # indexing the Datetime to get the time period on the x-axis.
df=train.drop('ID',1)           # drop ID variable to get only the Datetime on x-axis.
ts = df['Count']
plt.figure(figsize=(16,8))
plt.plot(ts, label='Passenger Count')
plt.title('Time Series')
plt.xlabel("Time(year-month)")
plt.ylabel("Passenger count")
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7f2ad5ce62b0>
```



Here we can infer that there is an increasing trend in the series, i.e., the number of count is increasing with respect to time. We can also see that at certain points there is a sudden increase in the number of counts. The possible reason behind this could be that on particular day, due to some event the traffic was high.

We will work on the train file for all the analysis and will use the test file for forecasting.

Lets recall the hypothesis that we made earlier:

- Traffic will increase as the years pass by
- Traffic will be high from May to October
- Traffic on weekdays will be more
- Traffic during the peak hours will be high

After having a look at the dataset, we will now try to validate our hypothesis and make other inferences from the dataset.

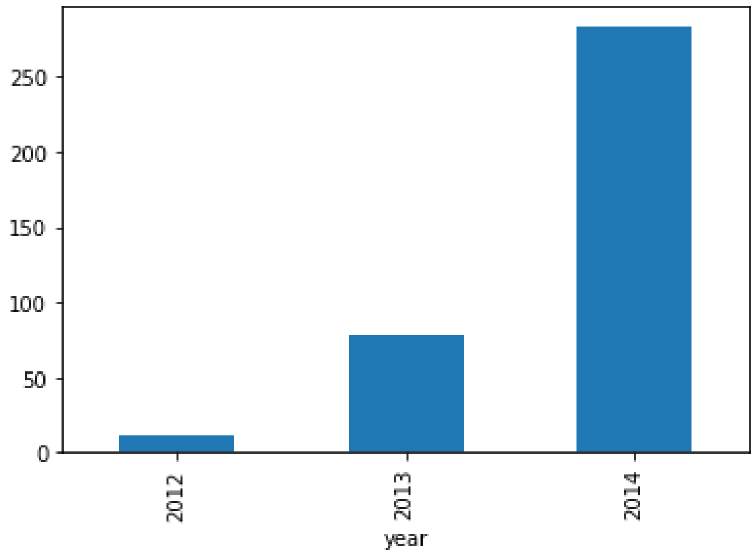
5) Exploratory Analysis

Let us try to verify our hypothesis using the actual data.

Our first hypothesis was traffic will increase as the years pass by. So let’s look at yearly passenger count.

```
train.groupby('year')['Count'].mean().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2cfb0424a8>
```

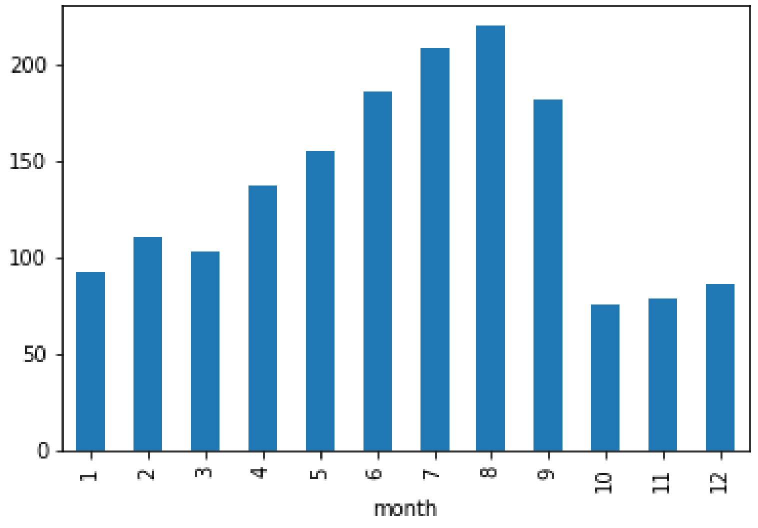


We see an exponential growth in the traffic with respect to year which validates our hypothesis.

Our second hypothesis was about increase in traffic from May to October. So, let’s see the relation between count and month.

```
train.groupby('month')['Count'].mean().plot.bar()
```

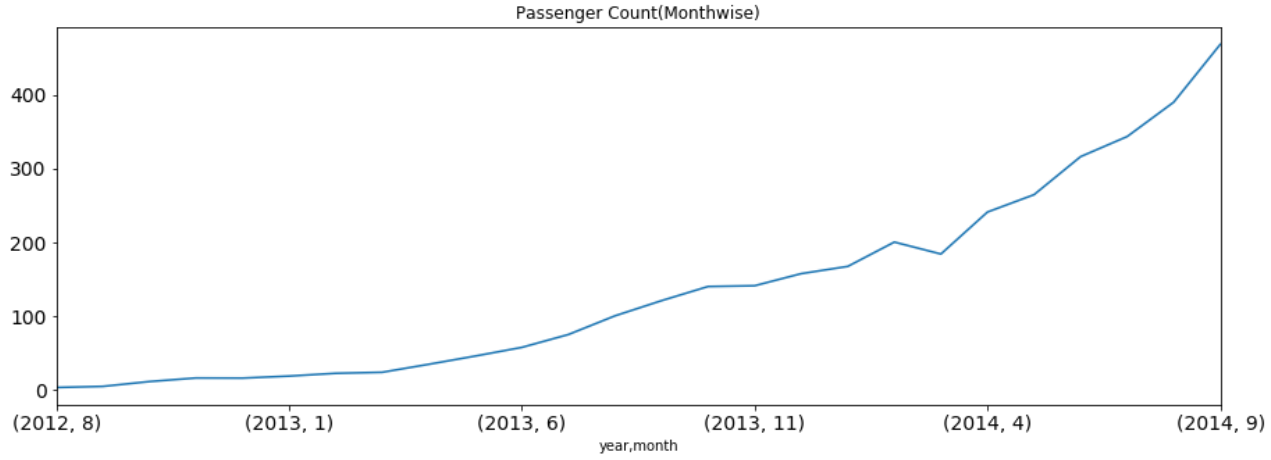
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2cfb82a7f0>
```



Here we see a decrease in the mean of passenger count in last three months. This does not look right. Let’s look at the monthly mean of each year separately.

```
temp=train.groupby(['year', 'month'])['Count'].mean()
temp.plot(figsize=(15,5), title= 'Passenger Count(Monthwise)', fontsize=14)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2cfb878eb8>
```



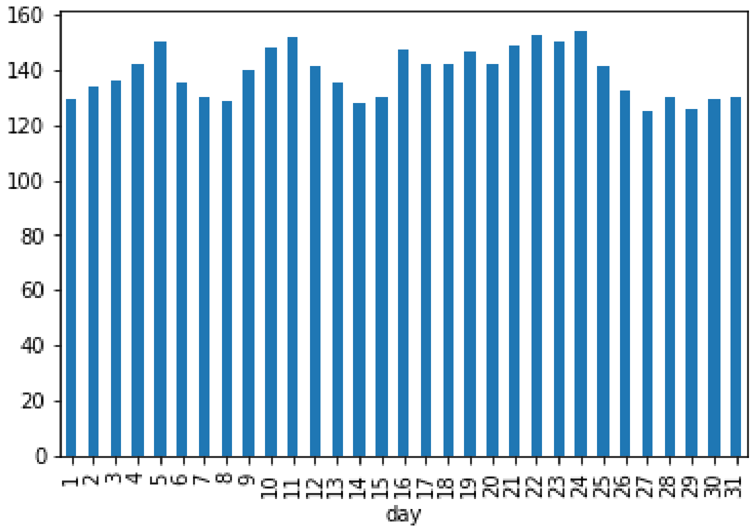
- We see that the months 10, 11 and 12 are not present for the year 2014 and the mean value for these months in year 2012 is very less.
- Since there is an increasing trend in our time series, the mean value for rest of the months will be more because of their larger passenger counts in year 2014 and we will get smaller value for these 3 months.

- In the above line plot we can see an increasing trend in monthly passenger count and the growth is approximately exponential.

Let’s look at the daily mean of passenger count.

```
train.groupby('day')['Count'].mean().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2cfb74dd68>
```

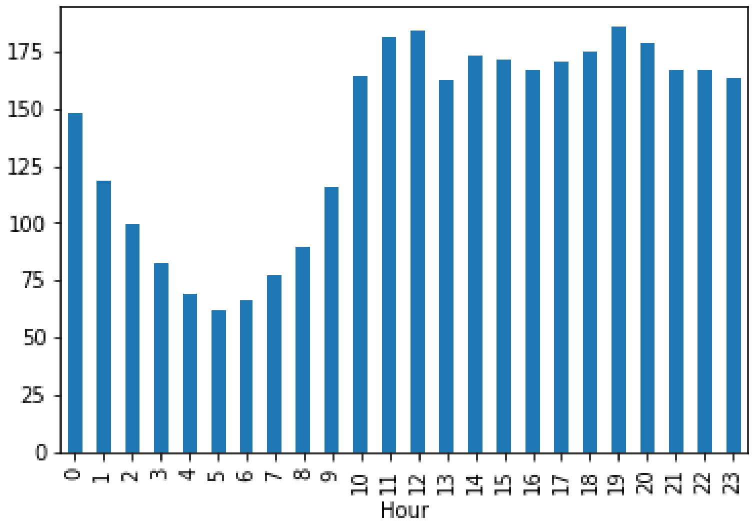


We are not getting much insights from day wise count of the passengers.

We also made a hypothesis that the traffic will be more during peak hours. So let’s see the mean of hourly passenger count.

```
train.groupby('Hour')['Count'].mean().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2cfb59d5c0>
```



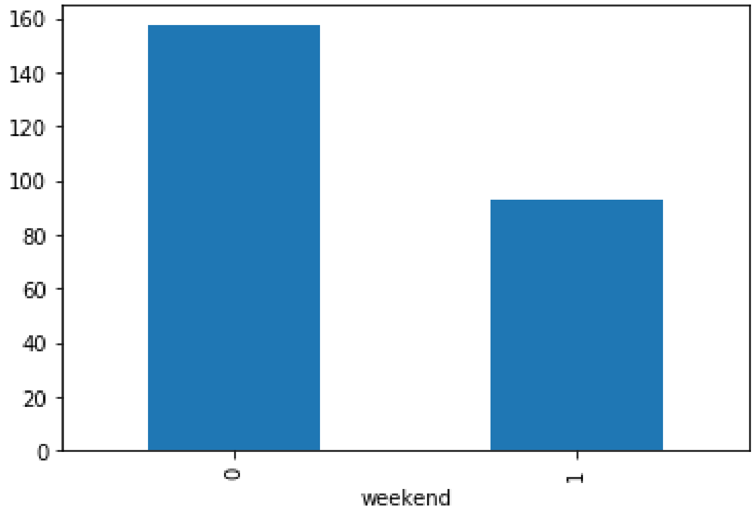
- It can be inferred that the peak traffic is at 7 PM and then we see a decreasing trend till 5 AM.
- After that the passenger count starts increasing again and peaks again between

11AM and 12 Noon.

Let’s try to validate our hypothesis in which we assumed that the traffic will be more on weekdays.

```
train.groupby('weekend')['Count'].mean().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2cfb4d7ac8>
```



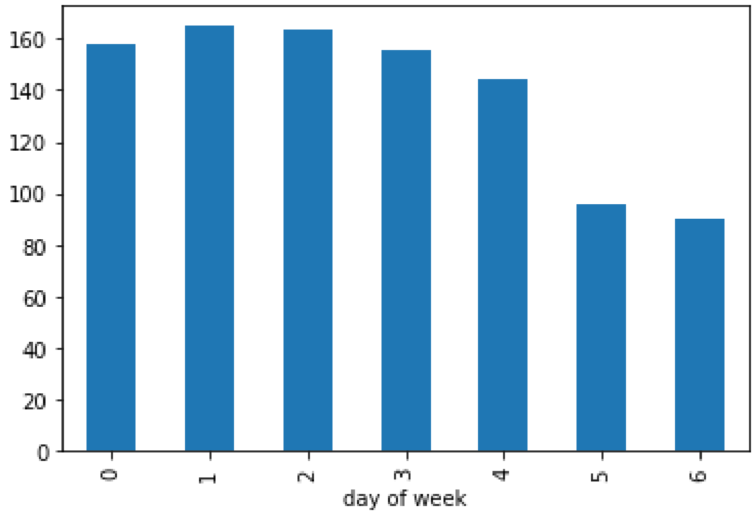
It can be inferred from the above plot that the traffic is more on weekdays as compared to weekends which validates our hypothesis.

Now we will try to look at the day wise passenger count.

Note - 0 is the starting of the week, i.e., 0 is Monday and 6 is Sunday.

```
train.groupby('day of week')['Count'].mean().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2cfb5ac2b0>
```



From the above bar plot, we can infer that the passenger count is less for saturday and sunday as compared to the other days of the week. Now we will look at basic modeling techniques. Before that we will drop the ID variable as it has nothing to do with the passenger count.

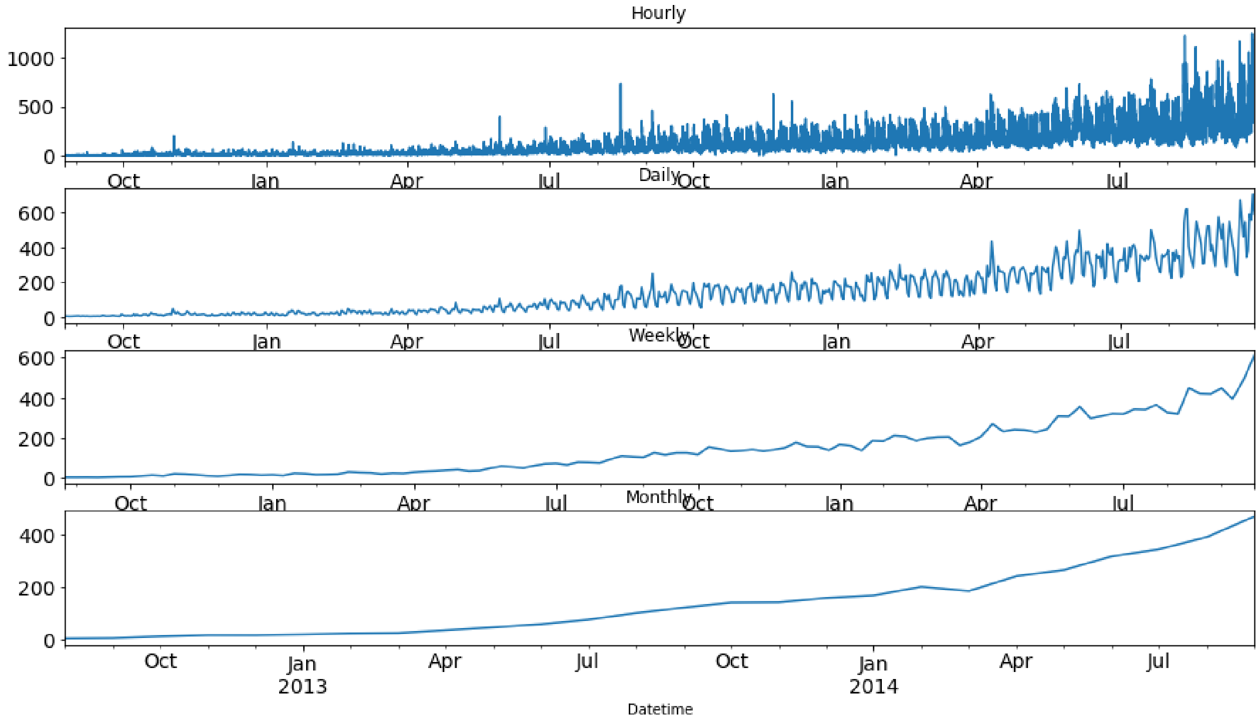
```
train=train.drop('ID',1)
```

As we have seen that there is a lot of noise in the hourly time series, we will aggregate the hourly time series to daily, weekly, and monthly time series to reduce the noise and make it more stable and hence would be easier for a model to learn.

```
train.Timestamp = pd.to_datetime(train.Datetime,format='%d-%m-%Y %H:%M')
train.index = train.Timestamp
# Hourly time series
hourly = train.resample('H').mean()
# Converting to daily mean
daily = train.resample('D').mean()
# Converting to weekly mean
weekly = train.resample('W').mean()
# Converting to monthly mean
monthly = train.resample('M').mean()
```

Let’s look at the hourly, daily, weekly and monthly time series.

```
fig, axs = plt.subplots(4,1)
hourly.Count.plot(figsize=(15,8), title= 'Hourly', fontsize=14, ax=axs[0]) daily.Count.plot(figsize=
(15,8), title= 'Daily', fontsize=14, ax=axs[1]) weekly.Count.plot(figsize=(15,8), title= 'Weekly',
fontsize=14, ax=axs[2]) monthly.Count.plot(figsize=(15,8), title= 'Monthly', fontsize=14, ax=axs[3])

plt.show()
```



We can see that the time series is becoming more and more stable when we are aggregating it on daily, weekly and monthly basis.

But it would be difficult to convert the monthly and weekly predictions to hourly predictions, as first we have to convert the monthly predictions to weekly, weekly to daily and daily to hourly predictions, which will become very expanded process. So, we will work on the daily time series.

```
test.Timestamp = pd.to_datetime(test.Datetime,format='%d-%m-%Y %H:%M')
test.index = test.Timestamp

# Converting to daily mean
test = test.resample('D').mean()

train.Timestamp = pd.to_datetime(train.Datetime,format='%d-%m-%Y %H:%M') train.index =
train.Timestamp
# Converting to daily mean
train = train.resample('D').mean()
```

Analytics Vidhya

courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6134173-splitting-the-data-into-training-and-validation-part

As we have validated all our hypothesis, let’s go ahead and build models for Time Series Forecasting. But before we do that, we will need a dataset(validation) to check the performance and generalisation ability of our model. Below are some of the properties of the dataset required for the purpose.

- The dataset should have the true values of the dependent variable against which the predictions can be checked. Therefore, test dataset cannot be used for the purpose.

- The model should not be trained on the validation dataset. Hence, we cannot train the model on the train dataset and validate on it as well.

So, for the above two reasons, we generally divide the train dataset into two parts. One part is used to train the model and the other part is used as the validation dataset. Now there are multiple ways to divide the train dataset such as Random Division etc. You can look for all of the different validation methods here: https://www.analyticsvidhya.com/blog/2015/11/improve-model-performance-cross-validation-in-python-r/.

For this course, We will be using a time based split explained below.

1) Splitting the data into training and validation part

Now we will divide our data in train and validation. We will make a model on the train part and predict on the validation part to check the accuracy of our predictions.

NOTE - It is always a good practice to create a validation set that can be used to assess our models locally. If the validation metric(rmse) is changing in proportion to public leaderboard score, this would imply that we have chosen a stable validation technique.

To divide the data into training and validation set, we will take last 3 months as the validation data and rest for training data. We will take only 3 months as the trend will be the most in them. If we take more than 3 months for the validation set, our training set will have less data points as the total duration is of 25 months. So, it will be a good choice to take 3 months for validation set.

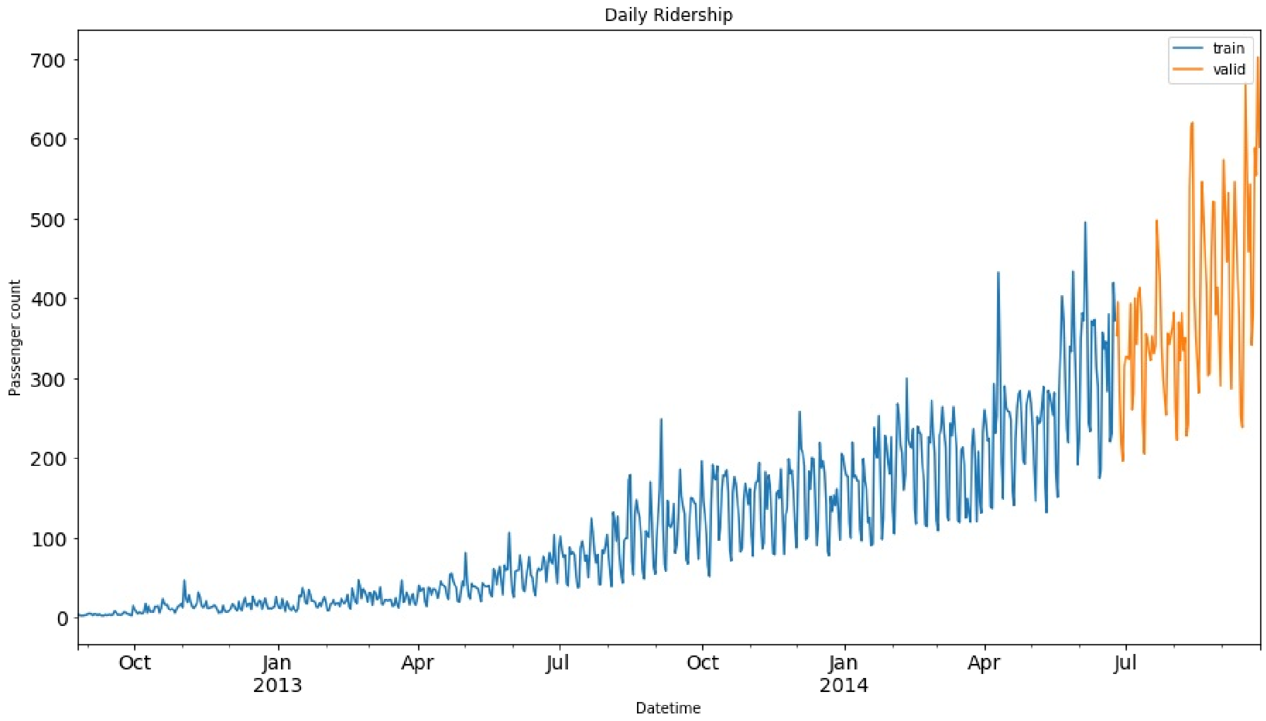
The starting date of the dataset is 25-08-2012 as we have seen in the exploration part and the end date is 25-09-2014.

```
Train=train.ix['2012-08-25':'2014-06-24'] valid=train.ix['2014-06-25':'2014-09-25']
```

- We have done time based validation here by selecting the last 3 months for the validation data and rest in the train data. If we would have done it randomly it may work well for the train dataset but will not work effectively on validation dataset.
- Lets understand it in this way: If we choose the split randomly it will take some values from the starting and some from the last years as well. It is similar to predicting the old values based on the future values which is not the case in real scenario. So, this kind of split is used while working with time related problems.

Now we will look at how the train and validation part has been divided.

```
Train.Count.plot(figsize=(15,8), title= 'Daily Ridership', fontsize=14, label='train')
valid.Count.plot(figsize=(15,8), title= 'Daily Ridership', fontsize=14, label='valid')
plt.xlabel("Datetime") plt.ylabel("Passenger count") plt.legend(loc='best') plt.show()
```



Here the blue part represents the train data and the orange part represents the validation data.

We will predict the traffic for the validation part and then visualize how accurate our predictions are. Finally we will make predictions for the test dataset.

Analytics Vidhya

courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6134208-modeling-techniques

We will look at various models now to forecast the time series . Methods which we will be discussing for the forecasting are:

i) Naive Approach

ii) Moving Average

iii) Simple Exponential Smoothing

iv) Holt’s Linear Trend Model

We will discuss each of these methods in detail now.

i) Naive Approach

> In this forecasting technique, we assume that the next expected point is equal to the last observed point. So we can expect a straight horizontal line as the prediction. Lets understand it with an example and an image:

Suppose we have passenger count for 5 days as shown below:

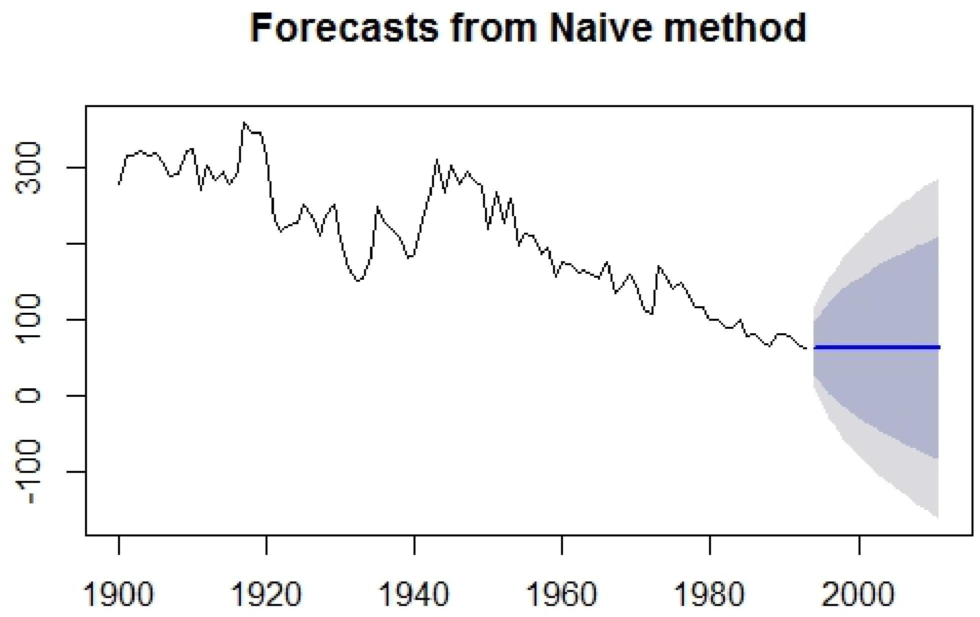
1	10
2	12
3	14
4	13
5	15

And we have to predict the passenger count for next 2 days. Naive approach will assign the 5th day’s passenger count to the 6th and 7th day, i.e., 15 will be assigned to the 6th and 7th day.

1	10
2	12
3	14
4	13

5	15
6	15
7	15

Now lets understand it with an example:

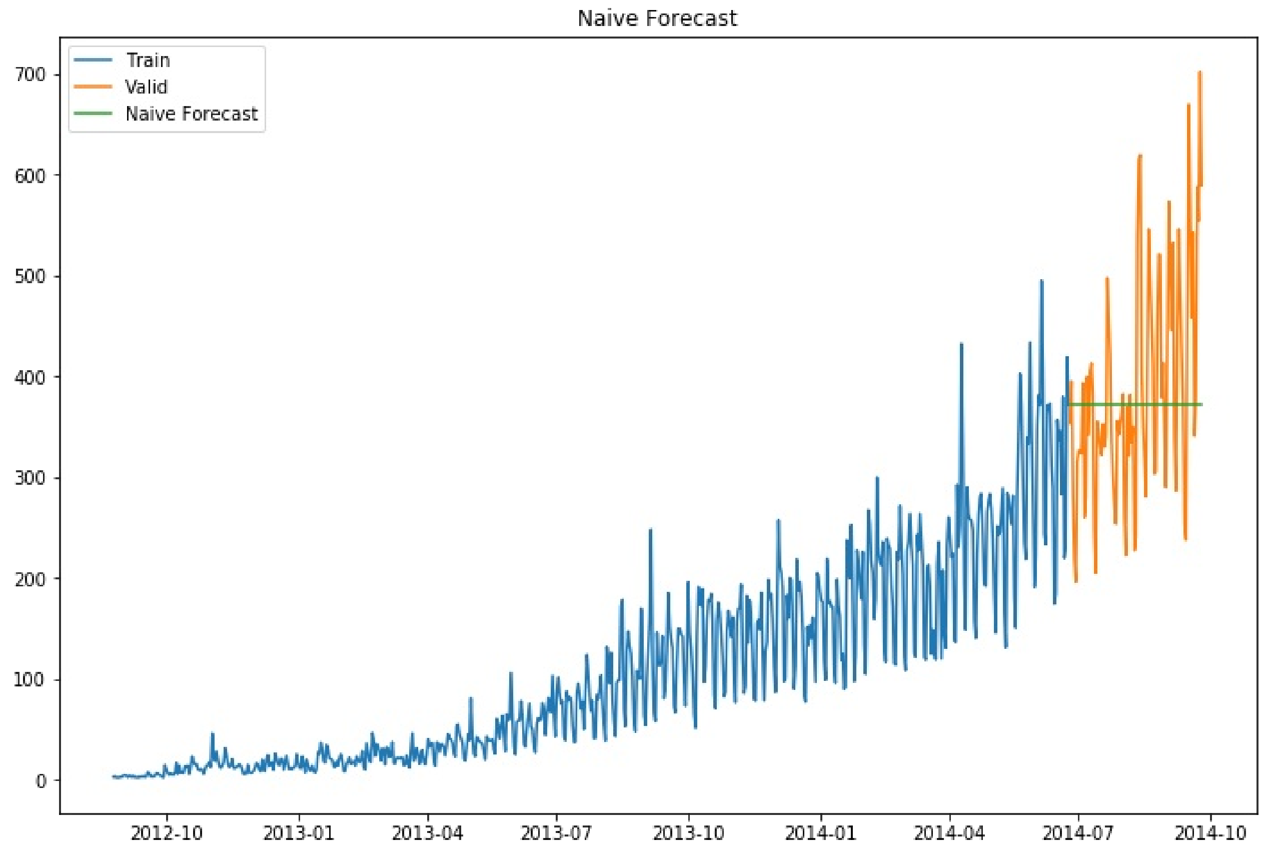


Example

The blue line is the prediction here. All the predictions are equal to the last observed point.

Let’s make predictions using naive approach for the validation set.

```
dd= np.asarray(Train.Count)
y_hat = valid.copy()
y_hat['naive'] = dd[len(dd)-1]
plt.figure(figsize=(12,8))
plt.plot(Train.index, Train['Count'], label='Train')
plt.plot(valid.index,valid['Count'], label='Valid')
plt.plot(y_hat.index,y_hat['naive'], label='Naive Forecast')
plt.legend(loc='best')
plt.title("Naive Forecast")
plt.show()
```



- We can calculate how accurate our predictions are using rmse(Root Mean Square Error).
- rmse is the standard deviation of the residuals.
- Residuals are a measure of how far from the regression line data points are.
- The formula for rmse is:

rmse=sqrt∑i=1N1N(p−a)2

We will now calculate RMSE to check the accuracy of our model on validation data set.

```
from sklearn.metrics import mean_squared_error
from math import sqrt
rms = sqrt(mean_squared_error(valid.Count, y_hat.naive))
```

```
111.79050467496724
```

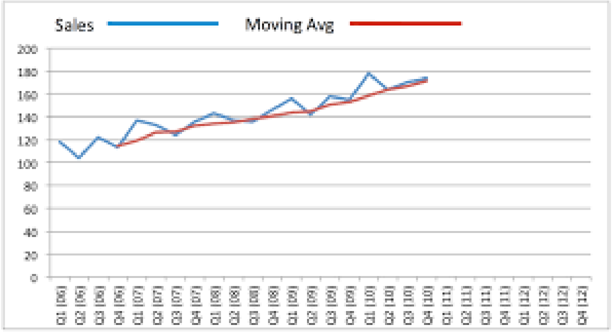
We can infer that this method is not suitable for datasets with high variability. We can reduce the rmse value by adopting different techniques.

ii) Moving Average

In this technique we will take the average of the passenger counts for last few time periods only.

Let’s take an example to understand it:

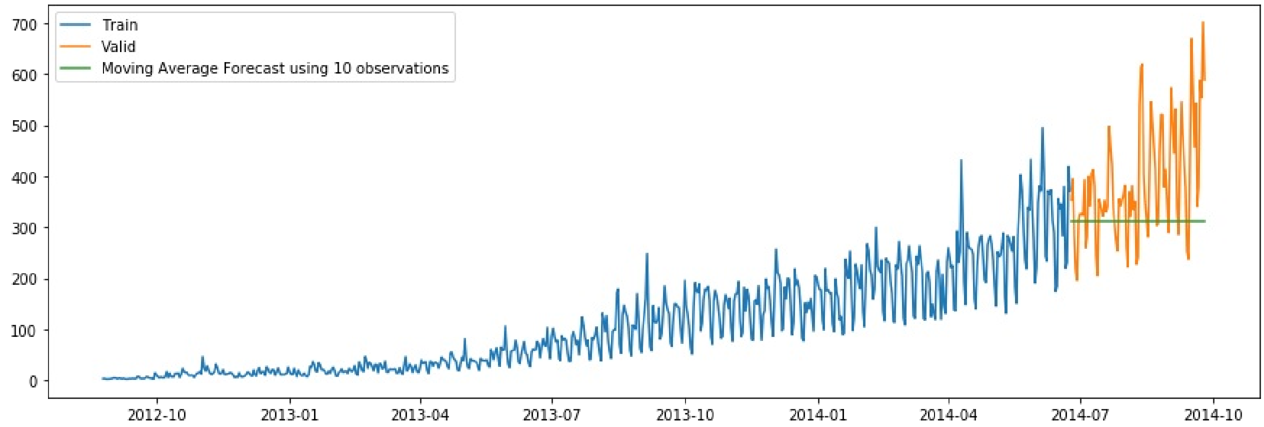
Example

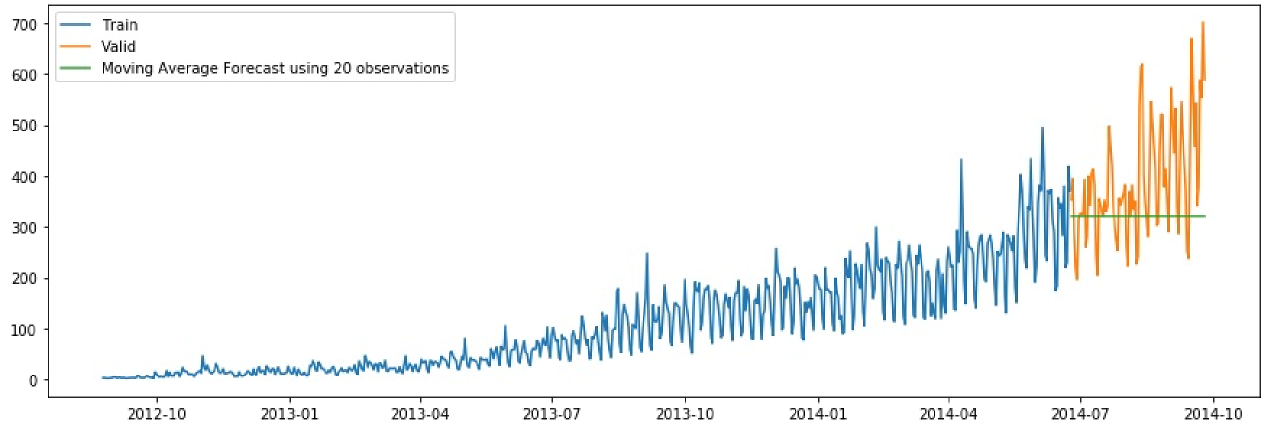


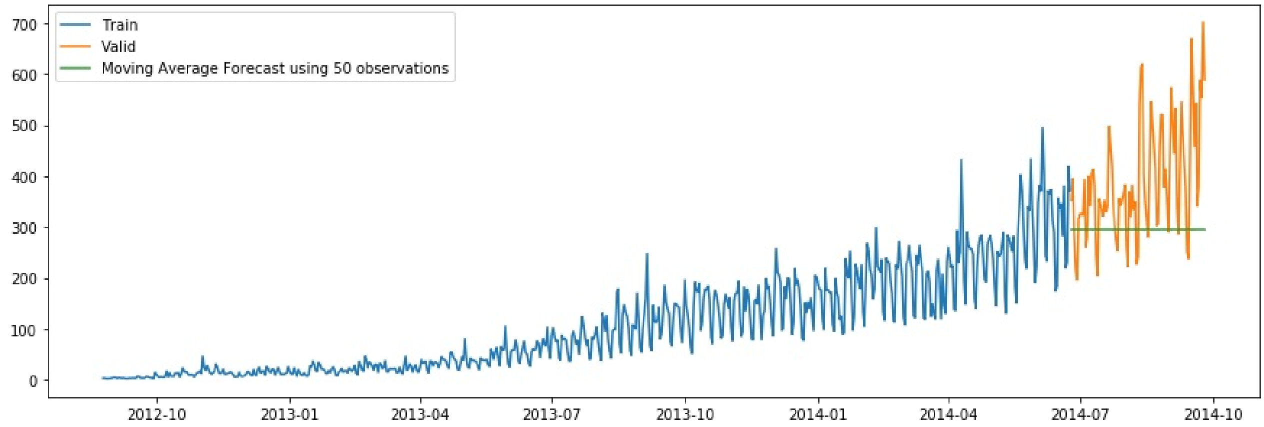
Here the predictions are made on the basis of the average of last few points instead of taking all the previously known values.

Lets try the rolling mean for last 10, 20, 50 days and visualize the results.

```
y_hat_avg = valid.copy()
y_hat_avg['moving_avg_forecast'] = Train['Count'].rolling(10).mean().iloc[-1] # average of last 10 observations.
plt.figure(figsize=(15,5))
plt.plot(Train['Count'], label='Train')
plt.plot(valid['Count'], label='Valid')
plt.plot(y_hat_avg['moving_avg_forecast'], label='Moving Average Forecast using 10 observations')
plt.legend(loc='best')
plt.show()
y_hat_avg = valid.copy()
y_hat_avg['moving_avg_forecast'] = Train['Count'].rolling(20).mean().iloc[-1] # average of last 20 observations.
plt.figure(figsize=(15,5))
plt.plot(Train['Count'], label='Train')
plt.plot(valid['Count'], label='Valid')
plt.plot(y_hat_avg['moving_avg_forecast'], label='Moving Average Forecast using 20 observations')
plt.legend(loc='best')
plt.show()
y_hat_avg = valid.copy()
y_hat_avg['moving_avg_forecast'] = Train['Count'].rolling(50).mean().iloc[-1] # average of last 50 observations.
plt.figure(figsize=(15,5))
plt.plot(Train['Count'], label='Train')
plt.plot(valid['Count'], label='Valid')
plt.plot(y_hat_avg['moving_avg_forecast'], label='Moving Average Forecast using 50 observations')
plt.legend(loc='best')
plt.show()
```







We took the average of last 10, 20 and 50 observations and predicted based on that. This value can be changed in the above code in .rolling().mean() part. We can see that the predictions are getting weaker as we increase the number of observations.

```
rms = sqrt(mean_squared_error(valid.Count, y_hat_avg.moving_avg_forecast))
```

```
144.19175679986802
```

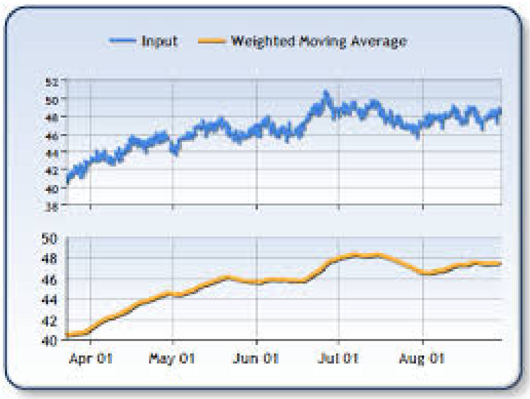
iii) Simple Exponential Smoothing

- In this technique, we assign larger weights to more recent observations than to observations from the distant past.
- The weights decrease exponentially as observations come from further in the past, the smallest weights are associated with the oldest observations.

NOTE - If we give the entire weight to the last observed value only, this method will be similar to the naive approach. So, we can say that naive approach is also a simple exponential smoothing technique where the entire weight is given to the last observed value.

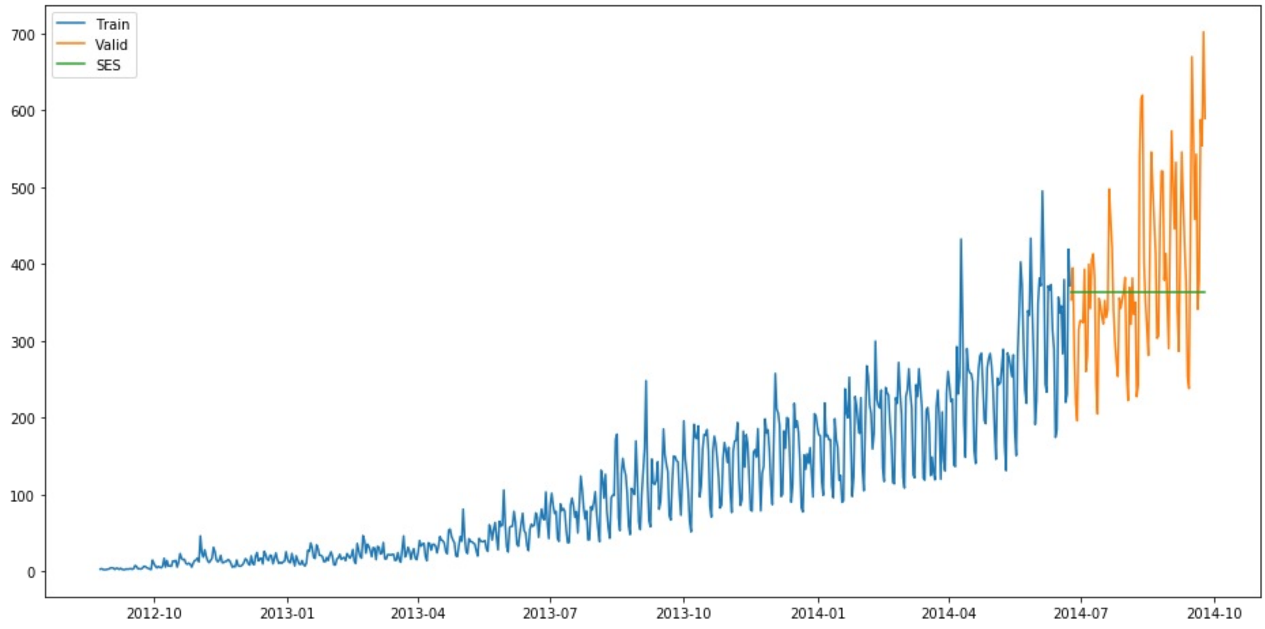
Let’s look at an example of simple exponential smoothing:

Example



Here the predictions are made by assigning larger weight to the recent values and lesser weight to the old values.

```
from statsmodels.tsa.api import ExponentialSmoothing,
SimpleExpSmoothing, Holt
y_hat_avg = valid.copy()
fit2 =

SimpleExpSmoothing(np.asarray(Train['Count'])).fit(smoothing_level=0.6,optimized=False)
y_hat_avg['SES'] = fit2.forecast(len(valid))
plt.figure(figsize=(16,8))
plt.plot(Train['Count'], label='Train')
plt.plot(valid['Count'], label='Valid')
plt.plot(y_hat_avg['SES'], label='SES')
plt.legend(loc='best')
plt.show()
```



```
rms = sqrt(mean_squared_error(valid.Count, y_hat_avg.SES))
```

```
113.43708111884514
```

We can infer that the fit of the model has improved as the rmse value has reduced.

iv) Holt’s Linear Trend Model

- It is an extension of simple exponential smoothing to allow forecasting of data with a trend.
- This method takes into account the trend of the dataset. The forecast function in this method is a function of level and trend.

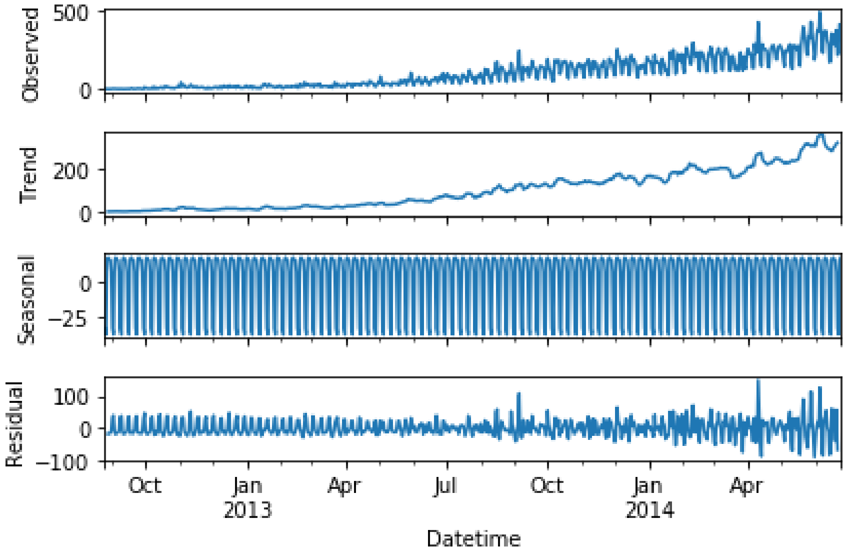
First of all let us visualize the trend, seasonality and error in the series.

We can decompose the time series in four parts.

- Observed, which is the original time series.
- Trend, which shows the trend in the time series, i.e., increasing or decreasing behaviour of the time series.
- Seasonal, which tells us about the seasonality in the time series.
- Residual, which is obtained by removing any trend or seasonality in the time series.

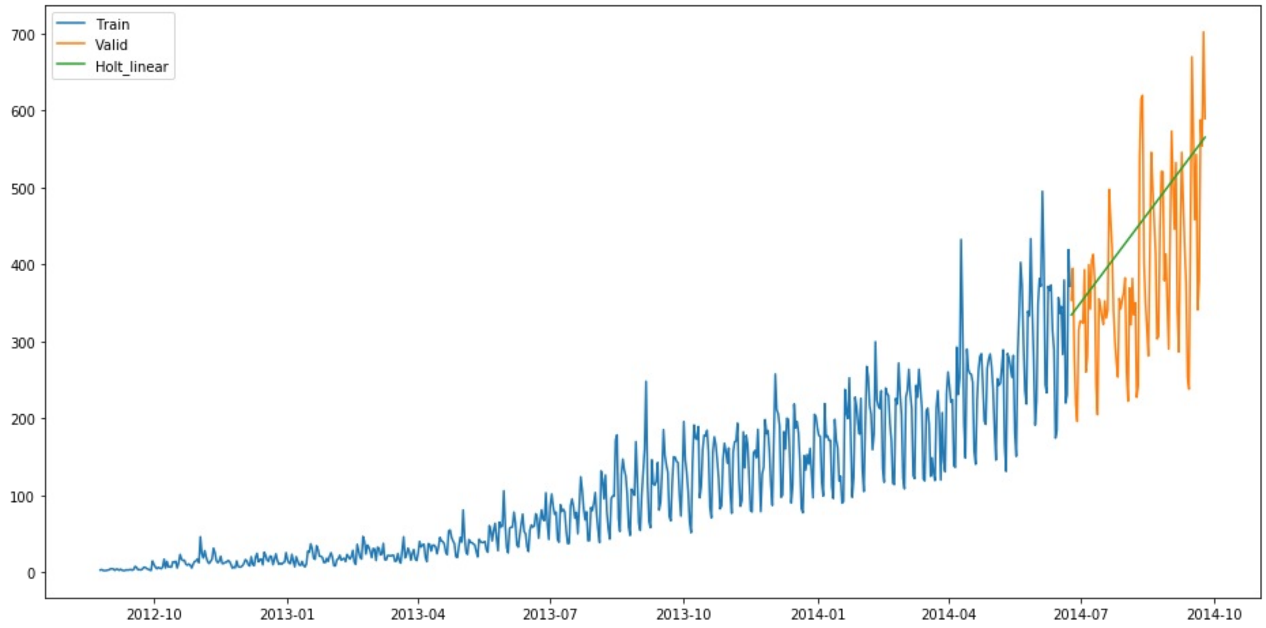
Lets visualize all these parts.

```
import statsmodels.api as sm
sm.tsa.seasonal_decompose(Train.Count).plot()
result = sm.tsa.stattools.adfuller(train.Count)
plt.show()
```



An increasing trend can be seen in the dataset, so now we will make a model based on the trend.

```
y_hat_avg = valid.copy()
fit1 = Holt(np.asarray(Train['Count'])).fit(smoothing_level = 0.3,smoothing_slope = 0.1)
y_hat_avg['Holt_linear'] = fit1.forecast(len(valid))
plt.figure(figsize=(16,8))
plt.plot(Train['Count'], label='Train')
plt.plot(valid['Count'], label='Valid')
plt.plot(y_hat_avg['Holt_linear'], label='Holt_linear')
plt.legend(loc='best')
plt.show()
```



We can see an inclined line here as the model has taken into consideration the trend of the time series.

Let’s calculate the rmse of the model.

```
rms = sqrt(mean_squared_error(valid.Count, y_hat_avg.Holt_linear))
```

```
112.94278345314041
```

It can be inferred that the rmse value has decreased.

Now we will be predicting the passenger count for the test dataset using various models.

Analytics Vidhya

courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6134232-holt-s-linear-trend-model-on-daily-time-series

3) Holt’s Linear Trend Model on daily time series

- Now let’s try to make holt’s linear trend model on the daily time series and make predictions on the test dataset.
- We will make predictions based on the daily time series and then will distribute that daily prediction to hourly predictions.
- We have fitted the holt’s linear trend model on the train dataset and validated it using validation dataset.

Now let’s load the submission file.

```
submission=pd.read_csv("submission.csv")
```

We only need ID and corresponding Count for the final submission.

Let’s make prediction for the test dataset.

```
predict=fit1.forecast(len(test))
```

Let’s save these predictions in test file in a new column.

```
test['prediction']=predict
```

Remember this is the daily predictions. We have to convert these predictions to hourly basis. * To do so we will first calculate the ratio of passenger count for each hour of every day. * Then we will find the average ratio of passenger count for every hour and we will get 24 ratios. * Then to calculate the hourly predictions we will multiply the daily prediction with the hourly ratio.

```
# Calculating the hourly ratio of count
train_original['ratio']=train_original['Count']/train_original['Count'].sum()

# Grouping the hourly ratio
temp=train_original.groupby(['Hour'])['ratio'].sum()

# Groupby to csv format
pd.DataFrame(temp, columns=['Hour','ratio']).to_csv('GROUPby.csv')

temp2=pd.read_csv("GROUPby.csv")
temp2=temp2.drop('Hour.1',1)

# Merge Test and test_original on day, month and year
merge=pd.merge(test, test_original, on=('day','month', 'year'), how='left')
merge['Hour']=merge['Hour_y']
merge=merge.drop(['year', 'month', 'Datetime','Hour_x','Hour_y'], axis=1)
# Predicting by merging merge and temp2
prediction=pd.merge(merge, temp2, on='Hour', how='left')

# Converting the ratio to the original scale
prediction['Count']=prediction['prediction']*prediction['ratio']*24 prediction['ID']=prediction['ID_y']
```

Let’s drop all other features from the submission file and keep ID and Count only.

```
submission=prediction.drop(['ID_x', 'day', 'ID_y','prediction','Hour', 'ratio'],axis=1)
# Converting the final submission to csv format
pd.DataFrame(submission, columns=['ID','Count']).to_csv('Holt linear.csv')
```

Holt’s linear model gave rmse of 274.1596 on the leaderboard.

Now let’s look at how well the Holt winters model predict the passenger counts for test dataset.

Analytics Vidhya

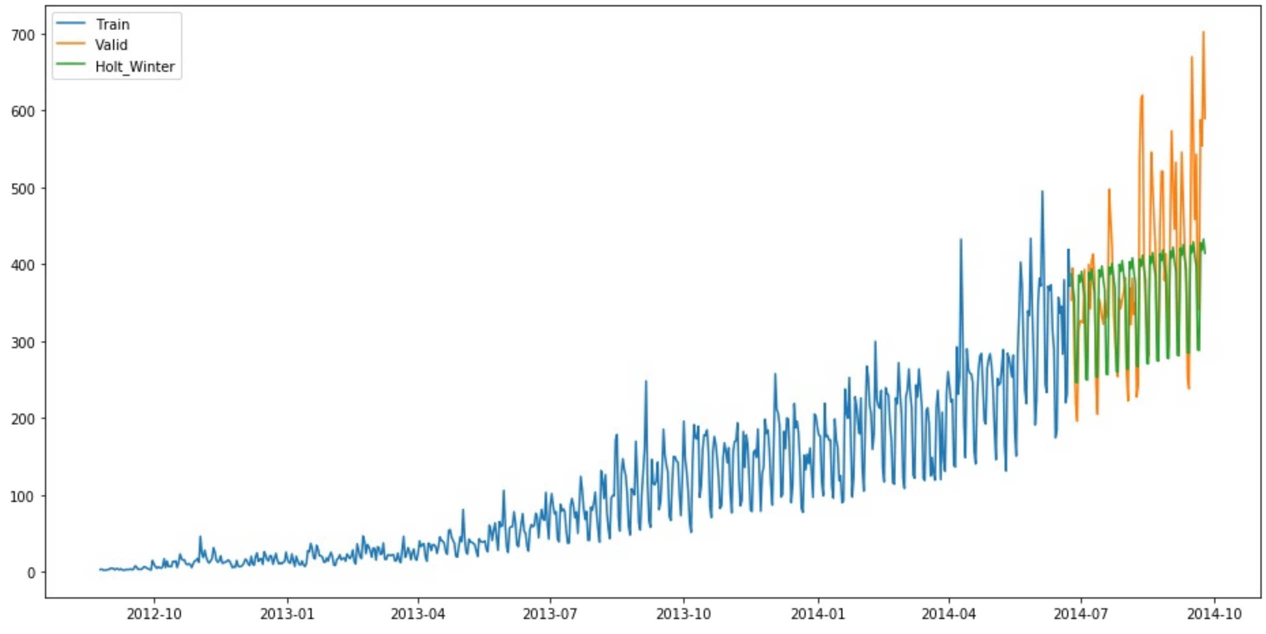
courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6142664-holt-winter-s-model-on-daily-time-series

4) Holt winter’s model on daily time series

- Datasets which show a similar set of pattern after fixed intervals of a time period suffer from seasonality.
- The above mentioned models don’t take into account the seasonality of the dataset while forecasting. Hence we need a method that takes into account both trend and seasonality to forecast future prices.
- One such algorithm that we can use in such a scenario is Holt’s Winter method. The idea behind Holt’s Winter is to apply exponential smoothing to the seasonal components in addition to level and trend.

Let’s first fit the model on training dataset and validate it using the validation dataset.

```
y_hat_avg = valid.copy()
fit1 = ExponentialSmoothing(np.asarray(Train['Count']) ,seasonal_periods=7 ,trend='add',
seasonal='add',).fit()
y_hat_avg['Holt_Winter'] = fit1.forecast(len(valid))
plt.figure(figsize=(16,8))
plt.plot( Train['Count'], label='Train')
plt.plot(valid['Count'], label='Valid')
plt.plot(y_hat_avg['Holt_Winter'], label='Holt_Winter')
plt.legend(loc='best')
plt.show()
```



```
rms = sqrt(mean_squared_error(valid.Count, y_hat_avg.Holt_Winter)) print(rms)
```

```
82.37373991413227
```

We can see that the rmse value has reduced a lot from this method. Let’s forecast the Counts for the entire length of the Test dataset.

```
predict=fit1.forecast(len(test))
```

Now we will convert these daily passenger count into hourly passenger count using the same approach which we followed above.

```
test['prediction']=predict

# Merge Test and test_original on day, month and year
merge=pd.merge(test, test_original, on=('day','month', 'year'), how='left')
merge['Hour']=merge['Hour_y']
merge=merge.drop(['year', 'month', 'Datetime','Hour_x','Hour_y'], axis=1)

# Predicting by merging merge and temp2
prediction=pd.merge(merge, temp2, on='Hour', how='left')

# Converting the ratio to the original scale
prediction['Count']=prediction['prediction']*prediction['ratio']*24
```

Let’s drop all features other than ID and Count

```
prediction['ID']=prediction['ID_y'] submission=prediction.drop(['day','Hour','ratio','prediction', 'ID_x',
'ID_y'],axis=1)

# Converting the final submission to csv format pd.DataFrame(submission, columns=
['ID','Count']).to_csv('Holt winters.csv')
```

- Holt winters model produced rmse of 328.356 on the leaderboard.
- The possible reason behind this may be that this model was not that good in predicting the trend of the time series but worked really well on the seasonality part.

Till now we have made different models for trend and seasonality. Can’t we make a model which will consider both the trend and seasonality of the time series?

Yes we can. We will look at the ARIMA model for time series forecasting.

5) Introduction to ARIMA model

- ARIMA stands for Auto Regression Integrated Moving Average. It is specified by three ordered parameters (p,d,q).

- Here p is the order of the autoregressive model(number of time lags)
- d is the degree of differencing(number of times the data have had past values subtracted)
- q is the order of moving average model. We will discuss more about these parameters in next section.

The ARIMA forecasting for a stationary time series is nothing but a linear (like a linear regression) equation.

What is a stationary time series?

There are three basic criterion for a series to be classified as stationary series :

- The mean of the time series should not be a function of time. It should be constant.
- The variance of the time series should not be a function of time.
- THe covariance of the ith term and the (i+m)th term should not be a function of time.

Why do we have to make the time series stationary?

We make the series stationary to make the variables independent. Variables can be dependent in various ways, but can only be independent in one way. So, we will get more information when they are independent. Hence the time series must be stationary.

If the time series is not stationary, firstly we have to make it stationary. For doing so, we need to remove the trend and seasonality from the data. To learn more about stationarity you can refer this article: https://www.analyticsvidhya.com/blog/2015/12/complete-tutorial-time-series-modeling/

Analytics Vidhya

courses.analyticsvidhya.com/courses/take/creating-time-series-forecast-using-python/texts/6142791-parameter-tuning-for-arima-model

6) Parameter tuning for ARIMA model

First of all we have to make sure that the time series is stationary. If the series is not stationary, we will make it stationary.

Stationarity Check

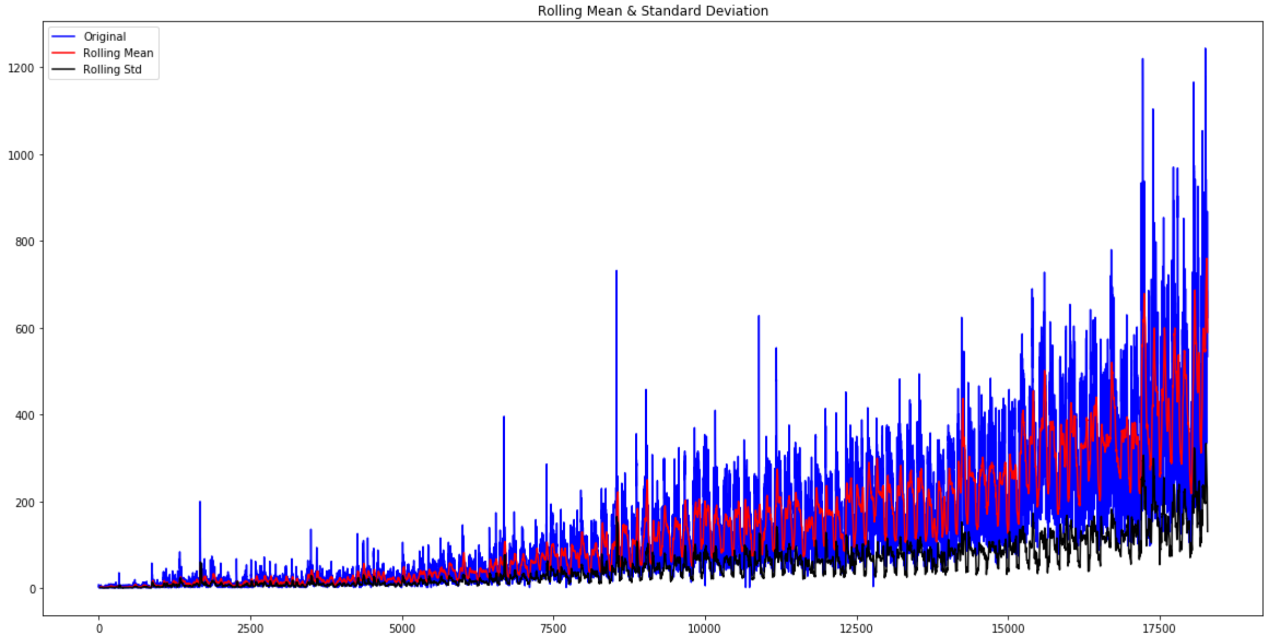
- We use Dickey Fuller test to check the stationarity of the series.
- The intuition behind this test is that it determines how strongly a time series is defined by a trend.
- The null hypothesis of the test is that time series is not stationary (has some time-dependent structure).
- The alternate hypothesis (rejecting the null hypothesis) is that the time series is stationary.

The test results comprise of a Test Statistic and some Critical Values for difference confidence levels. If the ‘Test Statistic’ is less than the ‘Critical Value’, we can reject the null hypothesis and say that the series is stationary.

We interpret this result using the Test Statistics and critical value. If the Test Statistics is smaller than critical value, it suggests we reject the null hypothesis (stationary), otherwise a greater Test Statistics suggests we accept the null hypothesis (non-stationary).

Let’s make a function which we can use to calculate the results of Dickey-Fuller test.

```
from statsmodels.tsa.stattools import adfuller
def test_stationarity(timeseries):
    #Determing rolling statistics
  rolmean = pd.rolling_mean(timeseries, window=24) # 24 hours on each day
  rolstd = pd.rolling_std(timeseries, window=24)
    #Plot rolling statistics:
  orig = plt.plot(timeseries, color='blue',label='Original')
  mean = plt.plot(rolmean, color='red', label='Rolling Mean')
  std = plt.plot(rolstd, color='black', label = 'Rolling Std')
  plt.legend(loc='best')
  plt.title('Rolling Mean & Standard Deviation')
  plt.show(block=False)
    #Perform Dickey-Fuller test:

  dftest = adfuller(timeseries, autolag='AIC')
  dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of
Observations Used'])

  for key,value in dftest[4].items():
    dfoutput['Critical Value (%s)'%key] = value

from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 20,10

test_stationarity(train_original['Count'])
```



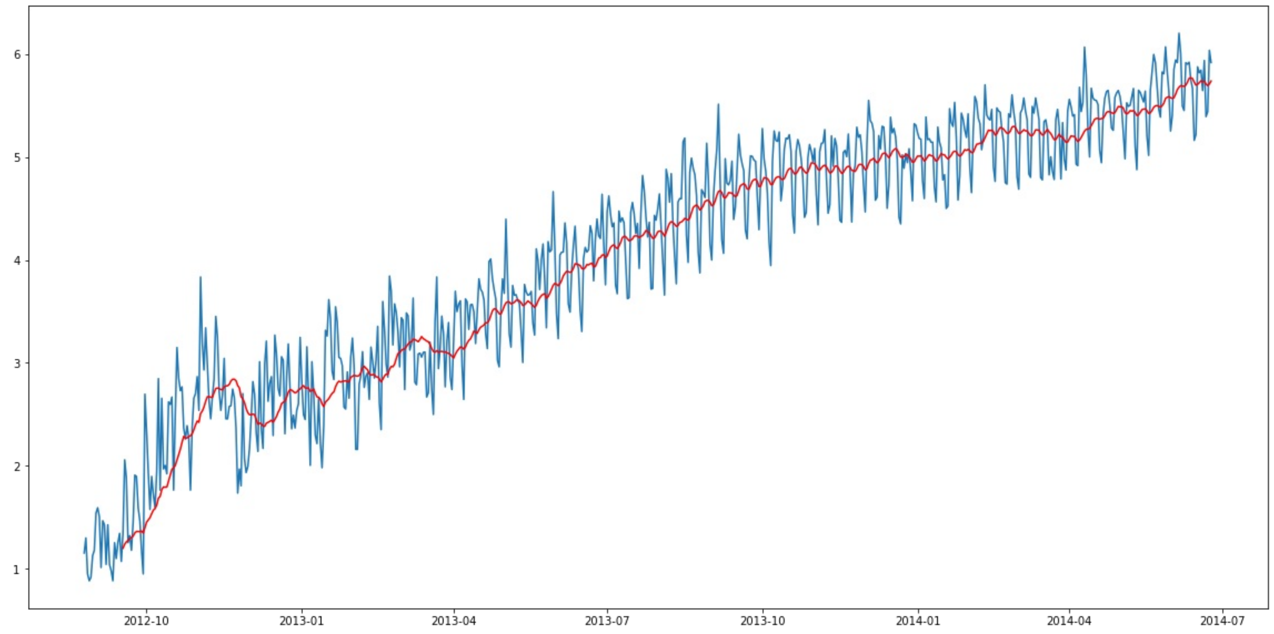
```
Results of Dickey-Fuller Test:
 Test Statistic                    -4.456561
 p-value                            0.000235
 #Lags Used                        45.000000
 Number of Observations Used    18242.000000
 Critical Value (1%)               -3.430709
 Critical Value (5%)               -2.861698
 Critical Value (10%)              -2.566854
 dtype: float64
```

The statistics shows that the time series is stationary as Test Statistic < Critical value but we can see an increasing trend in the data. So, firstly we will try to make the data more stationary. For doing so, we need to remove the trend and seasonality from the data.

Removing Trend

- A trend exists when there is a long-term increase or decrease in the data. It does not have to be linear.
- We see an increasing trend in the data so we can apply transformation which penalizes higher values more than smaller ones, for example log transformation.
- We will take rolling average here to remove the trend. We will take the window size of 24 based on the fact that each day has 24 hours.

```
Train_log = np.log(Train['Count'])
valid_log = np.log(valid['Count'])

moving_avg = pd.rolling_mean(Train_log, 24)
plt.plot(Train_log)
plt.plot(moving_avg, color = 'red')
plt.show()
```

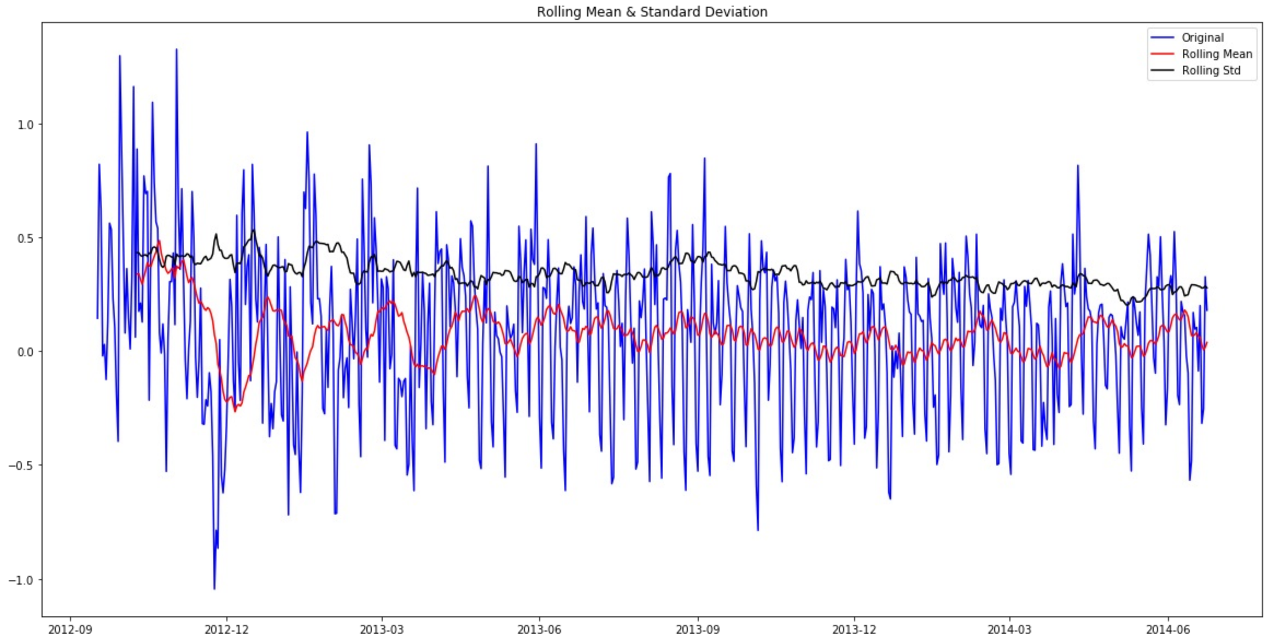


So we can observe an increasing trend. Now we will remove this increasing trend to make our time series stationary.

```
train_log_moving_avg_diff = Train_log - moving_avg
```

Since we took the average of 24 values, rolling mean is not defined for the first 23 values. So let's drop those null values.

```
train_log_moving_avg_diff.dropna(inplace = True) test_stationarity(train_log_moving_avg_diff)
```



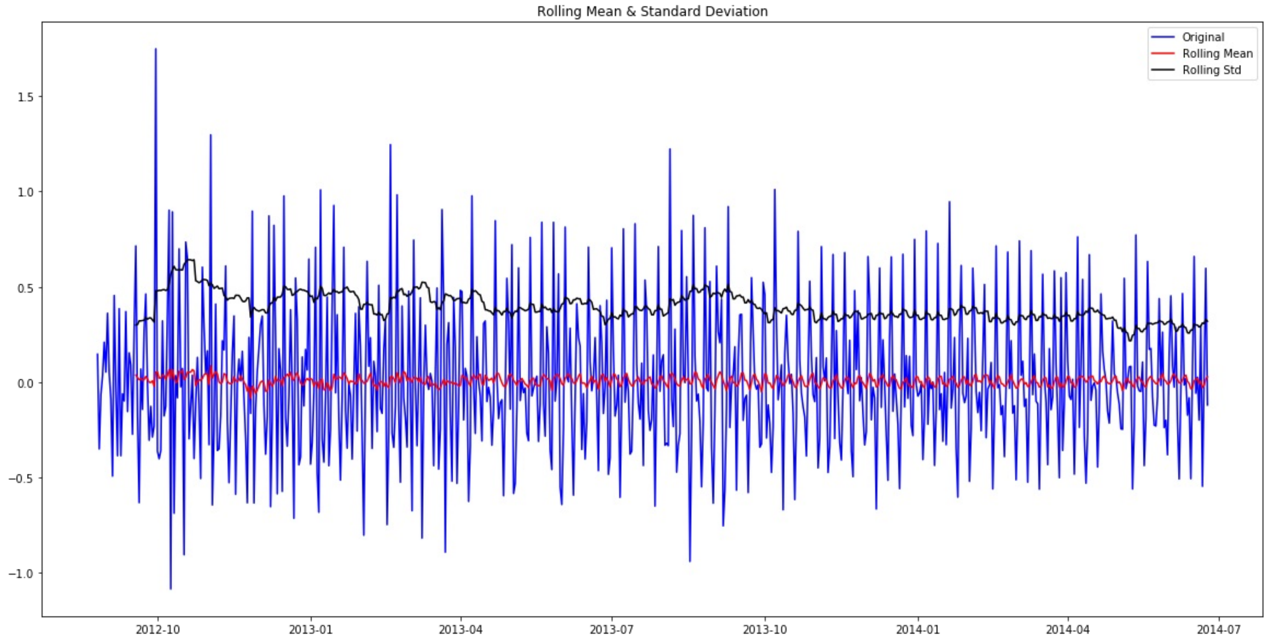
```
Results of Dickey-Fuller Test:
 Test Statistic                 -5.861646e+00
 p-value                        3.399422e-07
 #Lags Used                     2.000000e+01
 Number of Observations Used    6.250000e+02
 Critical Value (1%)           -3.440856e+00
 Critical Value (5%)           -2.866175e+00
 Critical Value (10%)          -2.569239e+00
 dtype: float64
```

We can see that the Test Statistic is very smaller as compared to the Critical Value. So, we can be confident that the trend is almost removed.

Let's now stabilize the mean of the time series which is also a requirement for a stationary time series.

> Differencing can help to make the series stable and eliminate the trend.

```
train_log_diff = Train_log - Train_log.shift(1)
test_stationarity(train_log_diff.dropna())
```



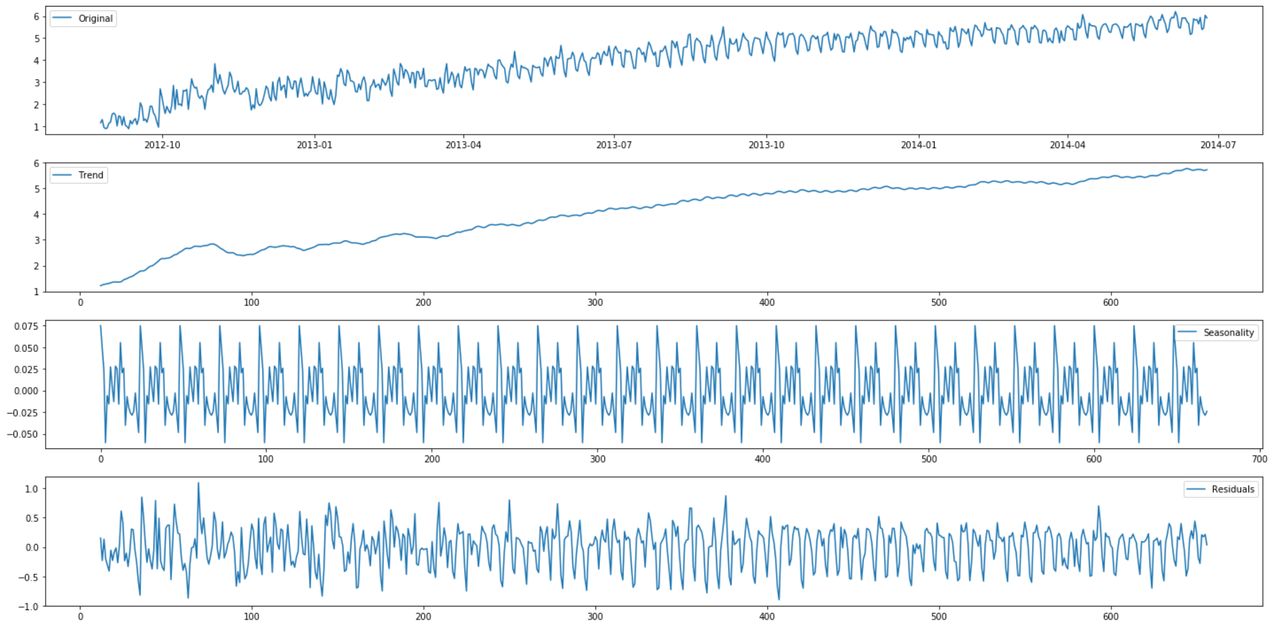
```
Results of Dickey-Fuller Test:
 Test Statistic                -8.237568e+00
 p-value                        5.834049e-13
 #Lags Used                     1.900000e+01
 Number of Observations Used    6.480000e+02
 Critical Value (1%)           -3.440482e+00
 Critical Value (5%)           -2.866011e+00
 Critical Value (10%)          -2.569151e+00
 dtype: float64
```

Now we will decompose the time series into trend and seasonality and will get the residual which is the random variation in the series.

Removing Seasonality

- By seasonality, we mean periodic fluctuations. A seasonal pattern exists when a series is influenced by seasonal factors (e.g., the quarter of the year, the month, or day of the week).
- Seasonality is always of a fixed and known period.
- We will use seasonal decompose to decompose the time series into trend, seasonality and residuals.

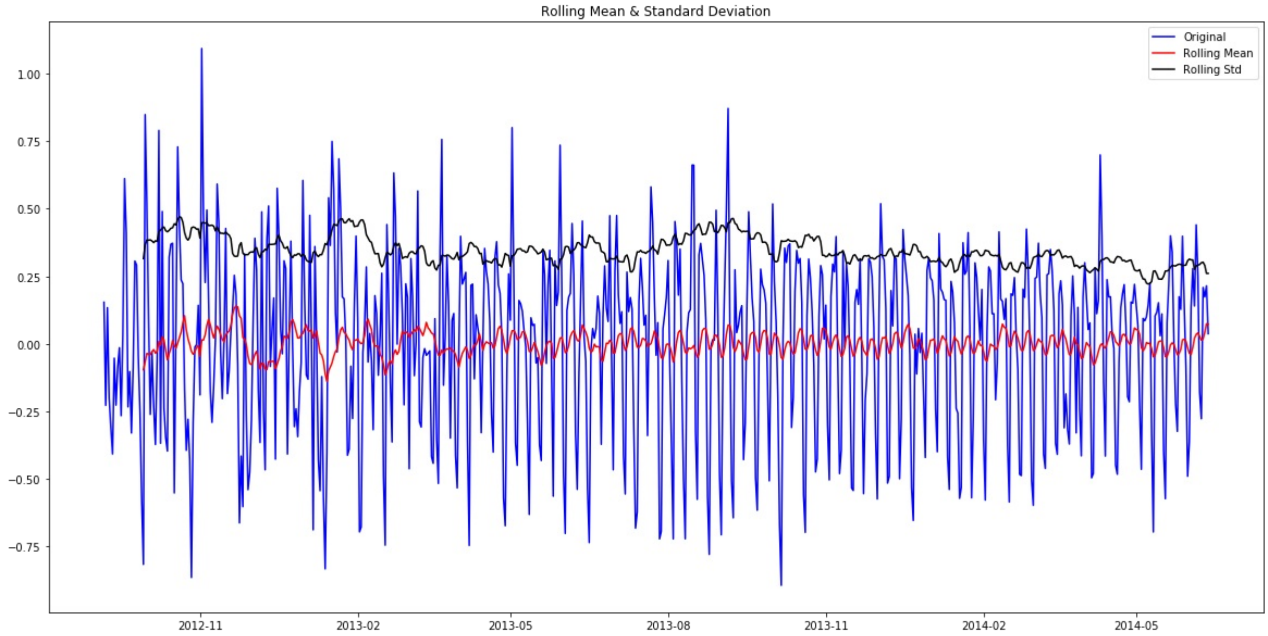
```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(pd.DataFrame(Train_log).Count.values, freq = 24)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(Train_log, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```



We can see the trend, residuals and the seasonality clearly in the above graph. Seasonality shows a constant trend in counter.

Let’s check stationarity of residuals.

```
train_log_decompose = pd.DataFrame(residual)
train_log_decompose['date'] = Train_log.index
train_log_decompose.set_index('date', inplace = True) train_log_decompose.dropna(inplace=True)
test_stationarity(train_log_decompose[0])
```



```
Results of Dickey-Fuller Test:

Test Statistic                -7.822096e+00
p-value                       6.628321e-12
#Lags Used                      2.000000e+01
Number of Observations Used    6.240000e+02
Critical Value (1%)           -3.440873e+00
Critical Value (5%)           -2.866183e+00
Critical Value (10%)           -2.569243e+00
dtype: float64
```

- It can be interpreted from the results that the residuals are stationary.
- Now we will forecast the time series using different models.

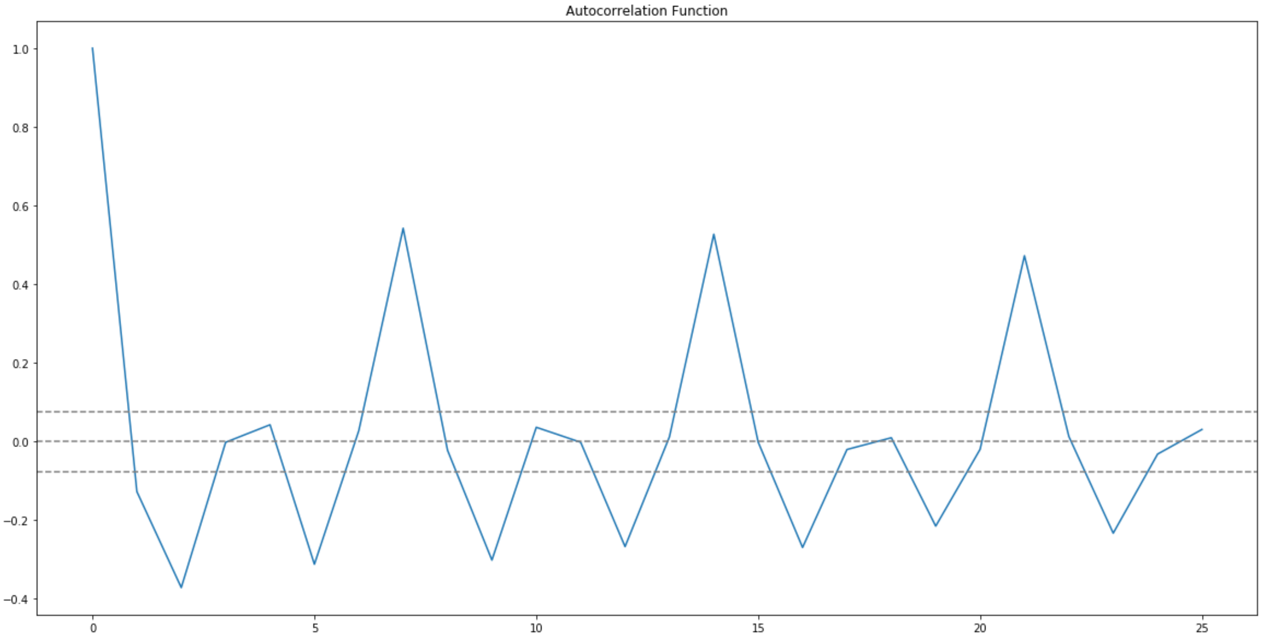
Forecasting the time series using ARIMA

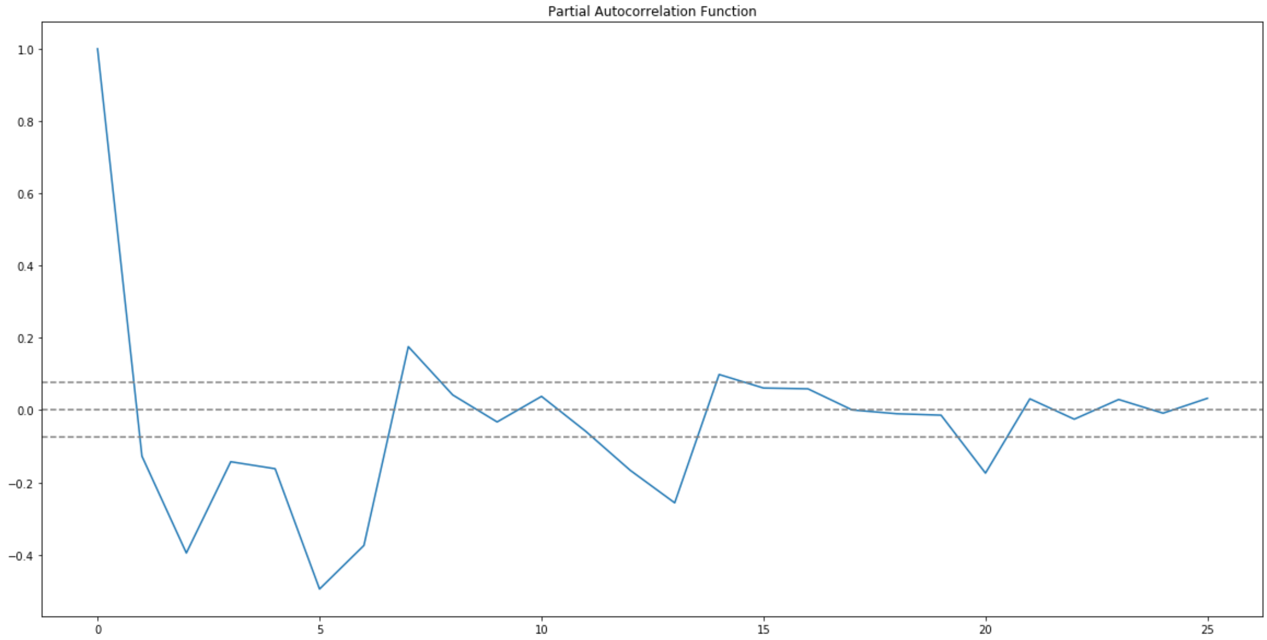
- First of all we will fit the ARIMA model on our time series for that we have to find the optimized values for the p,d,q parameters.
- To find the optimized values of these parameters, we will use ACF(Autocorrelation Function) and PACF(Partial Autocorrelation Function) graph.
- ACF is a measure of the correlation between the TimeSeries with a lagged version of itself.
- PACF measures the correlation between the TimeSeries with a lagged version of itself but after eliminating the variations already explained by the intervening comparisons.

```
from statsmodels.tsa.stattools import acf, pacf
lag_acf = acf(train_log_diff.dropna(), nlags=25)
lag_pacf = pacf(train_log_diff.dropna(), nlags=25, method='ols')
```

ACF and PACF plot

```
plt.plot(lag_acf)
plt.axhline(y=0,linestyle='--',color='gray') plt.axhline(y=-
1.96/np.sqrt(len(train_log_diff.dropna())),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(train_log_diff.dropna())),linestyle='--',color='gray')
plt.title('Autocorrelation Function')
plt.show()
plt.plot(lag_pacf)
plt.axhline(y=0,linestyle='--',color='gray') plt.axhline(y=-
1.96/np.sqrt(len(train_log_diff.dropna())),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(train_log_diff.dropna())),linestyle='--',color='gray') plt.title('Partial
Autocorrelation Function')
plt.show()
```





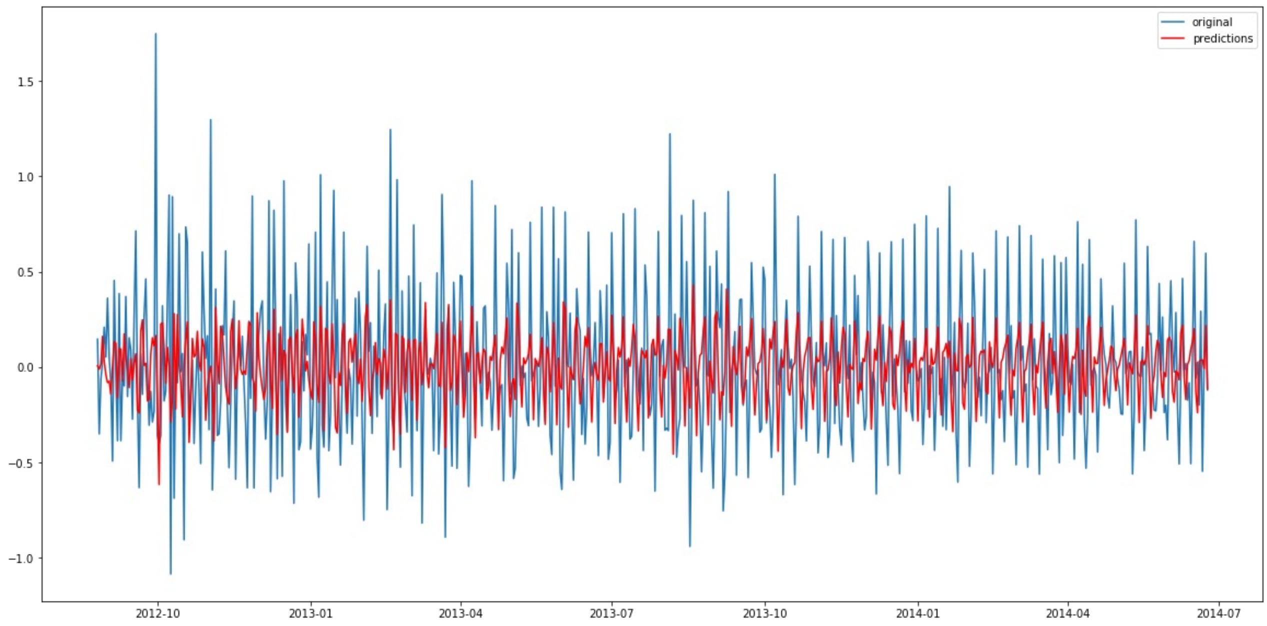
- p value is the lag value where the PACF chart crosses the upper confidence interval for the first time. It can be noticed that in this case p=1.
- q value is the lag value where the ACF chart crosses the upper confidence interval for the first time. It can be noticed that in this case q=1.

- Now we will make the ARIMA model as we have the p,q values. We will make the AR and MA model separately and then combine them together.

AR model

The autoregressive model specifies that the output variable depends linearly on its own previous values.

```
from statsmodels.tsa.arima_model import ARIMA

model = ARIMA(Train_log, order=(2, 1, 0))  # here the q value is zero since it is just the AR model
results_AR = model.fit(disp=-1)
plt.plot(train_log_diff.dropna(), label='original')
plt.plot(results_AR.fittedvalues, color='red', label='predictions')
plt.legend(loc='best')
plt.show()
```

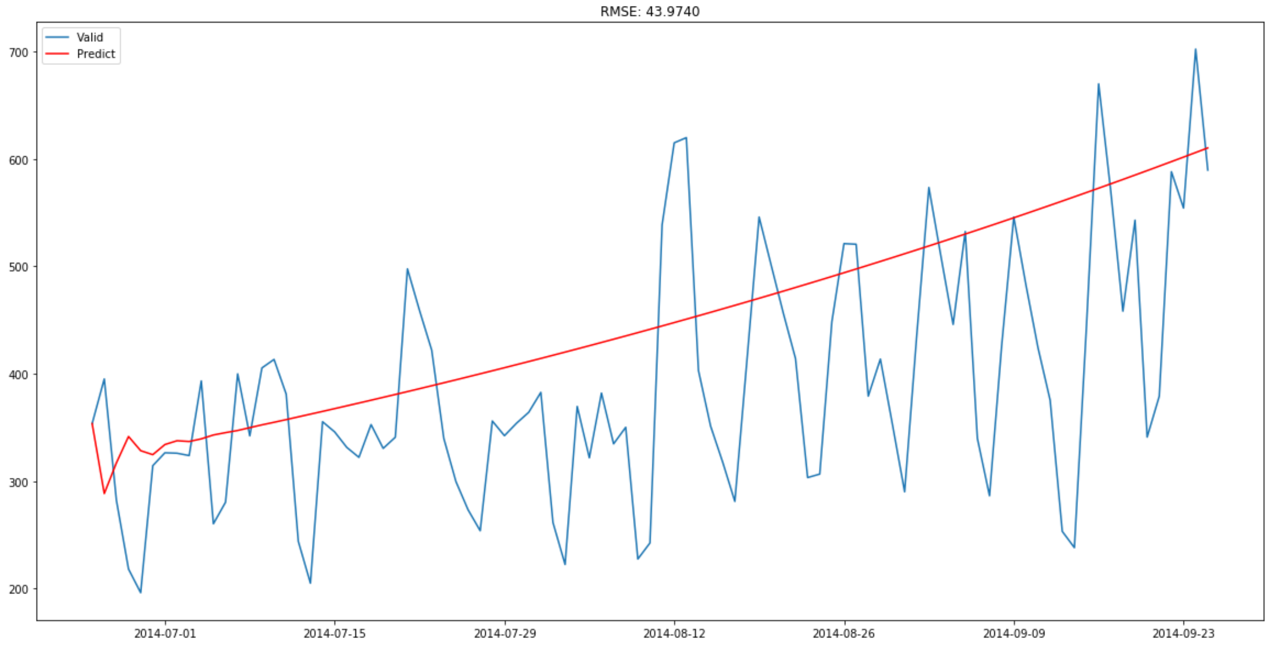


Lets plot the validation curve for AR model.

We have to change the scale of the model to the original scale.

First step would be to store the predicted results as a separate series and observe it.

```
AR_predict=results_AR.predict(start="2014-06-25", end="2014-09-25")
AR_predict=AR_predict.cumsum().shift().fillna(0) AR_predict1=pd.Series(np.ones(valid.shape[0]) *
np.log(valid['Count'])[0], index = valid.index)
AR_predict1=AR_predict1.add(AR_predict,fill_value=0)
AR_predict = np.exp(AR_predict1)

plt.plot(valid['Count'], label = "Valid")
plt.plot(AR_predict, color = 'red', label = "Predict")
plt.legend(loc= 'best')
plt.title('RMSE: %.4f'% (np.sqrt(np.dot(AR_predict, valid['Count']))/valid.shape[0])) plt.show()
```

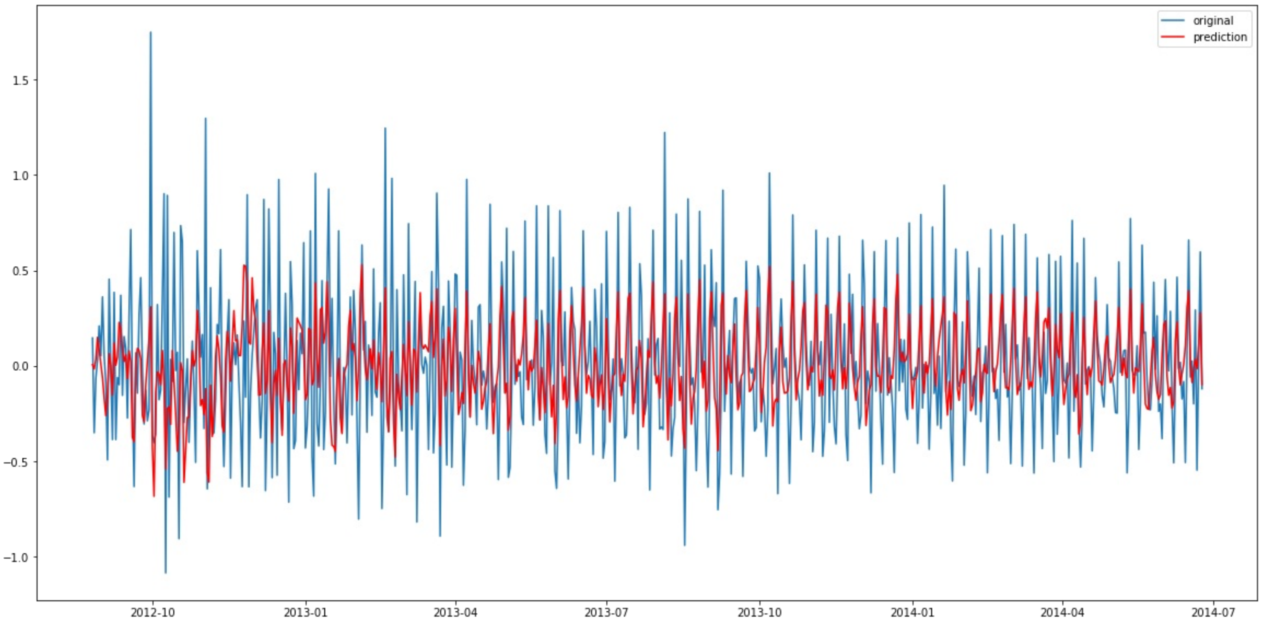


Here the red line shows the prediction for the validation set. Let's build the MA model now.

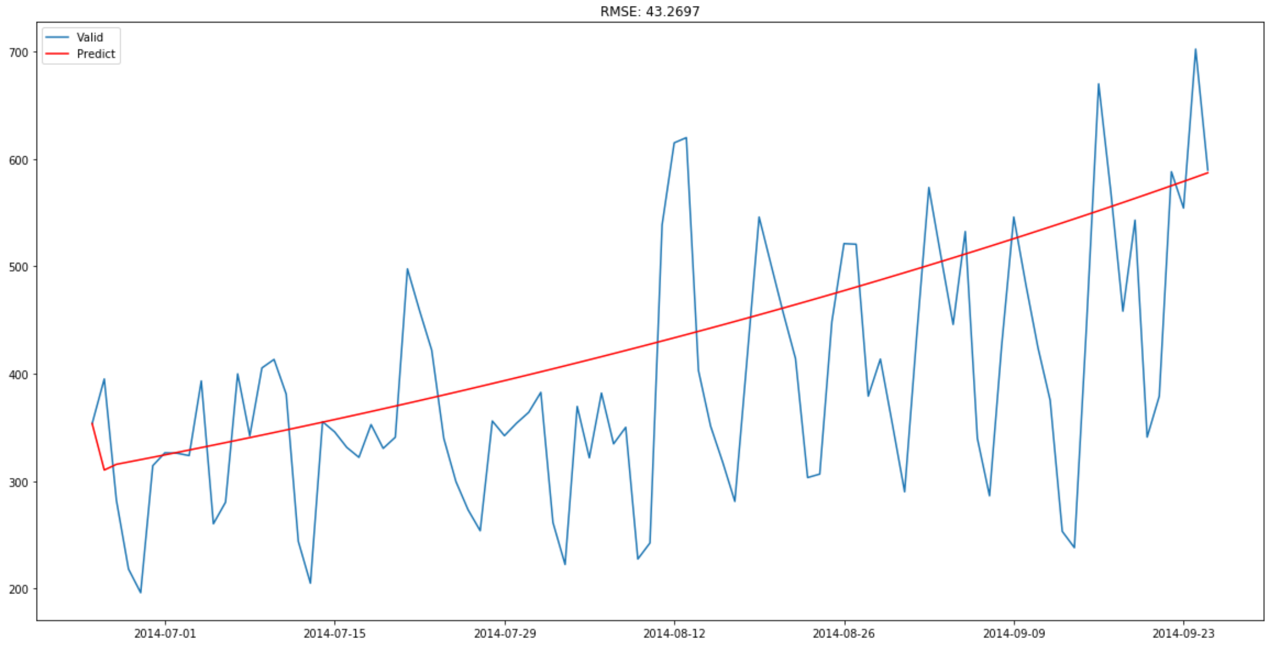
MA model

The moving-average model specifies that the output variable depends linearly on the current and various past values of a stochastic (imperfectly predictable) term.

```
model = ARIMA(Train_log, order=(0, 1, 2))  # here the p value is zero since it is just the MA model
results_MA = model.fit(disp=-1)
plt.plot(train_log_diff.dropna(), label='original')
plt.plot(results_MA.fittedvalues, color='red', label='prediction')
plt.legend(loc='best')
plt.show()
```



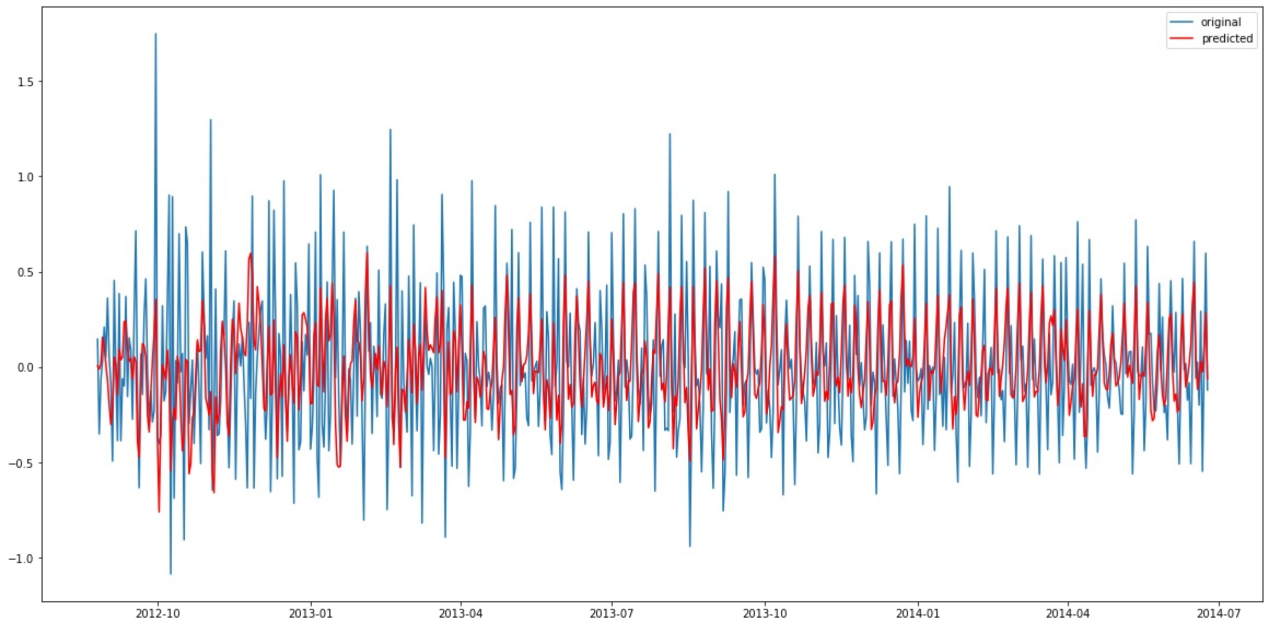
```
MA_predict=results_MA.predict(start="2014-06-25", end="2014-09-25")
MA_predict=MA_predict.cumsum().shift().fillna(0) MA_predict1=pd.Series(np.ones(valid.shape[0]) *
np.log(valid['Count'])[0], index = valid.index)
MA_predict1=MA_predict1.add(MA_predict,fill_value=0)
MA_predict = np.exp(MA_predict1)

plt.plot(valid['Count'], label = "Valid")
plt.plot(MA_predict, color = 'red', label = "Predict")
plt.legend(loc= 'best')
plt.title('RMSE: %.4f'% (np.sqrt(np.dot(MA_predict, valid['Count']))/valid.shape[0])) plt.show()
```



Now let’s combine these two models.

Combined model

```
model = ARIMA(Train_log, order=(2, 1, 2))
results_ARIMA = model.fit(disp=-1)
plt.plot(train_log_diff.dropna(),  label='original')
plt.plot(results_ARIMA.fittedvalues, color='red', label='predicted')
plt.legend(loc='best')
plt.show()
```

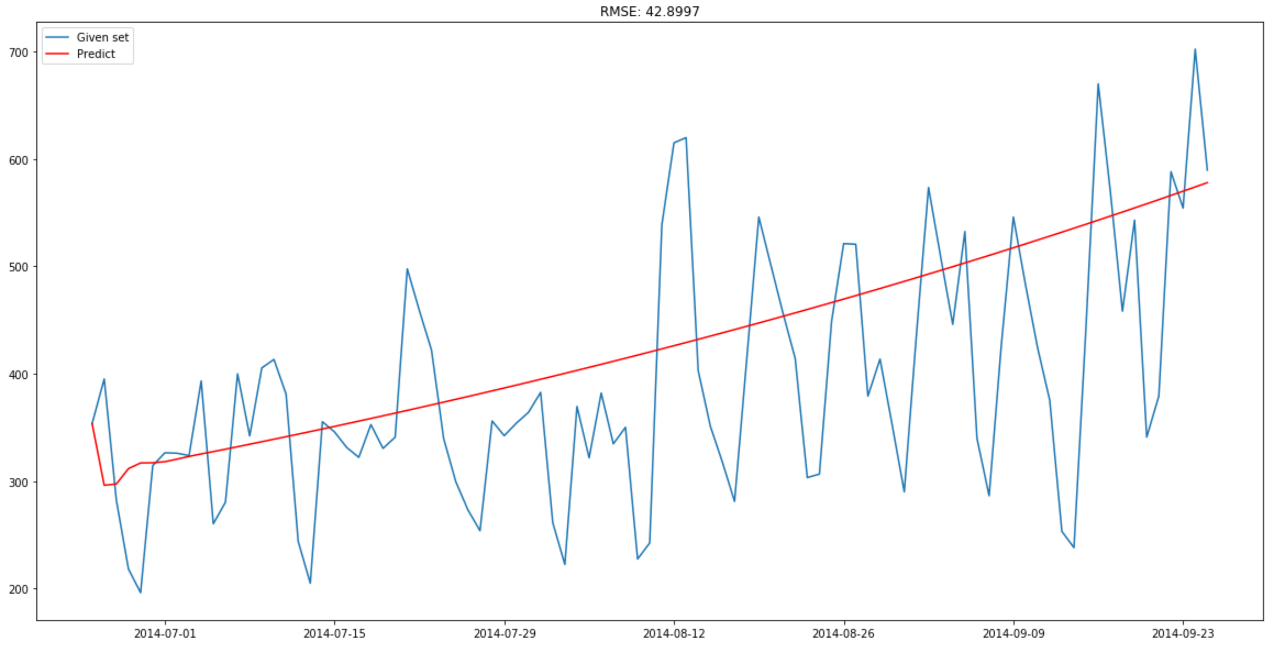


Let’s define a function which can be used to change the scale of the model to the original scale.

```
def check_prediction_diff(predict_diff, given_set):
    predict_diff= predict_diff.cumsum().shift().fillna(0)
    predict_base = pd.Series(np.ones(given_set.shape[0]) * np.log(given_set['Count'])[0], index =
given_set.index)
    predict_log = predict_base.add(predict_diff,fill_value=0)
    predict = np.exp(predict_log)

    plt.plot(given_set['Count'], label = "Given set")
    plt.plot(predict, color = 'red', label = "Predict")
    plt.legend(loc= 'best')
    plt.title('RMSE: %.4f'% (np.sqrt(np.dot(predict, given_set['Count']))/given_set.shape[0]))
    plt.show()

def check_prediction_log(predict_log, given_set):
    predict = np.exp(predict_log)

    plt.plot(given_set['Count'], label = "Given set")
    plt.plot(predict, color = 'red', label = "Predict")
    plt.legend(loc= 'best')
    plt.title('RMSE: %.4f'% (np.sqrt(np.dot(predict, given_set['Count']))/given_set.shape[0]))
    plt.show()
```

Let’s predict the values for validation set.

```
ARIMA_predict_diff=results_ARIMA.predict(start="2014-06-25", end="2014-09-25")

check_prediction_diff(ARIMA_predict_diff, valid)
```

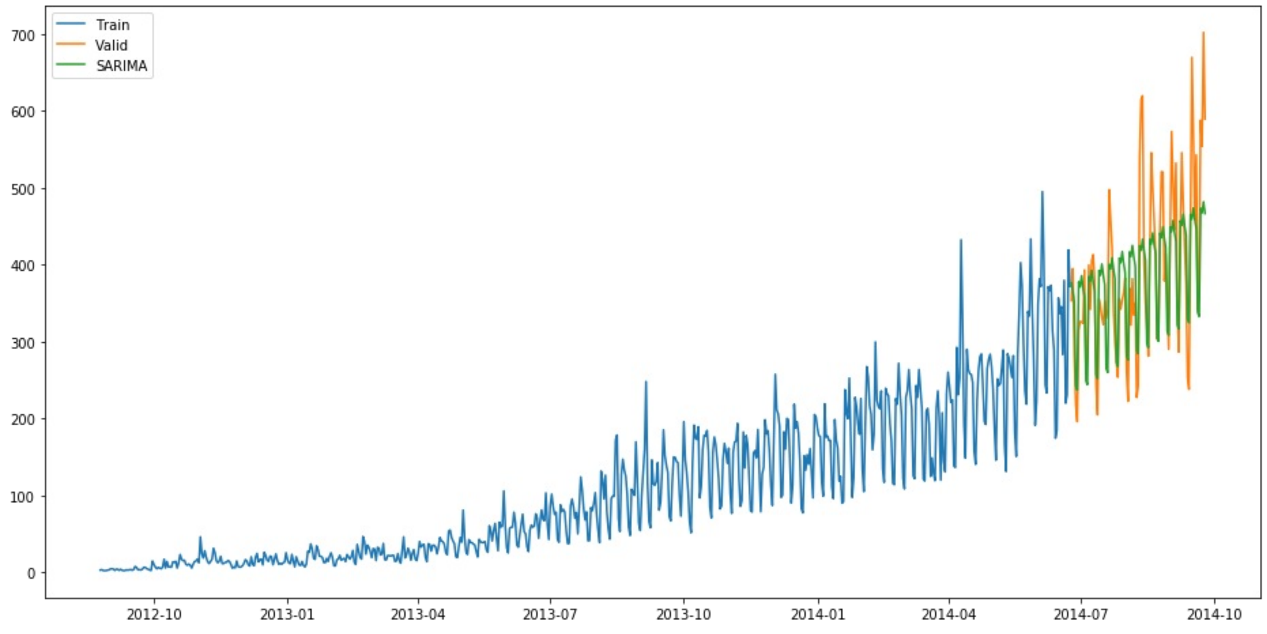

RMSE: 42.8997
Given set
Predict
700
600
500
400
300
200
2014-07-01
2014-07-15
2014-07-29
2014-08-12
2014-08-26
2014-09-09
2014-09-23

7) SARIMAX model on daily time series

SARIMAX model takes into account the seasonality of the time series. So we will build a SARIMAX model on the time series.

```
import statsmodels.api as sm

y_hat_avg = valid.copy()
fit1 = sm.tsa.statespace.SARIMAX(Train.Count, order=(2, 1, 4),seasonal_order=(0,1,1,7)).fit()
y_hat_avg['SARIMA'] = fit1.predict(start="2014-6-25", end="2014-9-25", dynamic=True)
plt.figure(figsize=(16,8))
plt.plot( Train['Count'], label='Train')
plt.plot(valid['Count'], label='Valid')
plt.plot(y_hat_avg['SARIMA'], label='SARIMA')
plt.legend(loc='best')
plt.show()
```

```
/home/pulkit/miniconda3/envs/av/lib/python3.6/site-packages/statsmodels-0.8.0-py3.6-linux-x86_64.egg/statsmodels/base/model.py:511: ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```



- Order in the above model represents the order of the autoregressive model(number of time lags), the degree of differencing(number of times the data have had past values subtracted) and the order of moving average model.

- Seasonal order represents the order of the seasonal component of the model for the AR parameters, differences, MA parameters, and periodicity.

- In our case the periodicity is 7 since it is daily time series and will repeat after every 7 days.

Let’s check the rmse value for the validation part.

```
rms = sqrt(mean_squared_error(valid.Count, y_hat_avg.SARIMA))
```

```
69.70093730473587
```

Now we will forecast the time series for Test data which starts from 2014-9-26 and ends at 2015-4-26.

```
predict=fit1.predict(start="2014-9-26", end="2015-4-26", dynamic=True)
```

Note that these are the daily predictions and we need hourly predictions. So, we will distribute this daily prediction into hourly counts. To do so, we will take the ratio of hourly distribution of passenger count from train data and then we will distribute the predictions in the same ratio.

```
test['prediction']=predict

# Merge Test and test_original on day, month and year
merge=pd.merge(test, test_original, on=('day','month', 'year'), how='left')
merge['Hour']=merge['Hour_y']
merge=merge.drop(['year', 'month', 'Datetime','Hour_x','Hour_y'], axis=1)

# Predicting by merging merge and temp2
prediction=pd.merge(merge, temp2, on='Hour', how='left')

# Converting the ratio to the original scale
prediction['Count']=prediction['prediction']*prediction['ratio']*24
```

Let’s drop all variables other than ID and Count

```
prediction['ID']=prediction['ID_y'] submission=prediction.drop(['day','Hour','ratio','prediction', 'ID_x', 'ID_y'],axis=1)

# Converting the final submission to csv format
pd.DataFrame(submission, columns=['ID','Count']).to_csv('SARIMAX.csv')
```

This method gave us the least rmse score. The rmse on the leaderboard was 219.095.

What else can be tried to improve your model further?

- You can try to make a weekly time series and make predictions for that series and then distribute those predictions into daily and then hourly predictions.

- Use combination of models(ensemble) to reduce the rmse. To read more about ensemble techniques you can refer these articles:

- https://www.analyticsvidhya.com/blog/2015/08/introduction-ensemble-learning/
- https://www.analyticsvidhya.com/blog/2015/09/questions-ensemble-modeling/
- To read further about the time series analysis you can refer these articles:
- https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/
- https://www.analyticsvidhya.com/blog/2018/02/time-series-forecasting-methods/