Name: CHITIPOLU SRI SUDHEERA

Forecast traffic on a mode of transportation

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, need to forecast the traffic on JetRail for the next 7 months.

Table of Contents¶

7) SARIMAX model on daily time series

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:
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. The traffic will be high from May to October.
Explanation - Tourist visits generally increases during this time period. 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 Traffic during the peak hours will be high.
Explanation - People will travel to work, college.
Import all the packages

In [1]:

```
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")
```

Read the train and test data

In [3]:

```
train=pd.read_csv("C:/Users/ADMIN/Desktop/Train_SU63ISt.csv")


test=pd.read_csv("C:/Users/ADMIN/Desktop/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.

In [4]:

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

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.

In [5]:

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

Out[5]:

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

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.

In [6]:

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

Out[6]:

```
(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.

In [7]:

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

Out[7]:

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

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.

In [8]:

```
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.

In [9]:

```
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.

In [10]:

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

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

In [11]:

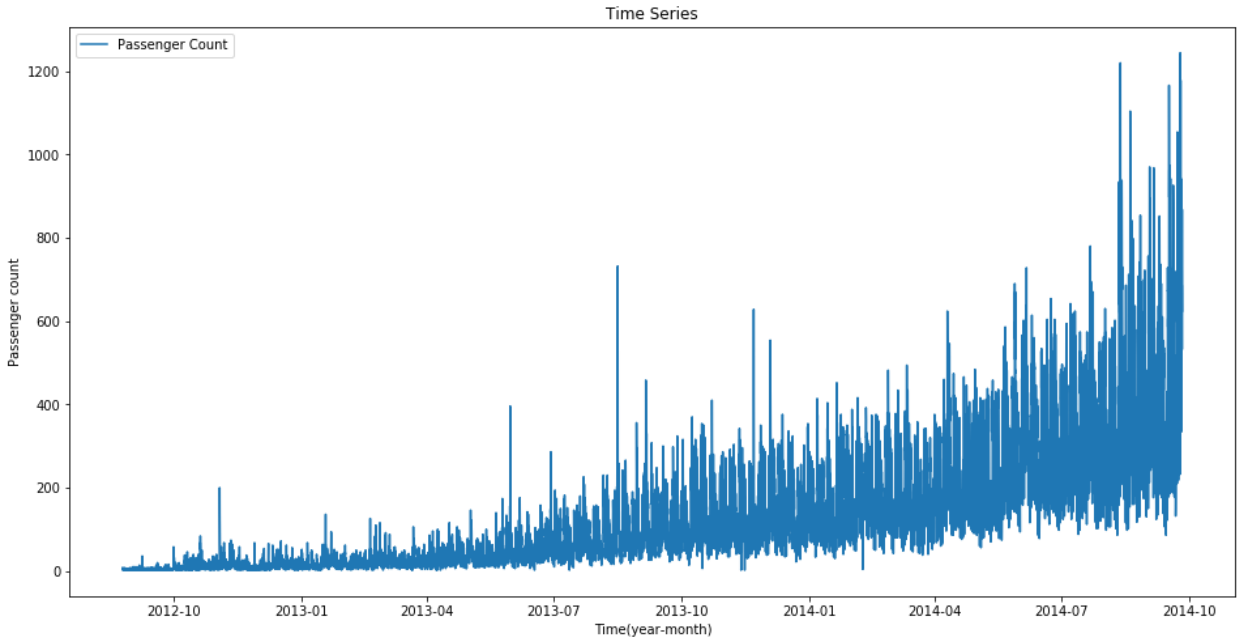
```
def applyer(row):
    if row.dayofweek == 5 or row.dayofweek == 6:
        return 1
    else:
        return 0


temp2 = train['Datetime'].apply(applyer)
train['weekend']=temp2
```

In [12]:

```
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')
```

Out[12]:

```
<matplotlib.legend.Legend at 0x27cf6462ba8>
```



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.

5) Exploratory Analysis¶

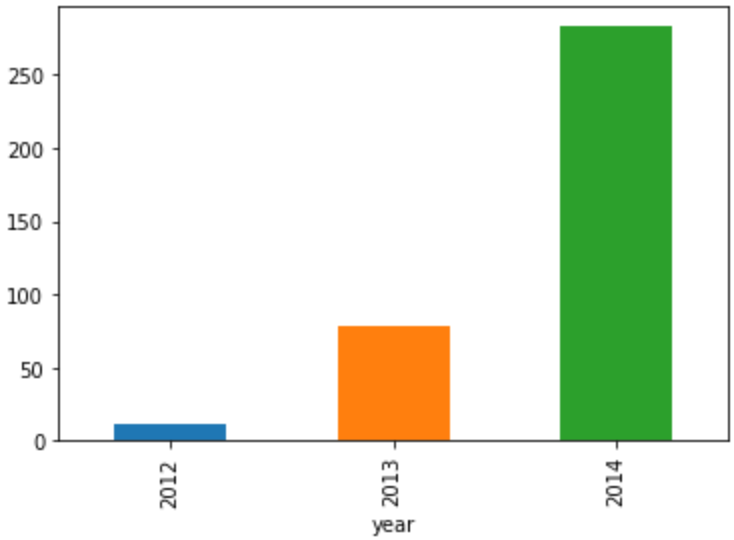
Let us try to verify our hypothesis using the actual data.
Our first hypothesis was traffic will increase as the years pass by. So let's look at yearly passenger count.

In [13]:

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

Out[13]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x27cf6409710>
```



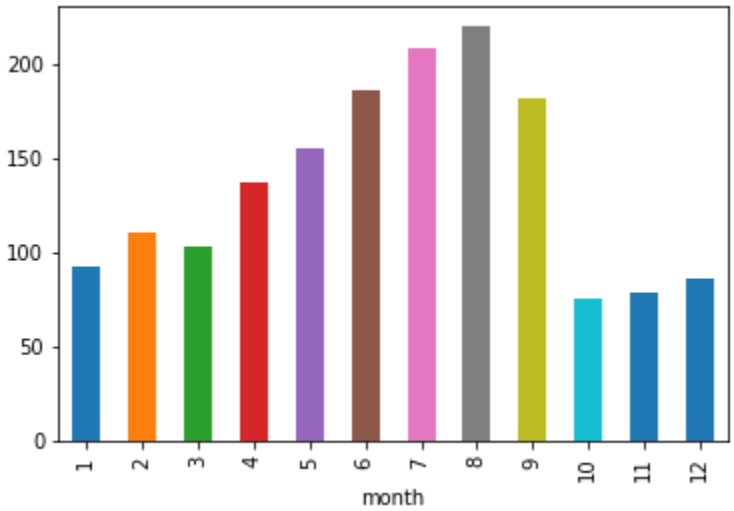
We see an exponential growth in the traffic with respect to year which validates our hypothesis. Our second hypothesis was about increase in traffic from May to October. So, let's see the relation between count and month

In [14]:

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

Out[14]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x27cf6433710>
```



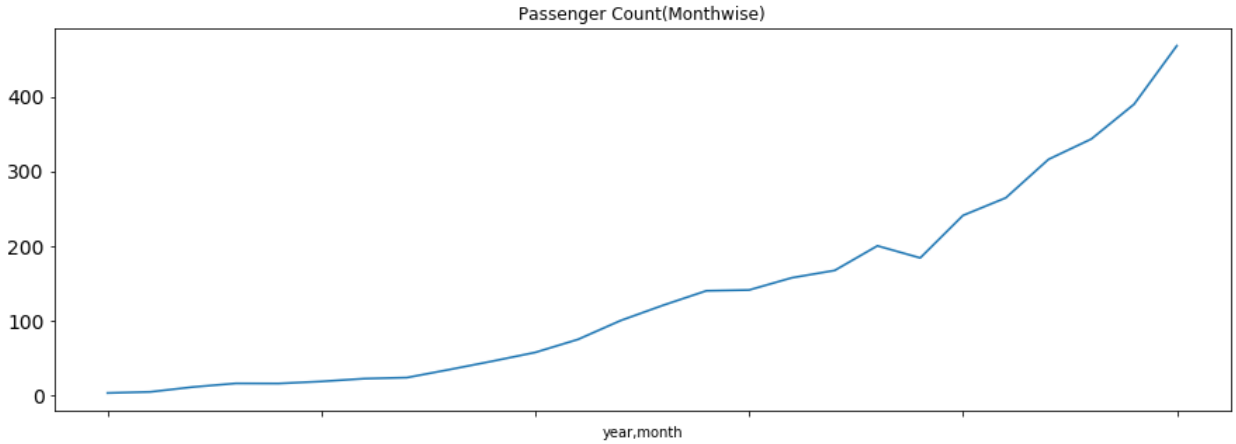
Here we see a decrease in the mean of passenger count in last three months. This does not look right. Let's look at the monthly mean of each year separately.

In [15]:

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

Out[15]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x27cf64aad30>
```



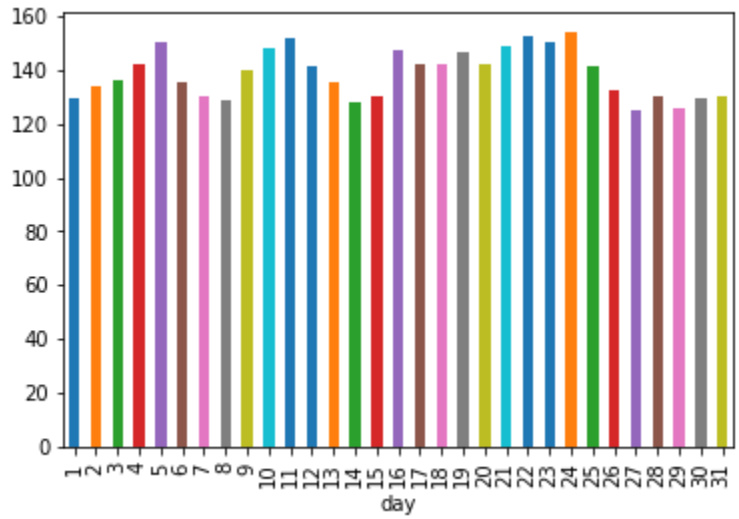
We see that the months 10, 11 and 12 are not present for the year 2014 and the mean value for these months in year 2012 is very less. Since there is an increasing trend in our time series, the mean value for rest of the months will be more because of their larger passenger counts in year 2014 and we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

In [16]:

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

Out[16]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x27cf6554978>
```



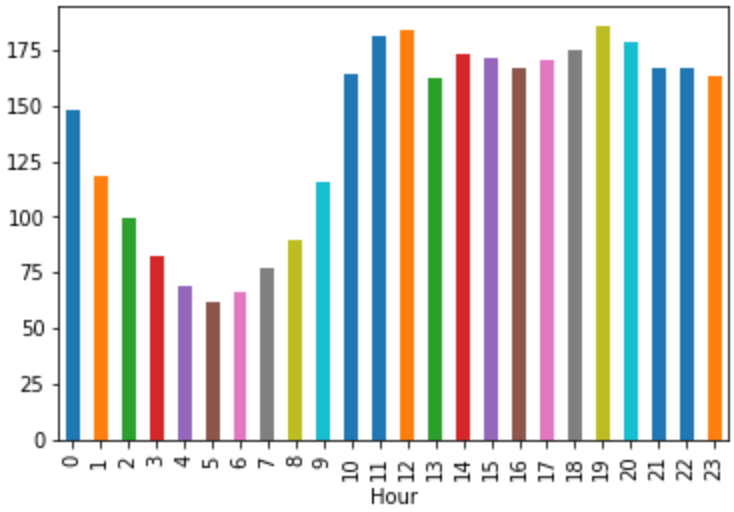
We are not getting much insights from day wise count of the passengers.
We also made a hypothesis that the traffic will be more during peak hours. So let's see the mean of hourly passenger count.

In [17]:

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

Out[17]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x27cf97ef198>
```



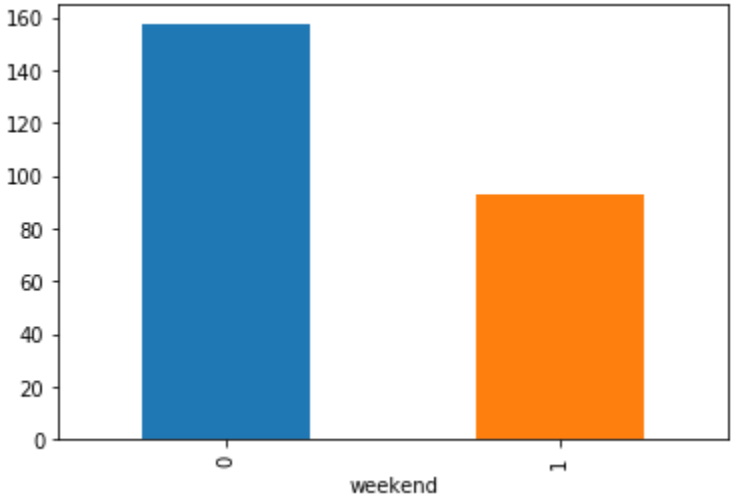
It can be inferred that the peak traffic is at 7 PM and then we see a decreasing trend till 5 AM.
After that the passenger count starts increasing again and peaks again between 11AM and 12 Noon.
Let's try to validate our hypothesis in which we assumed that the traffic will be more on weekdays.

In [18]:

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

Out[18]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x27cf98b60f0>
```



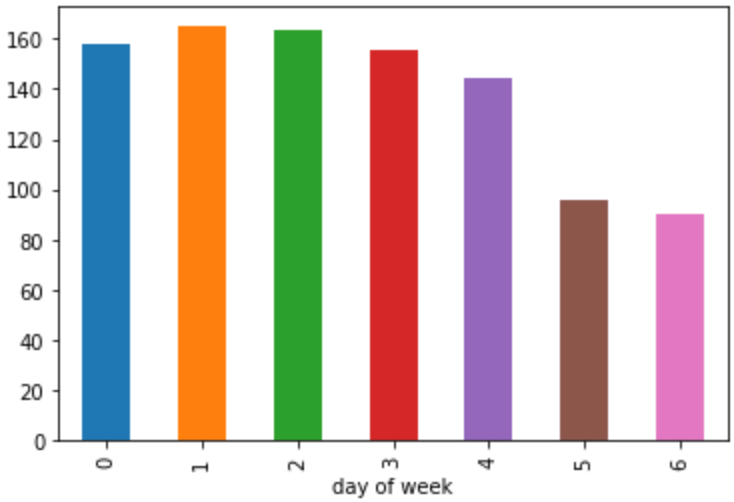
It can be inferred from the above plot that the traffic is more on weekdays as compared to weekends which validates our hypothesis.
Now we will try to look at the day wise passenger count.
Note - 0 is the starting of the week, i.e., 0 is Monday and 6 is Sunday.

In [19]:

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

Out[19]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x27cf98cae48>
```



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.

In [20]:

```
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.

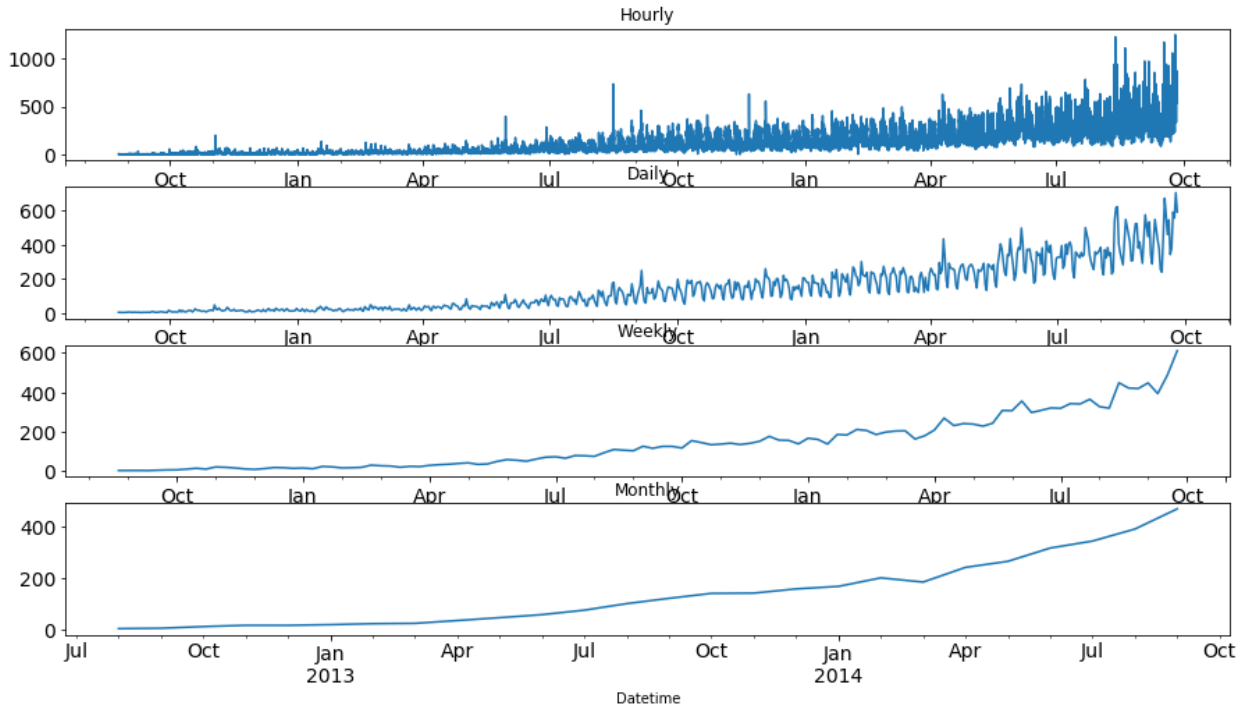
In [21]:

```
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.

In [22]:

```
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.

In [23]:

```
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()
```

Splitting the data into training and validation part¶

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.

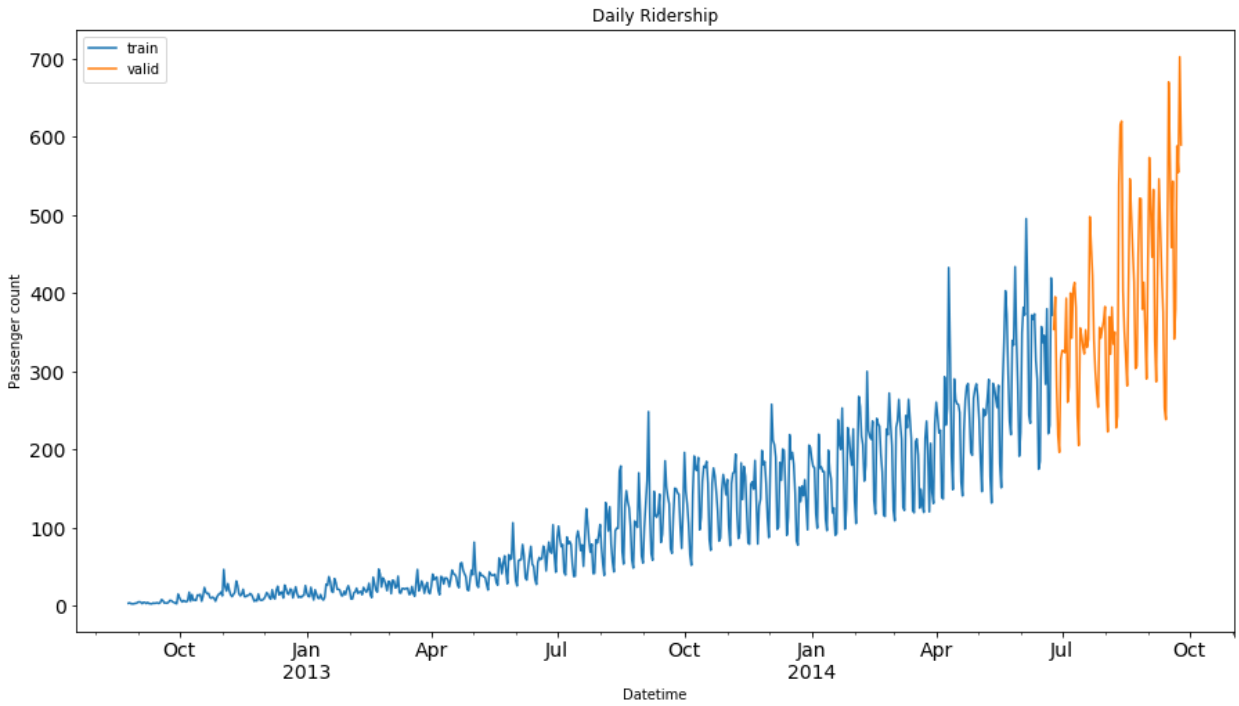
In [24]:

```
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.

In [25]:

```
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.

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

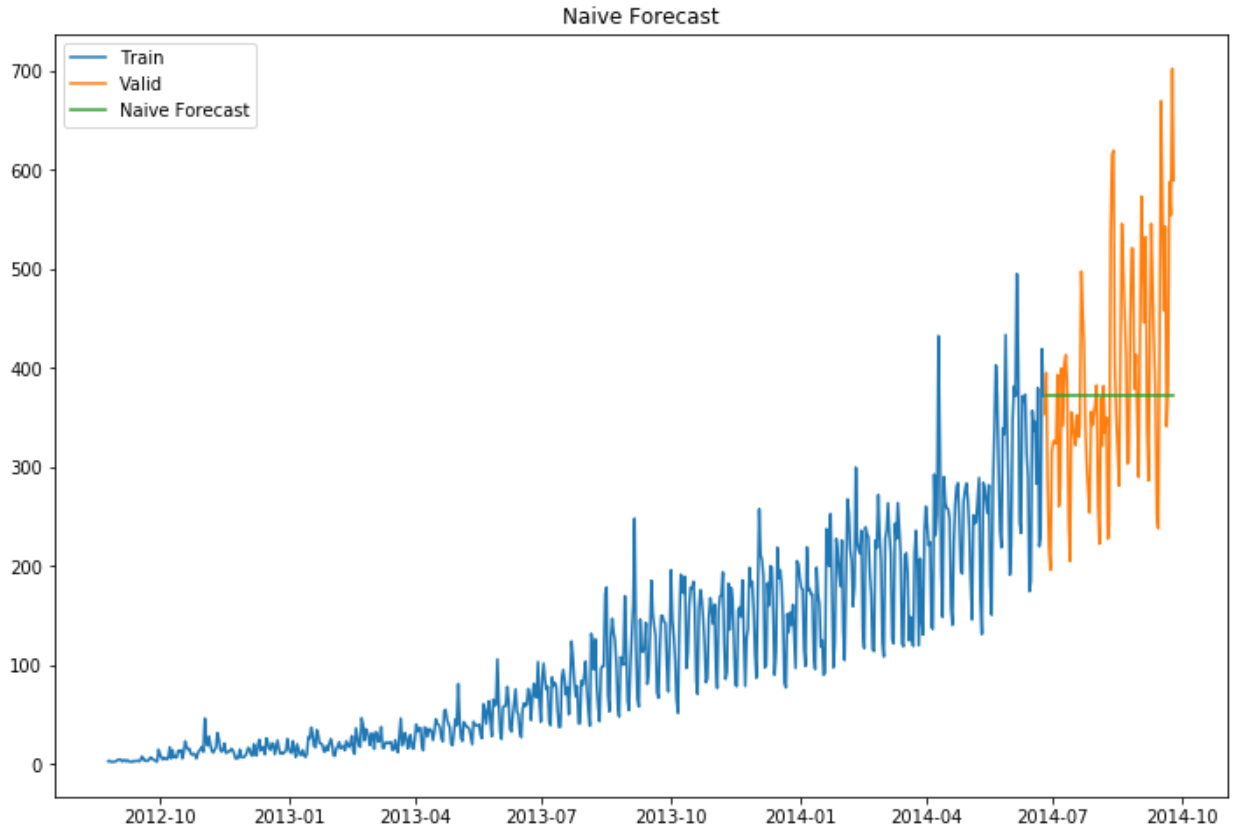
i) Naive Approach¶

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

In [26]:

```
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()
```



In [27]:

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

```
111.79050467496724
```

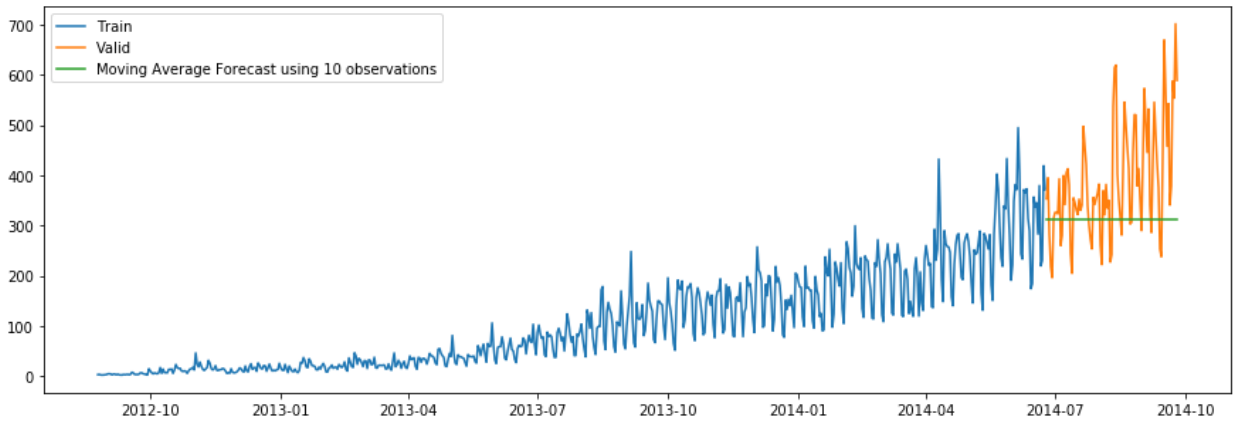
ii) Moving Average¶

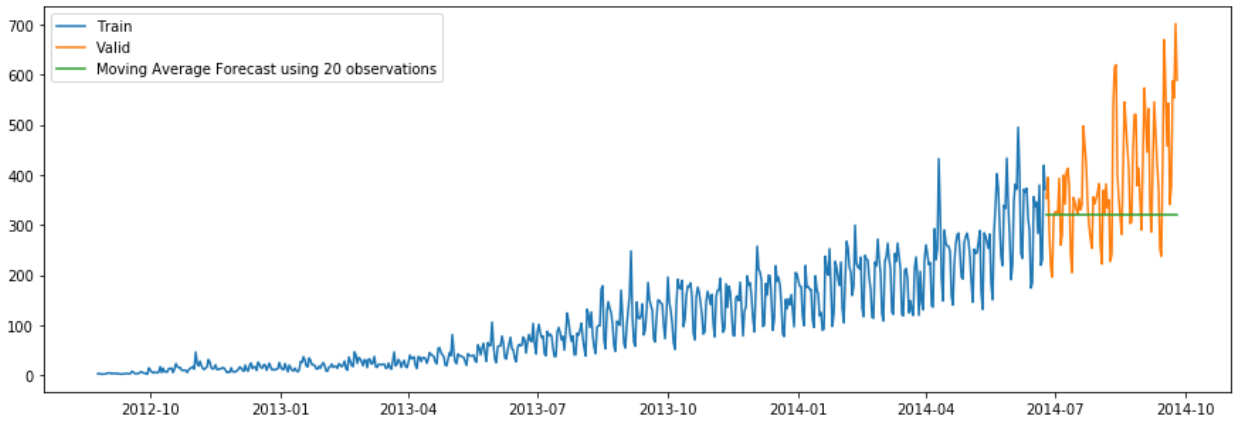
In this technique we will take the average of the passenger counts for last few time periods only.

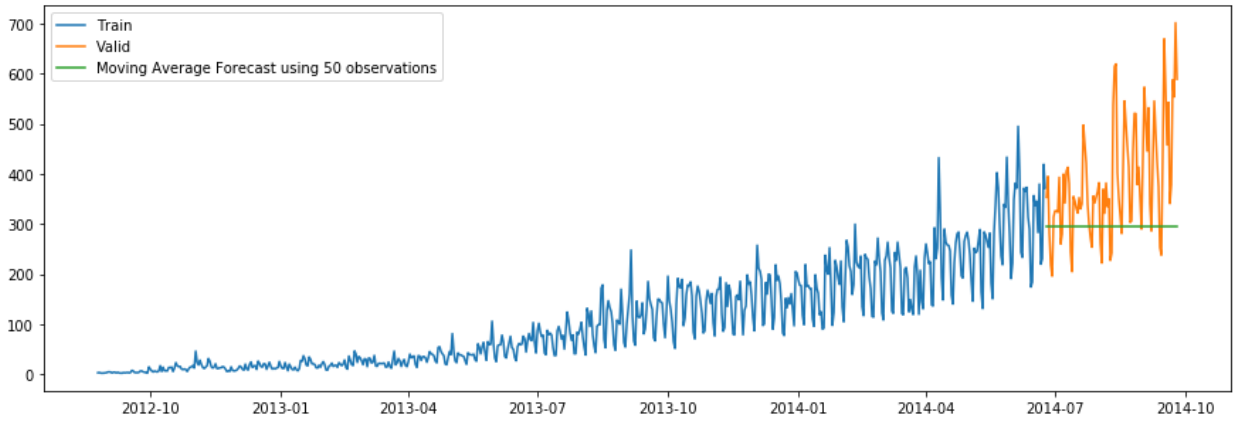
In [28]:

```
y_hat_avg = valid.copy()
y_hat_avg['moving_avg_forecast'] = Train['Count'].rolling(10).mean().iloc[-1] # averag
e 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 obs
ervations')

plt.legend(loc='best')

plt.show()

y_hat_avg = valid.copy()

y_hat_avg['moving_avg_forecast'] = Train['Count'].rolling(20).mean().iloc[-1] # averag
e 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 obs
ervations')

plt.legend(loc='best')

plt.show()

y_hat_avg = valid.copy()

y_hat_avg['moving_avg_forecast'] = Train['Count'].rolling(50).mean().iloc[-1] # averag
e 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 obs
ervations')

plt.legend(loc='best')

plt.show()
```







In [29]:

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

print(rms)
```

```
144.19175679986802
```

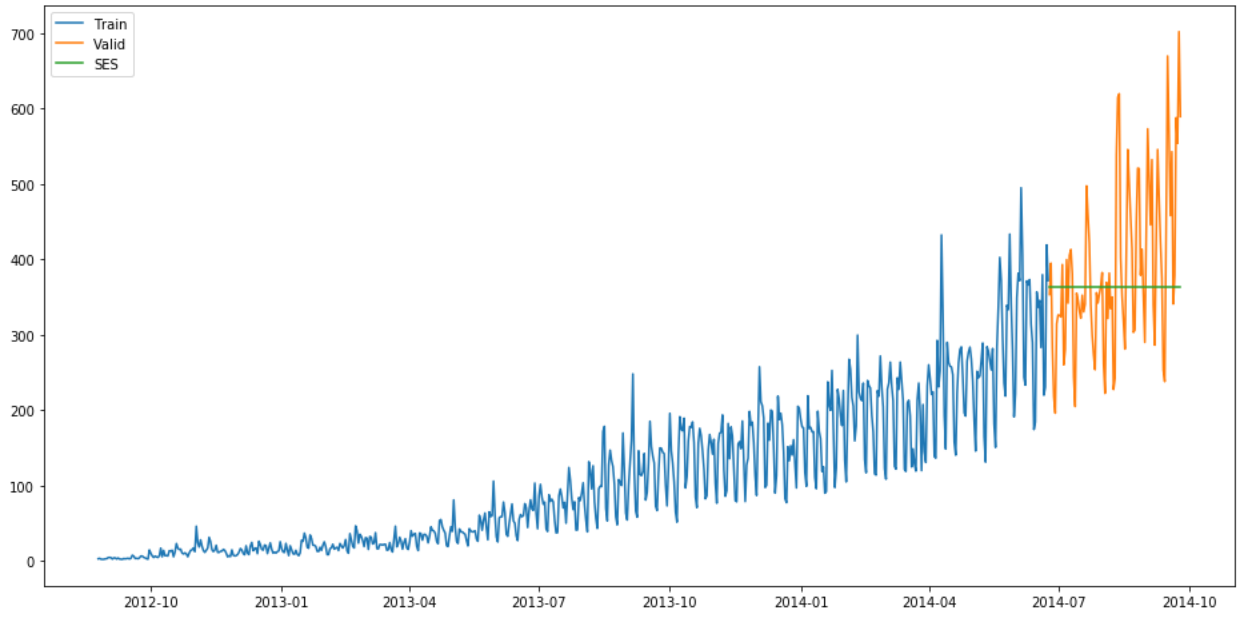
iii) Simple Exponential Smoothing¶

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.

In [30]:

```
from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing, Holt

y_hat_avg = valid.copy()

fit2 = SimpleExpSmoothing(np.asarray(Train['Count'])).fit(smoothing_level=0.6,optimize
d=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()
```



In [31]:

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

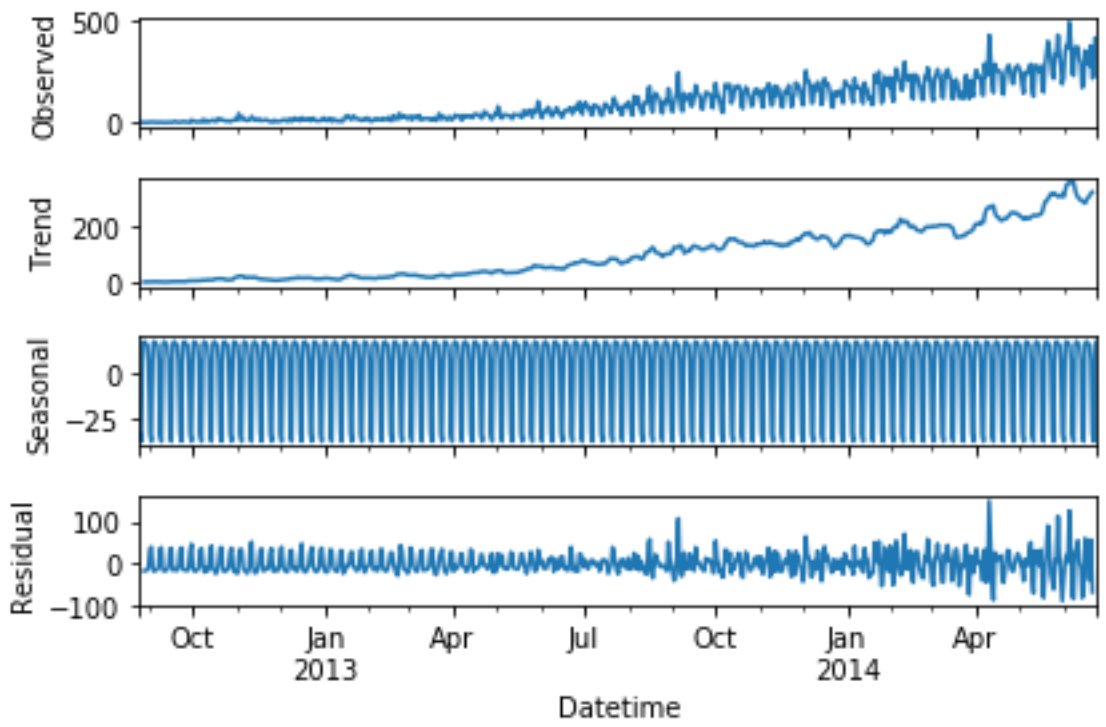
```
113.43708111884514
```

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.

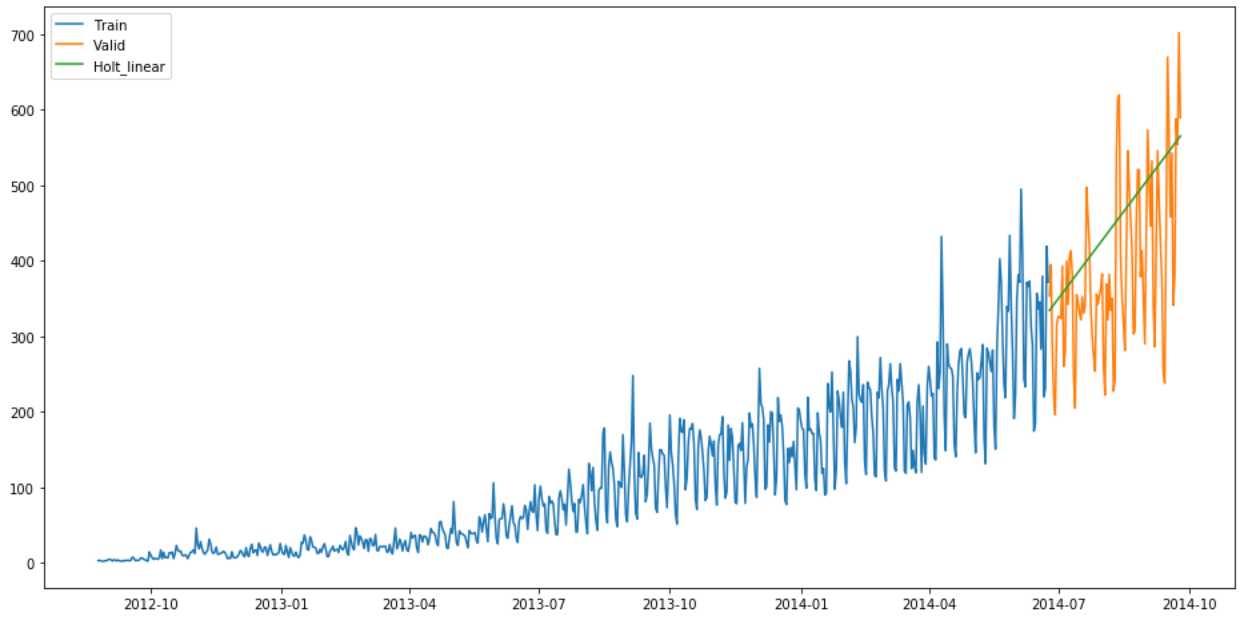
In [32]:

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



In [33]:

```
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()
```



In [34]:

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

```
112.94278345314041
```

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.

In [35]:

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

In [36]:

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

In [37]:

```
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.

In [38]:

```
# 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']
```

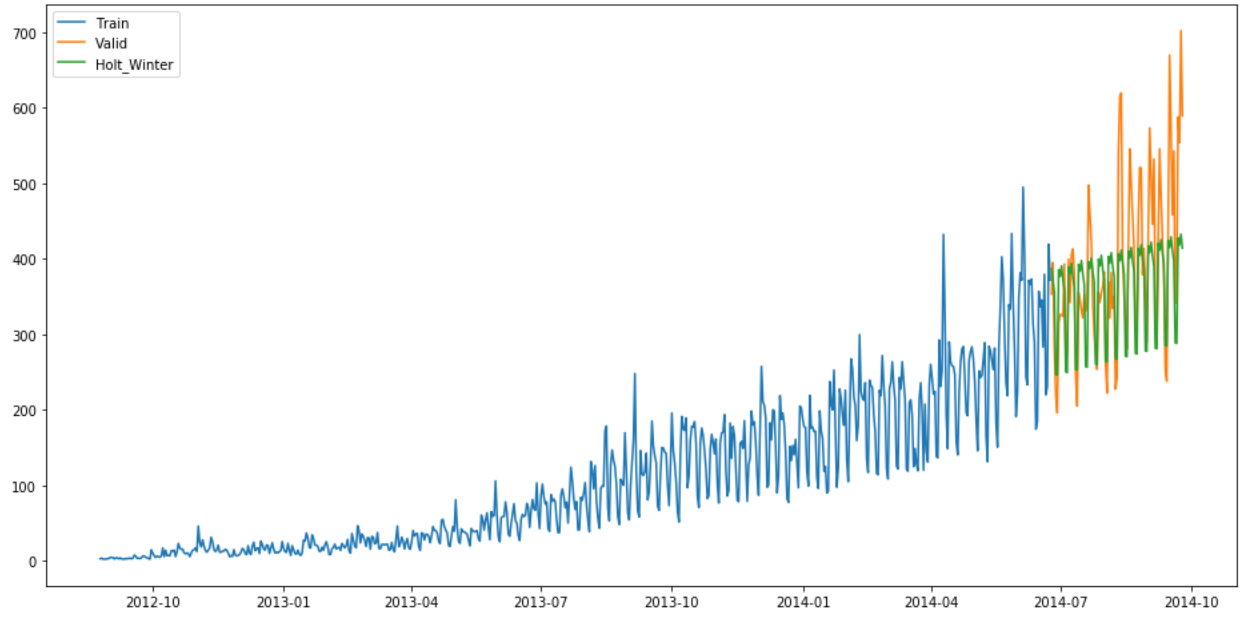
Holt winter's model on daily time series¶

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.

In [39]:

```
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()
```



In [40]:

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

```
82.37373991413227
```

In [41]:

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

In [42]:

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

In [43]:

```
# 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
```

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.

In [44]:

```
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:
    print ('Results of 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
```

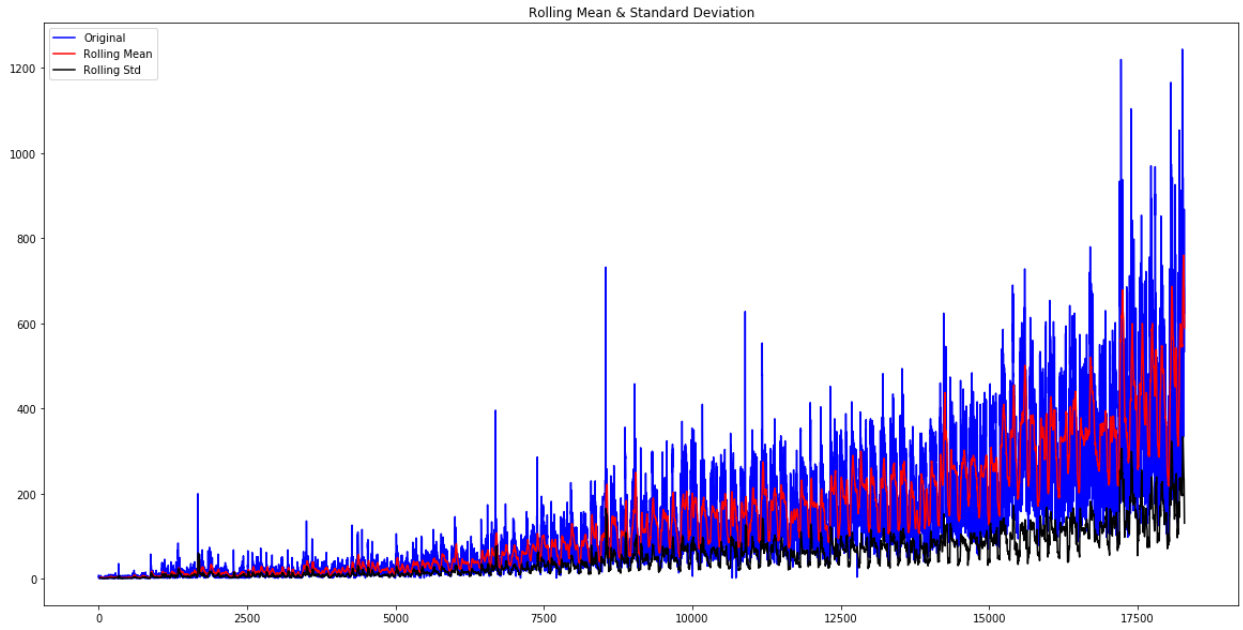
```
    print (dfoutput)
```

In [45]:

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

In [46]:

```
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
```

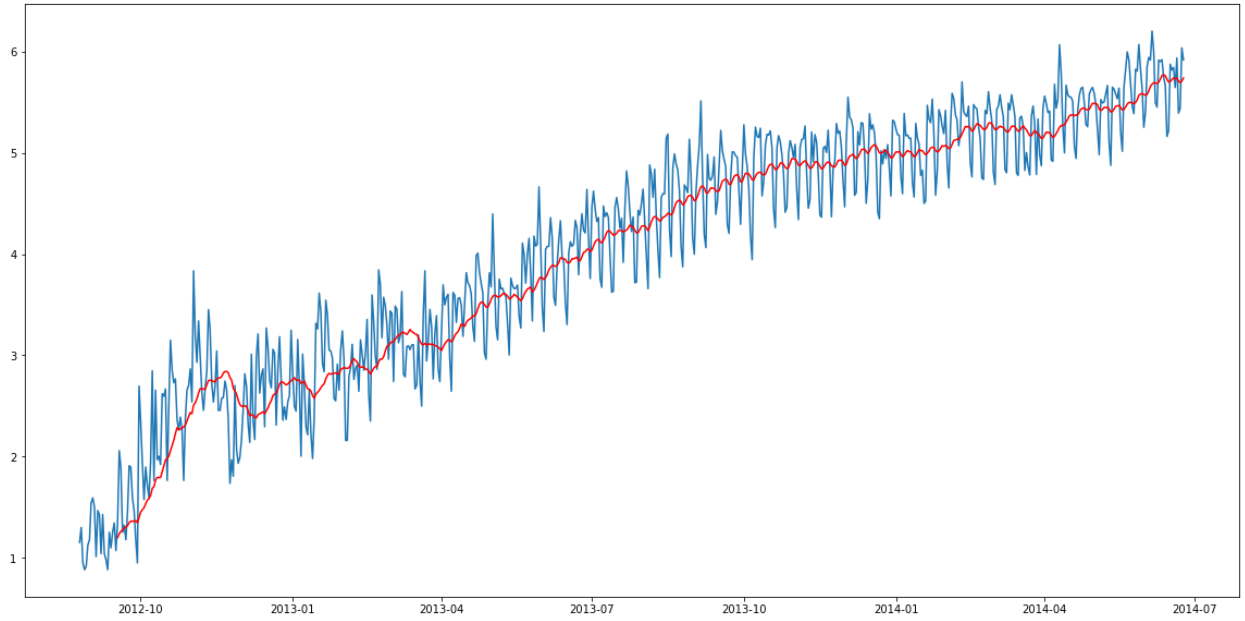
Removing Trend

In [47]:

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

In [48]:

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

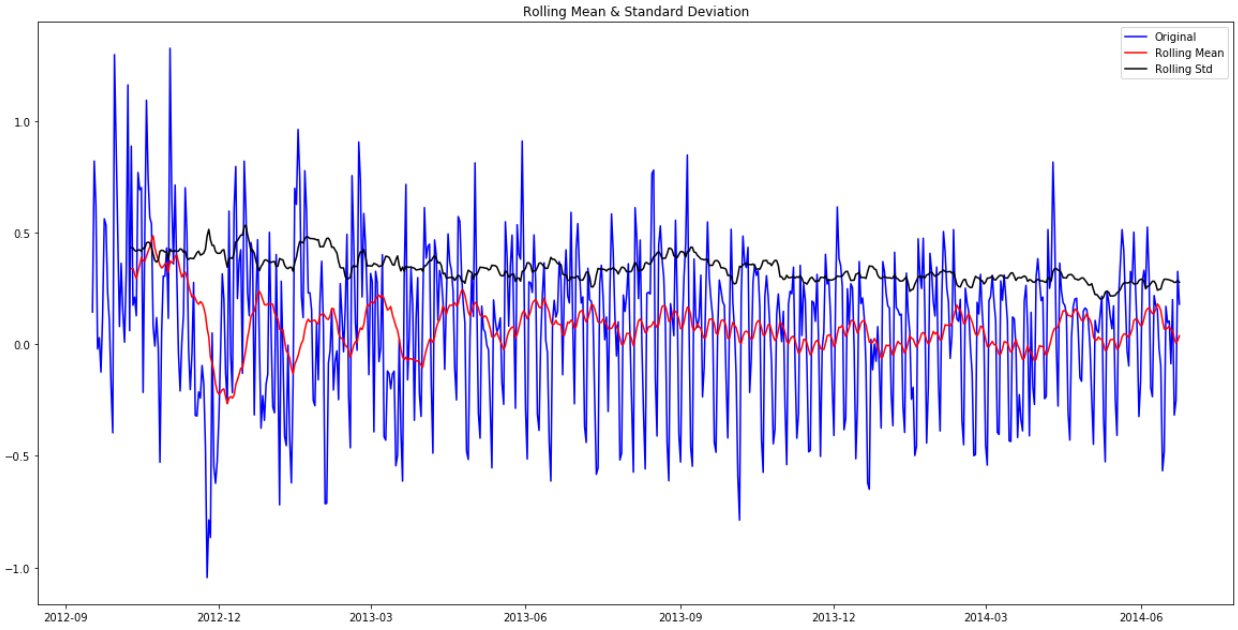


In [49]:

```
train_log_moving_avg_diff = Train_log - moving_avg
```

In [50]:

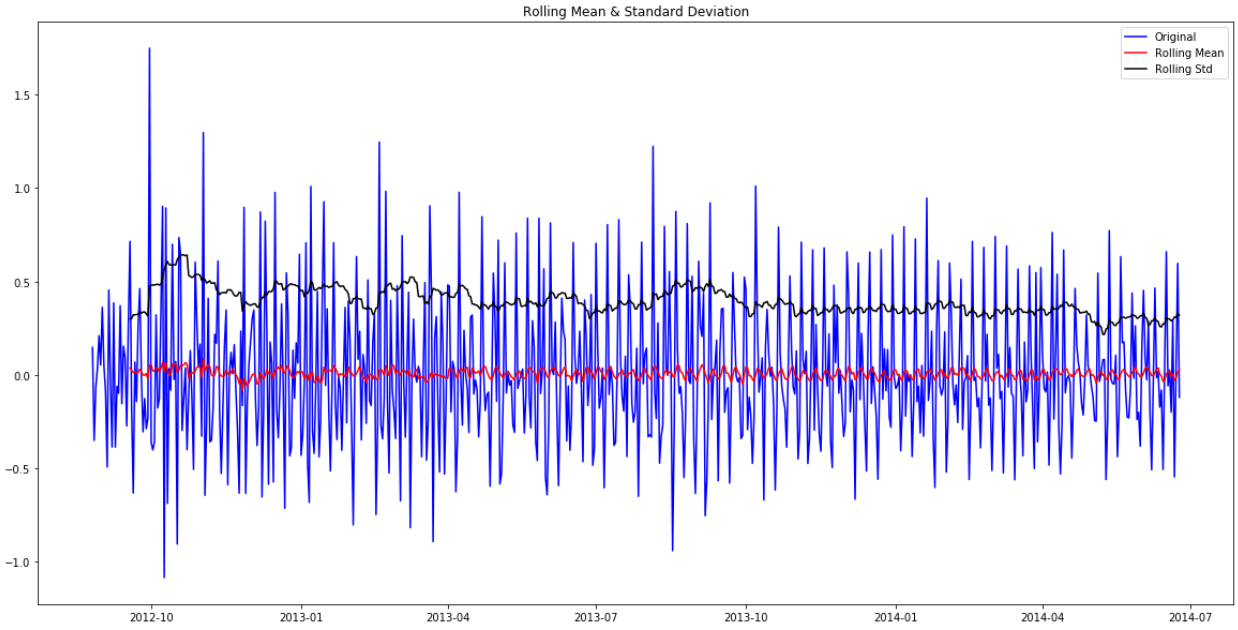
```
train_log_moving_avg_diff.dropna(inplace = True)
test_stationarity(train_log_moving_avg_diff)
```




```
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
```

In [51]:

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



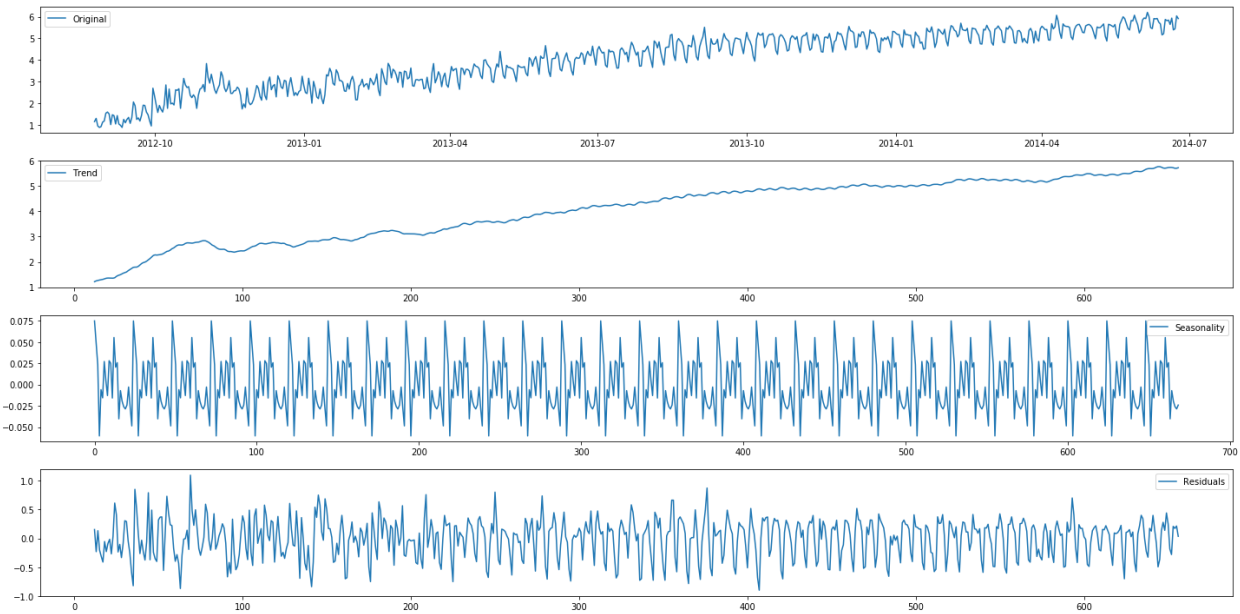

```
Results of Dickey-Fuller Test:

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
```

Removing Seasonality

In [52]:

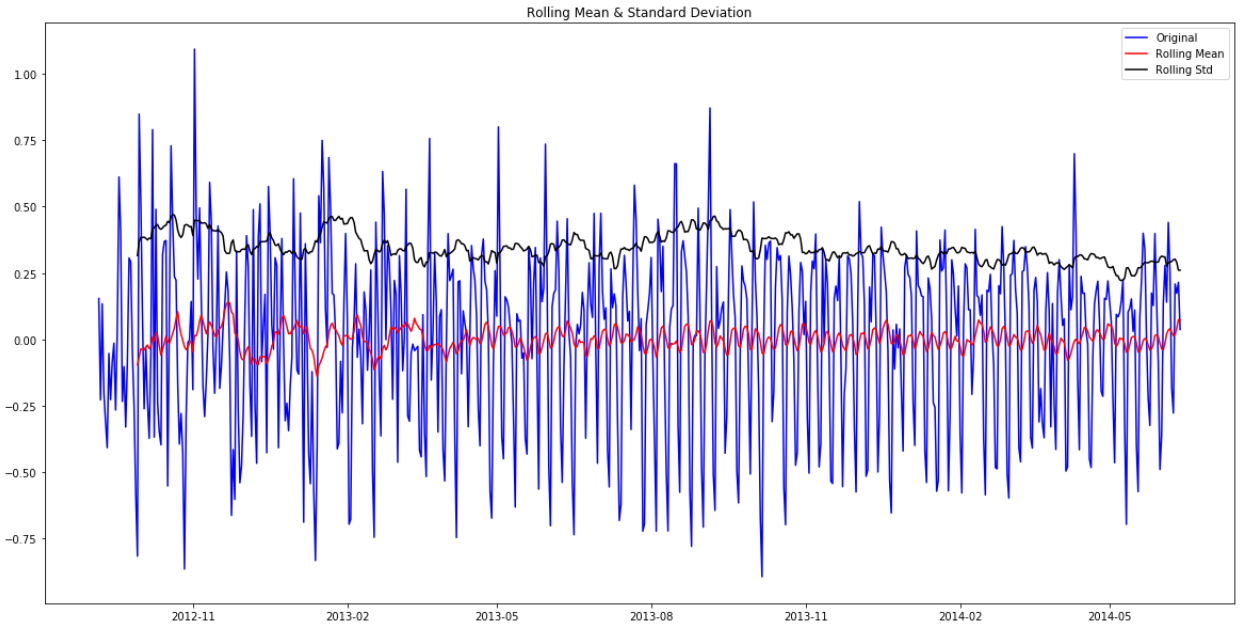
```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(pd.DataFrame(Train_log).Count.values, freq = 24)


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()
```



In [53]:

```
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
```

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.

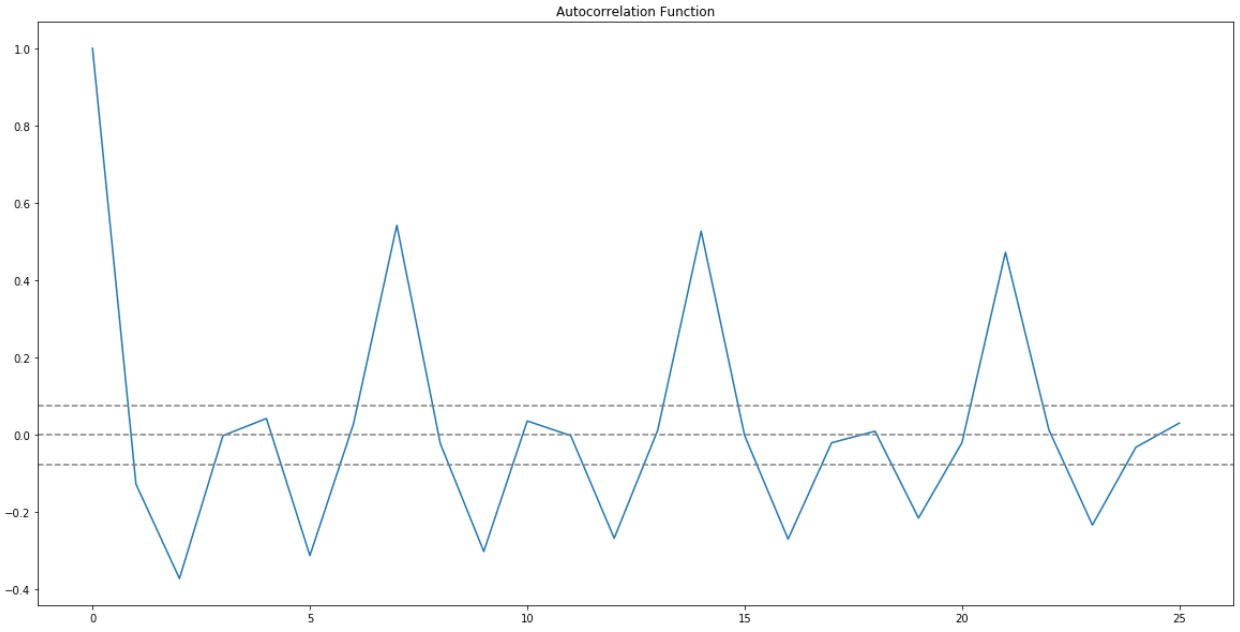
PACF measures the correlation between the TimeSeries with a lagged version of itself but after eliminating the variations already explained by the intervening comparisons.

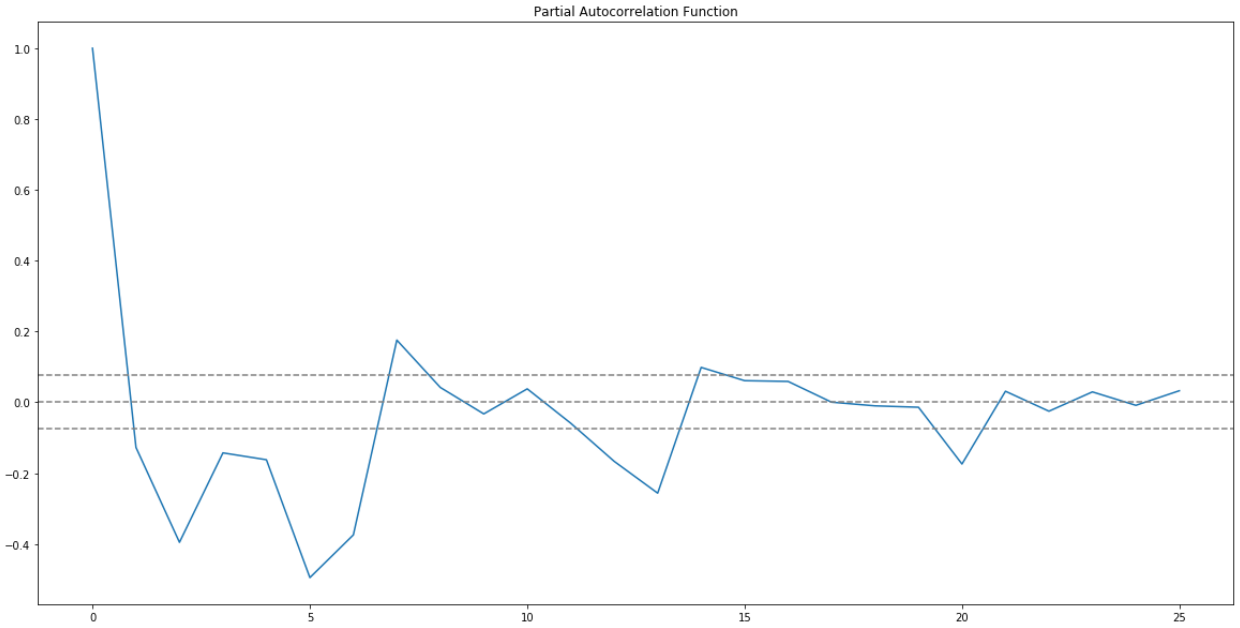
In [54]:

```
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')
```

In [55]:

```
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()
```

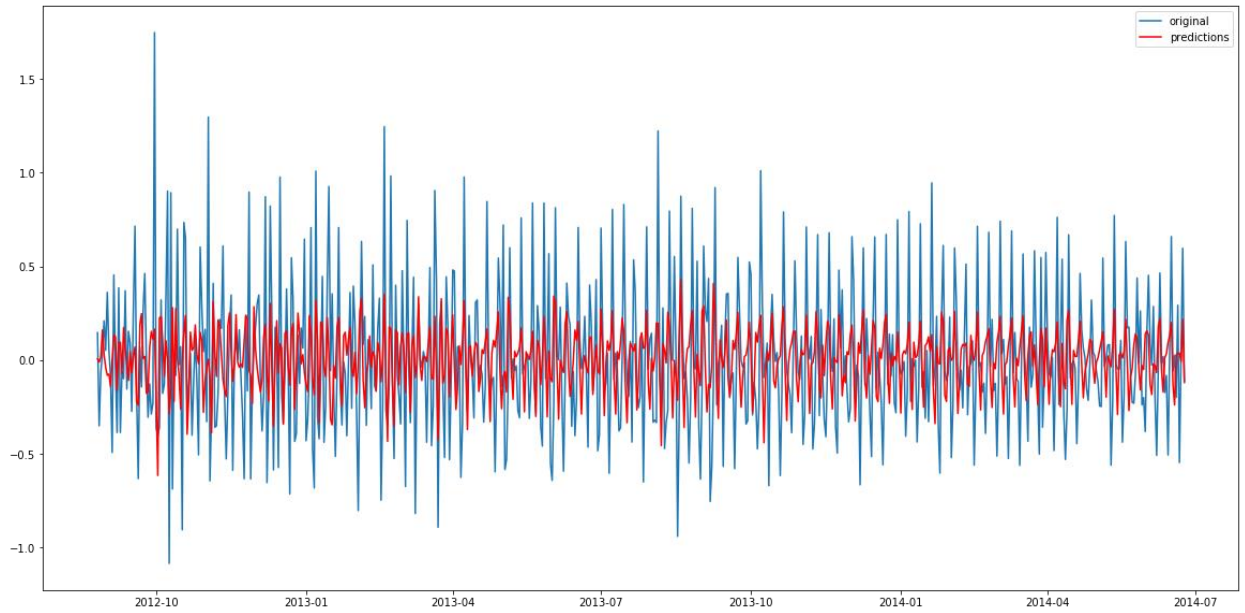




AR model¶

In [56]:

```
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()
```

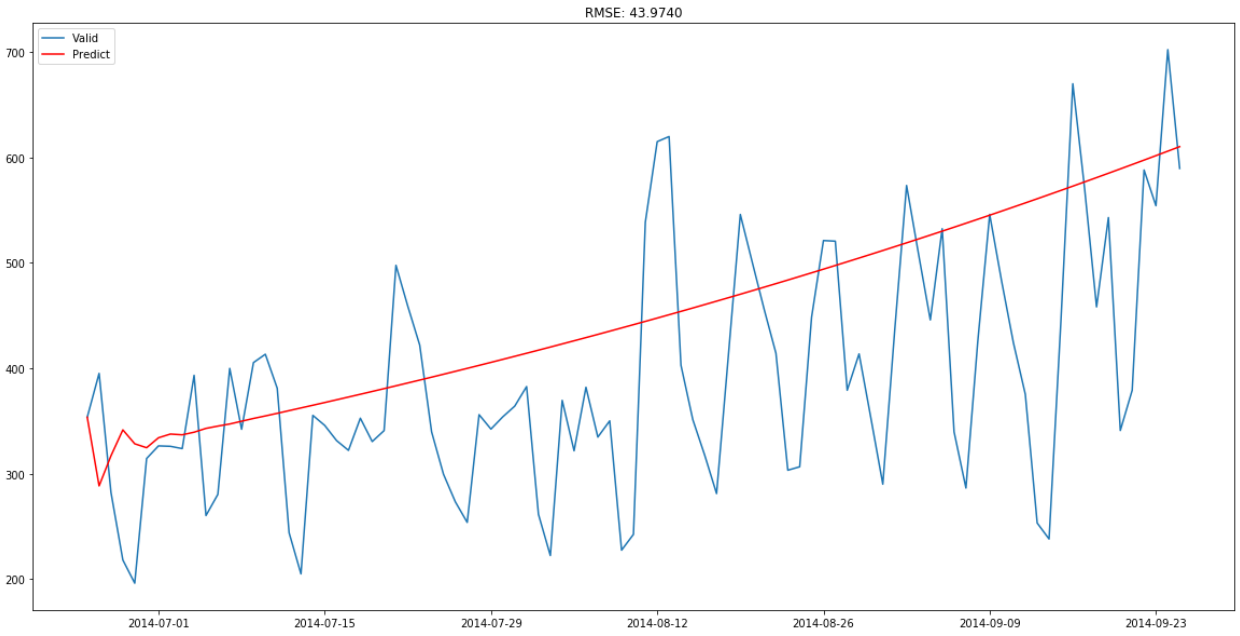


In [57]:

```
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 = val
id.index)

AR_predict1=AR_predict1.add(AR_predict,fill_value=0)

AR_predict = np.exp(AR_predict1)
```

In [58]:

```
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()
```

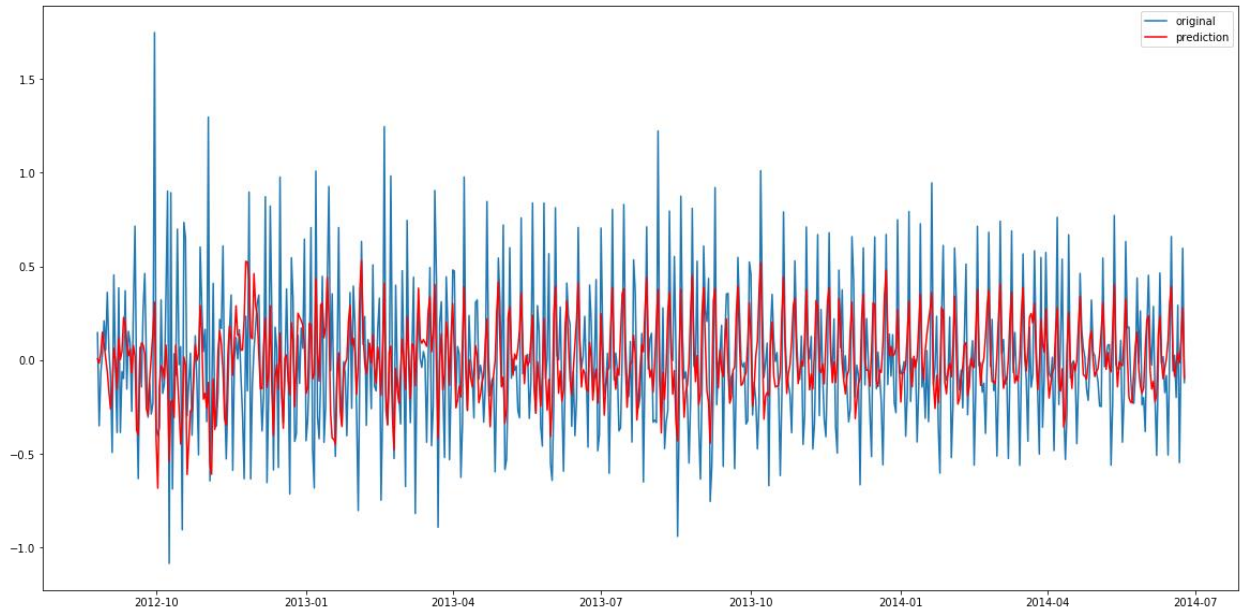


MA model¶

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

In [59]:

```
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()
```

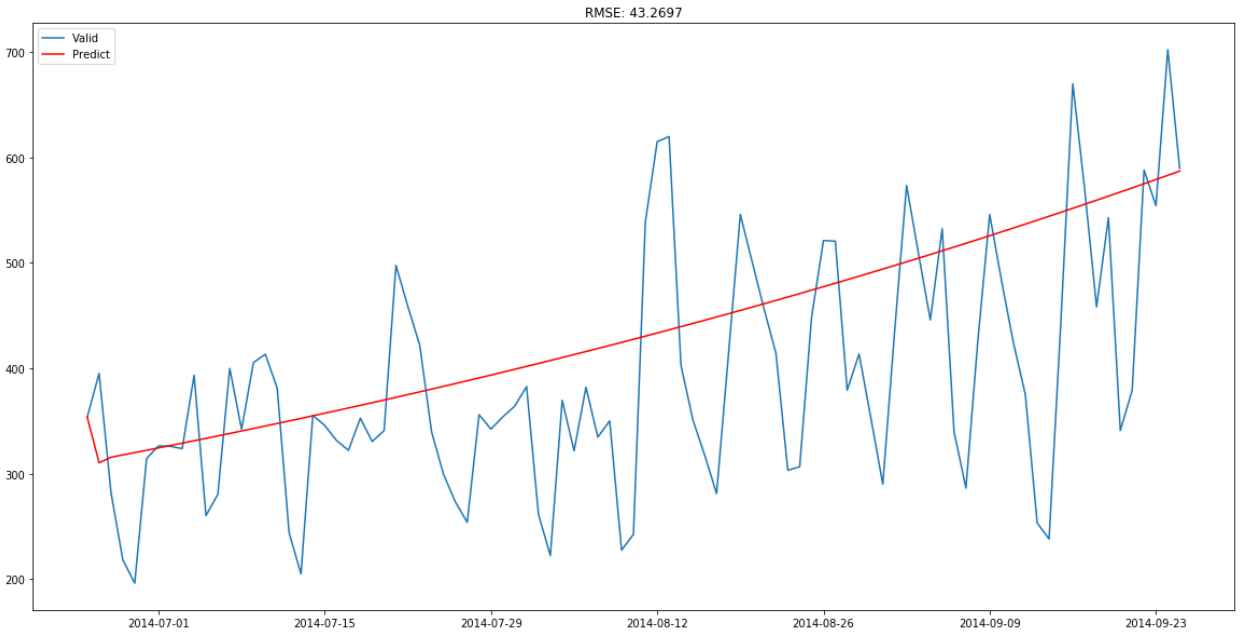


In [60]:

```
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 = val
id.index)

MA_predict1=MA_predict1.add(MA_predict,fill_value=0)

MA_predict = np.exp(MA_predict1)
```

In [61]:

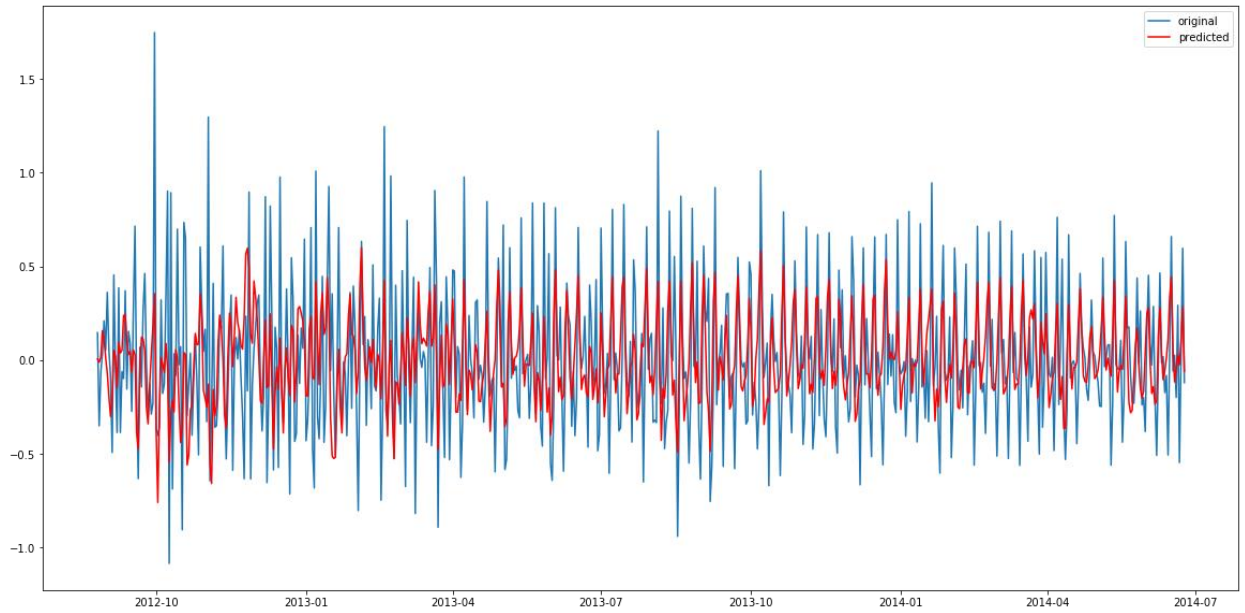
```
plt.plot(valid['Count'], label = "Valid")

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()
```



Combined model¶

In [62]:

```
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()
```



In [63]:

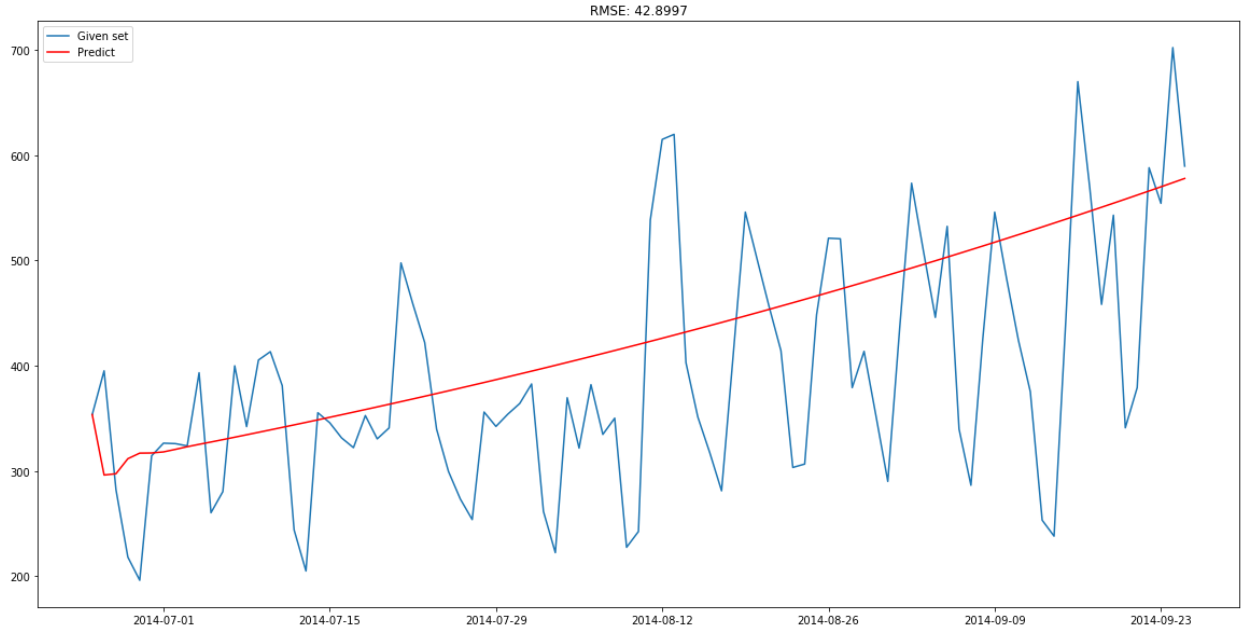
```
def check_prediction_diff(predict_diff, given_set):

    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.sh
ape[0]))

    plt.show()
```

In [64]:

```
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.sh
ape[0]))

    plt.show()
```

In [65]:

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

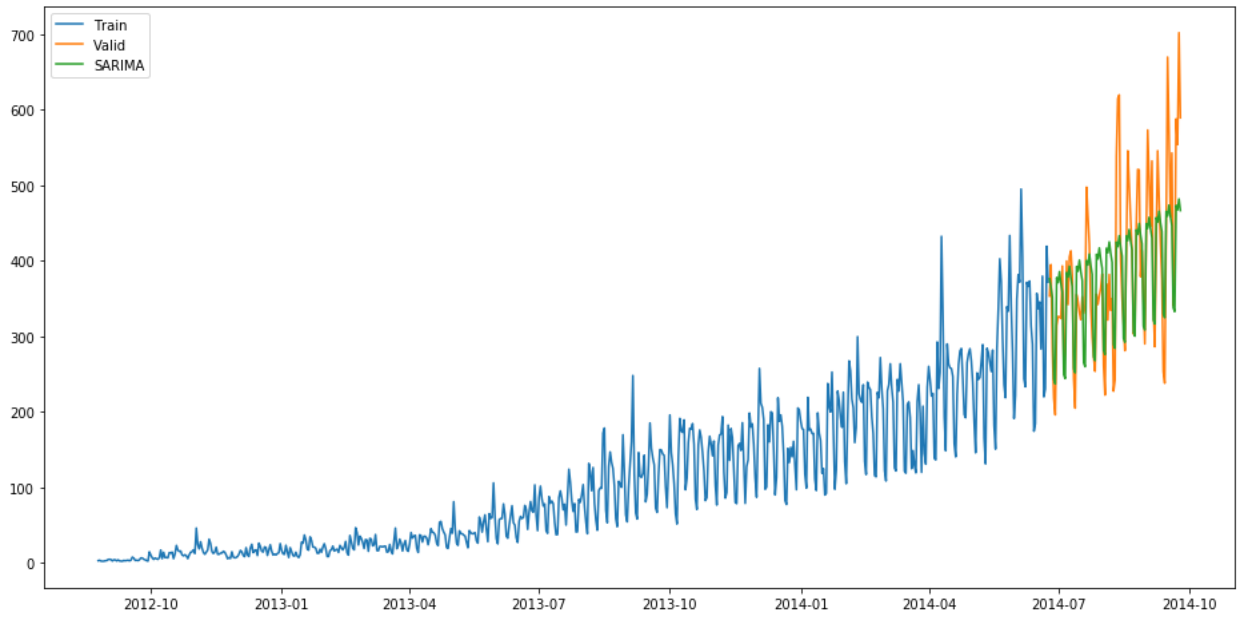


SARIMAX model on daily time series¶

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

In [66]:

```
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()
```



In [67]:

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

```
69.70093730473587
```

In [68]:

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

In [69]:

```
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
```

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.