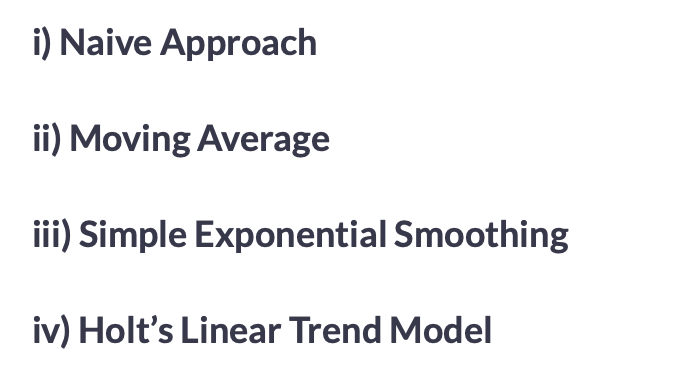
Some time series forecasting techniques:

­(pip install numpy --user –U)

if no pip3: (python3 -m pip)

e.g. python3 -m pip install numpy

rather than pip3 install numpy



**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:

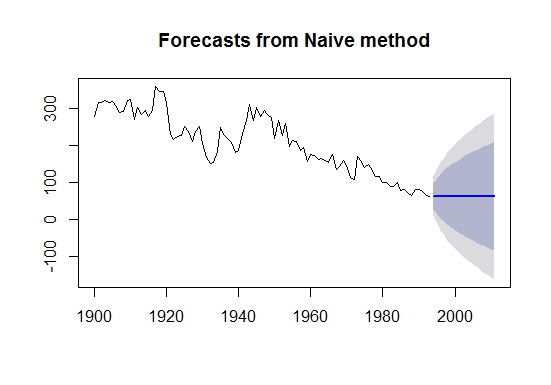
Suppose we have passenger count for 5 days as shown below:

|  |  |
| --- | --- |
| **Day** | **Passenger Count** |
| 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.

|  |  |
| --- | --- |
| **Day** | **Passenger Count** |
| 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')

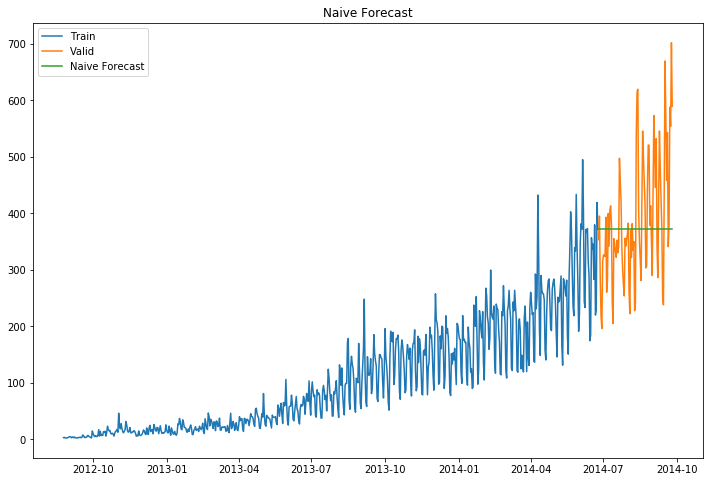
plt.plot(valid.index,valid['Count'], label='Valid')

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))

print(rms)

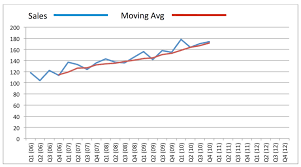
111.79050467496724

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))

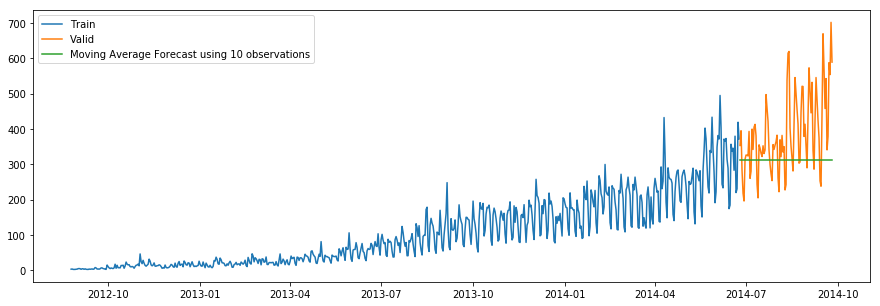
plt.plot(Train['Count'], label='Train')

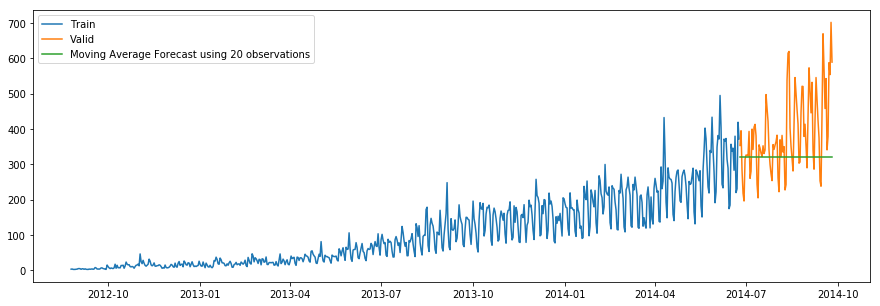
plt.plot(valid['Count'], label='Valid')

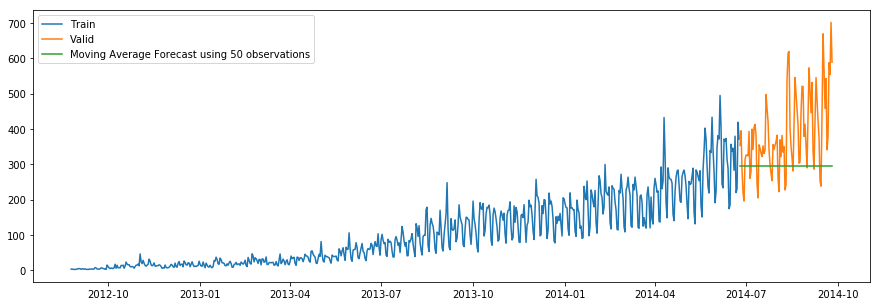
plt.plot(y\_hat\_avg['moving\_avg\_forecast'], label='Moving Average Forecast using 50 observations')

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))

print(rms)

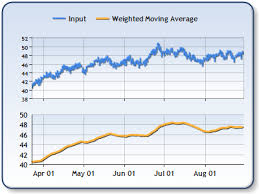
144.19175679986802

**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.

**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))

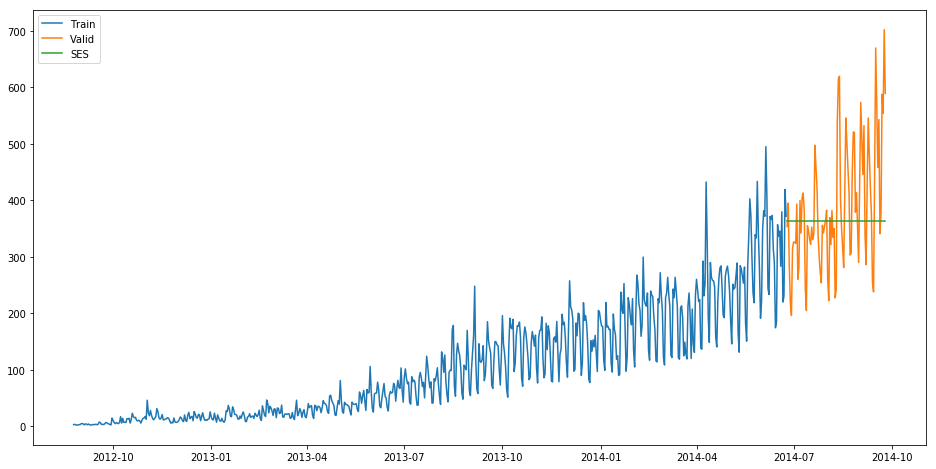
plt.plot(Train['Count'], label='Train')

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))

print(rms)

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.

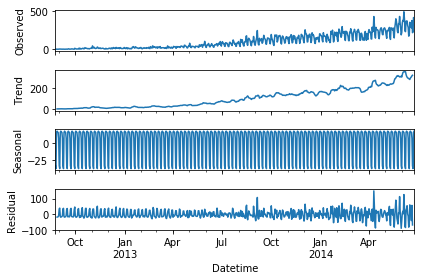
Lets visualize all these parts.

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))

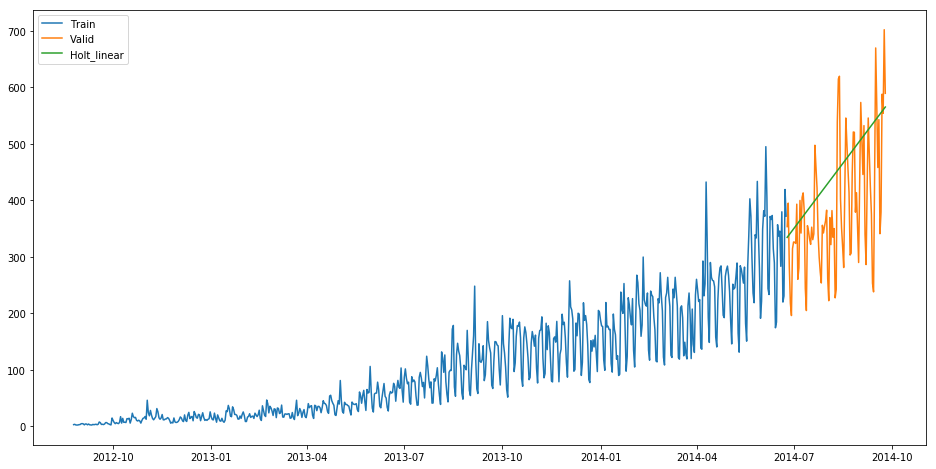
plt.plot(Train['Count'], label='Train')

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))

print(rms)

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.

**COMPLETE & CONTINUE**

**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.

**COMPLETE & CONTINUE**

**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))

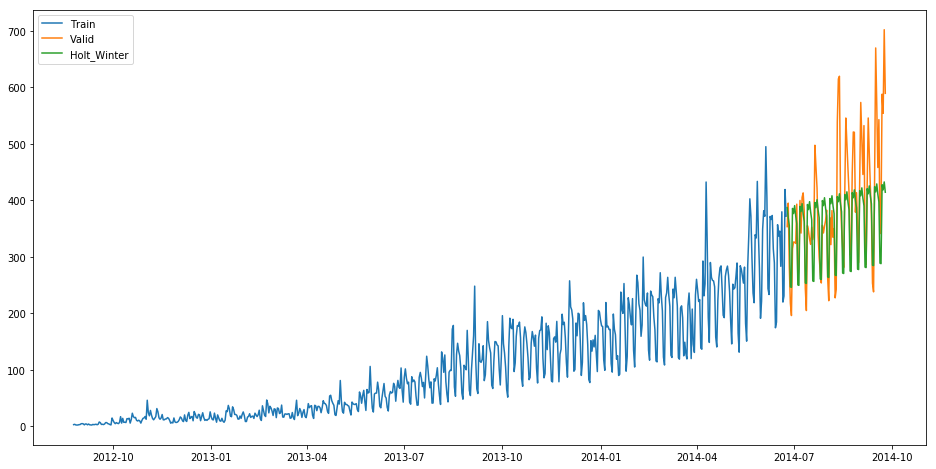
plt.plot( Train['Count'], label='Train')

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.

**COMPLETE & CONTINUE**

## 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 :

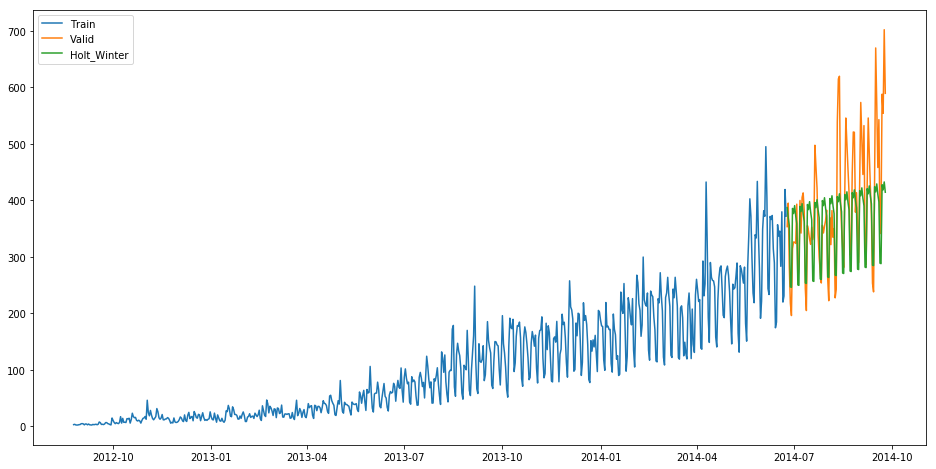
* The mean of the time series should not be a function of time. It should be constant.
* The variance of the time series should not be a function of time.
* THe covariance of the ith term and the (i+m)th term should not be a function of time.

### 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/>

**COMPLETE & CONTINUE**



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.

**COMPLETE & CONTINUE**

**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:

    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():

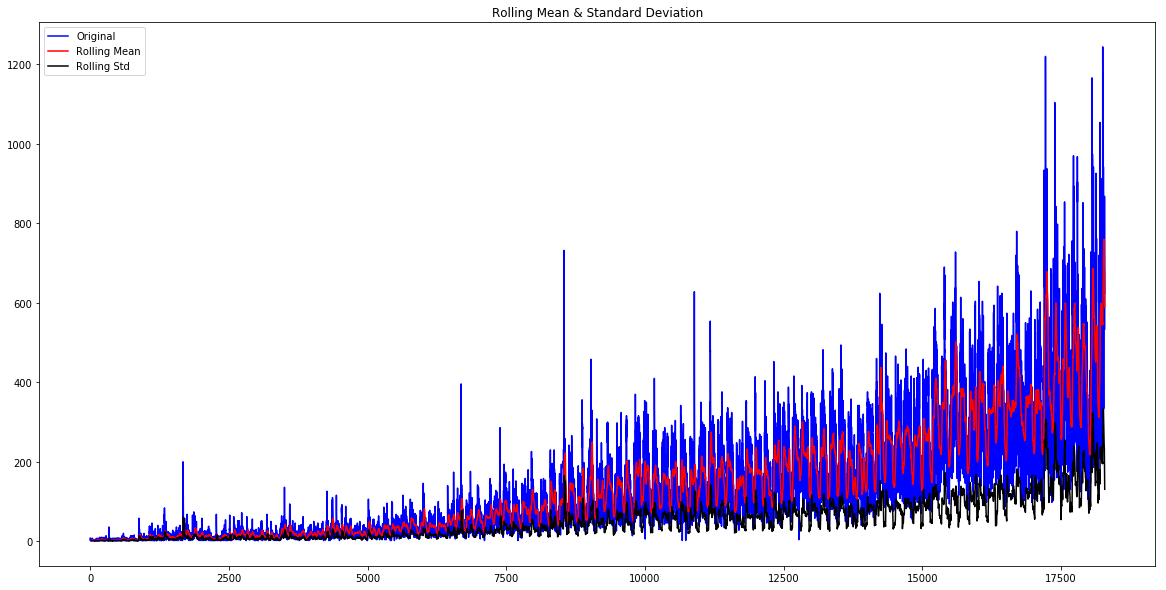
        dfoutput['Critical Value (%s)'%key] = value

    print (dfoutput)

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'])

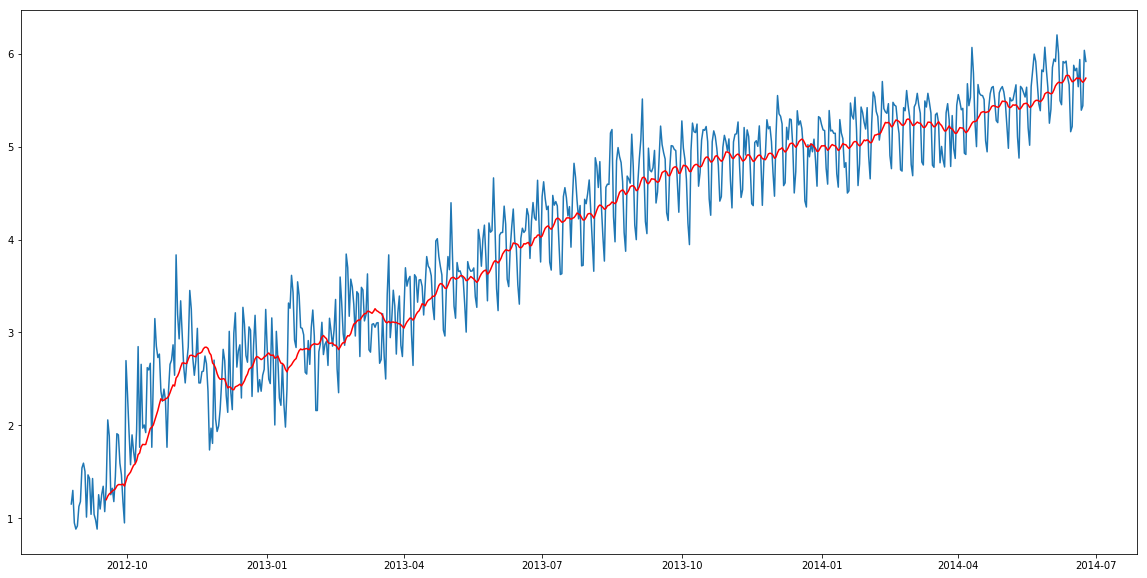
valid\_log = np.log(valid['Count'])

moving\_avg = pd.rolling\_mean(Train\_log, 24)

plt.plot(Train\_log)

plt.plot(moving\_avg, color = 'red')

plt.show()

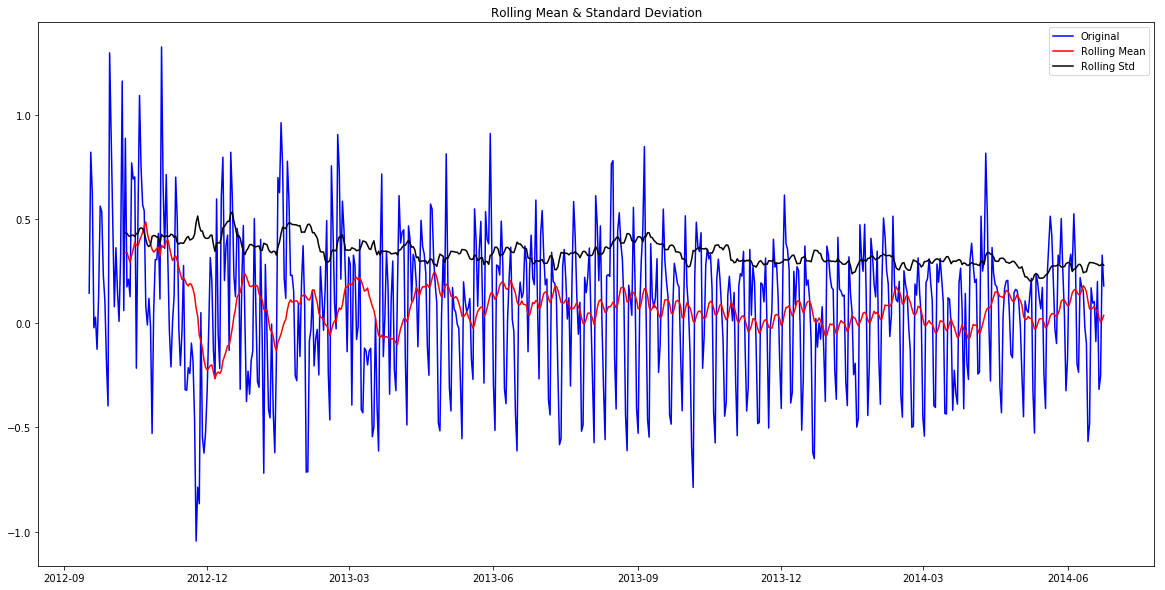


So we can observe an increasing trend. Now we will remove this increasing trend to make our time series stationary.

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

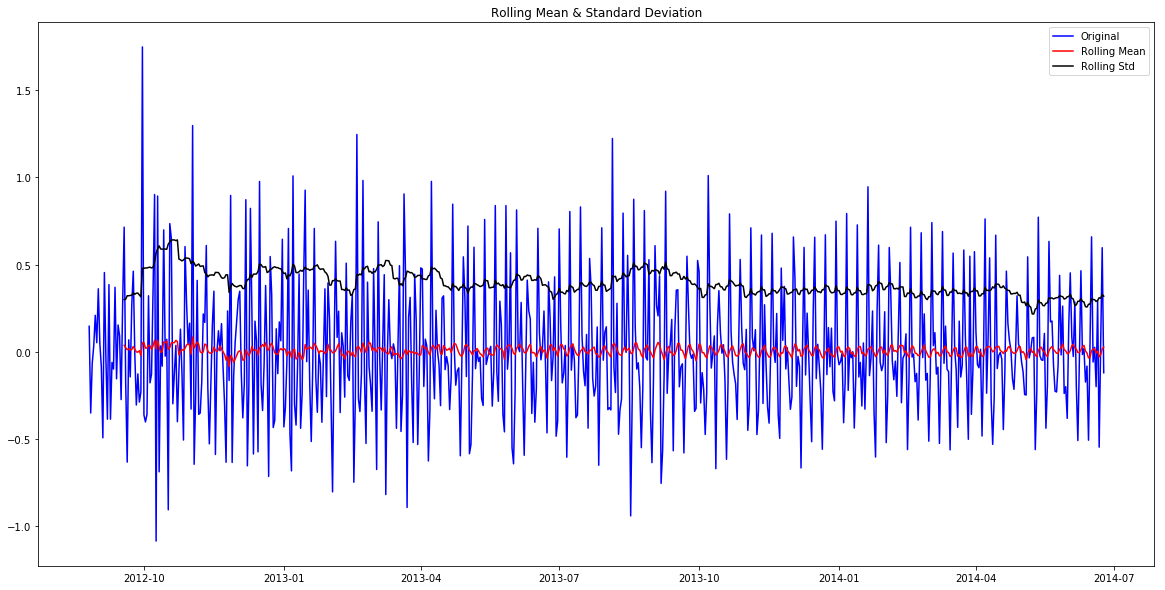
We can see that the Test Statistic is very smaller as compared to the Critical Value. So, we can be confident that the trend is almost removed.

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')

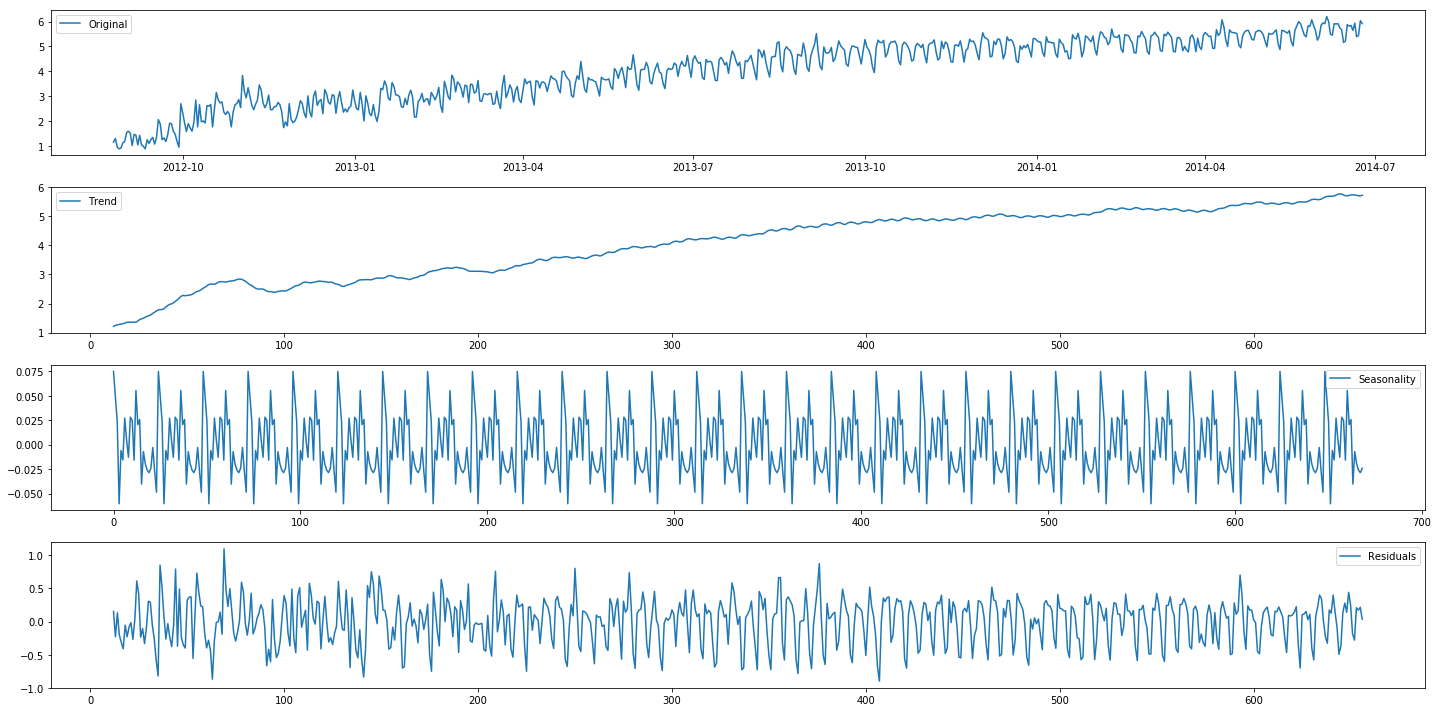
plt.subplot(414)

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.

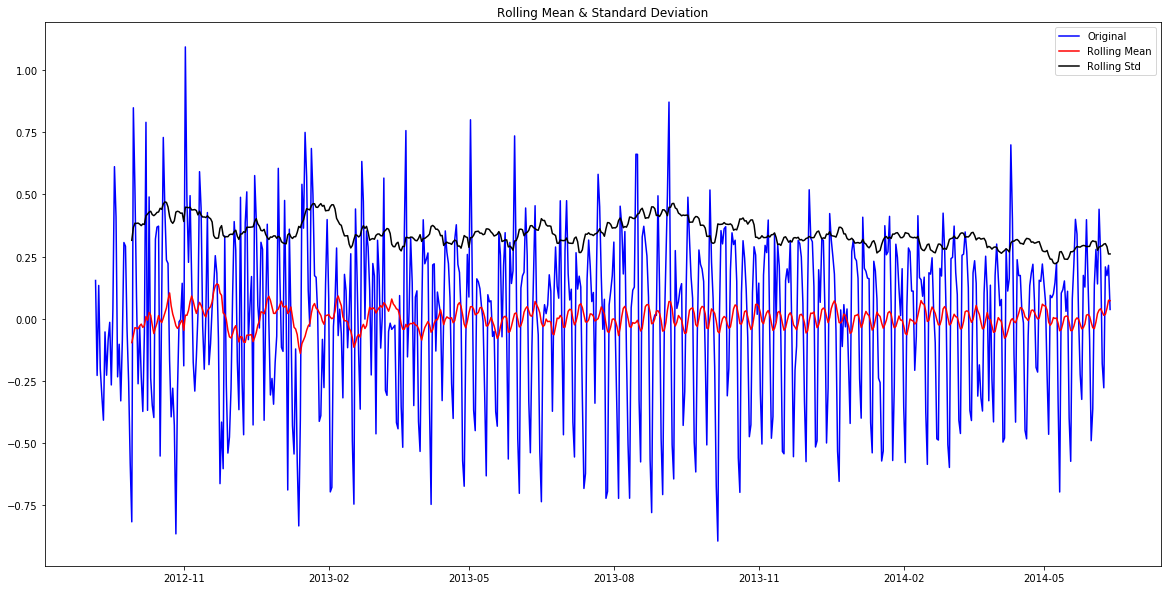
Let’s check stationarity of residuals.

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)

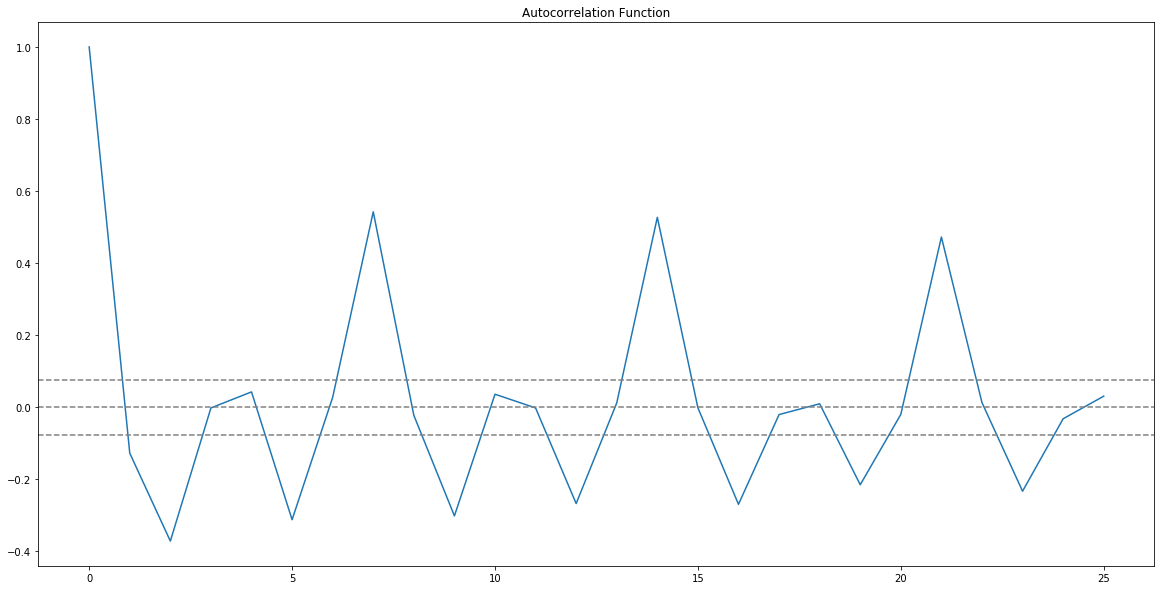
plt.axhline(y=0,linestyle='--',color='gray') plt.axhline(y=-1.96/np.sqrt(len(train\_log\_diff.dropna())),linestyle='--',color='gray') plt.axhline(y=1.96/np.sqrt(len(train\_log\_diff.dropna())),linestyle='--',color='gray') plt.title('Autocorrelation Function')

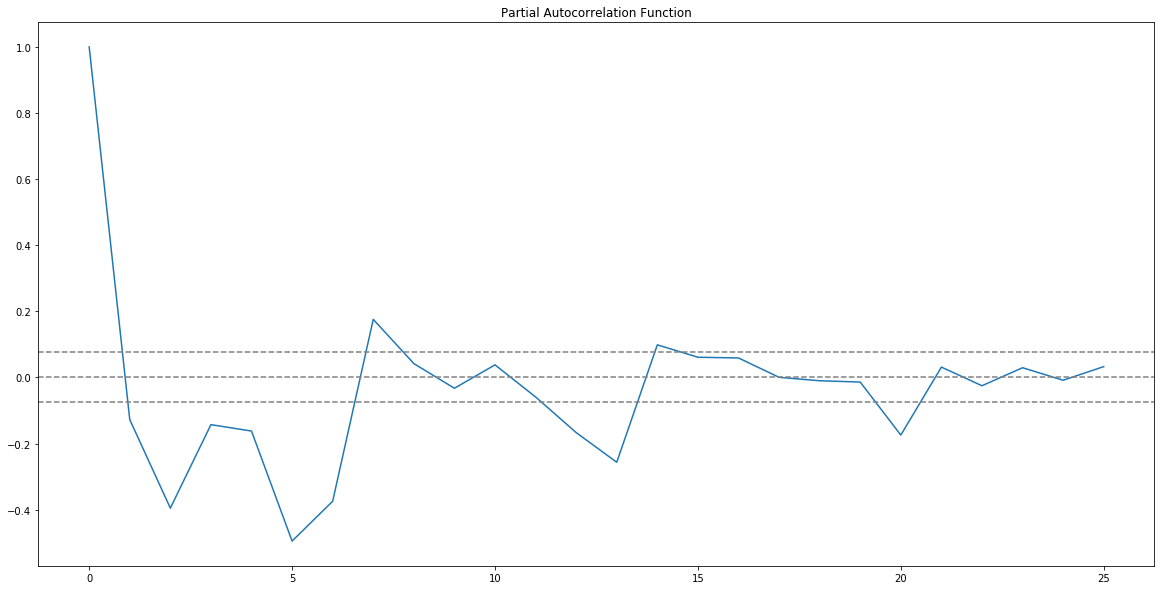
plt.show()

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

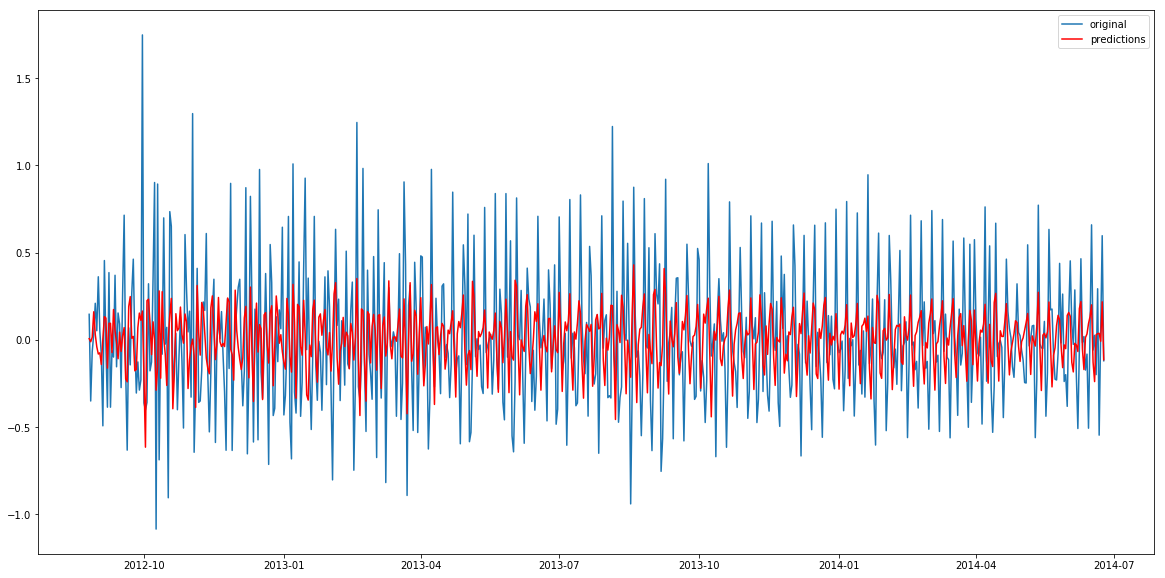
results\_AR = model.fit(disp=-1)

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)

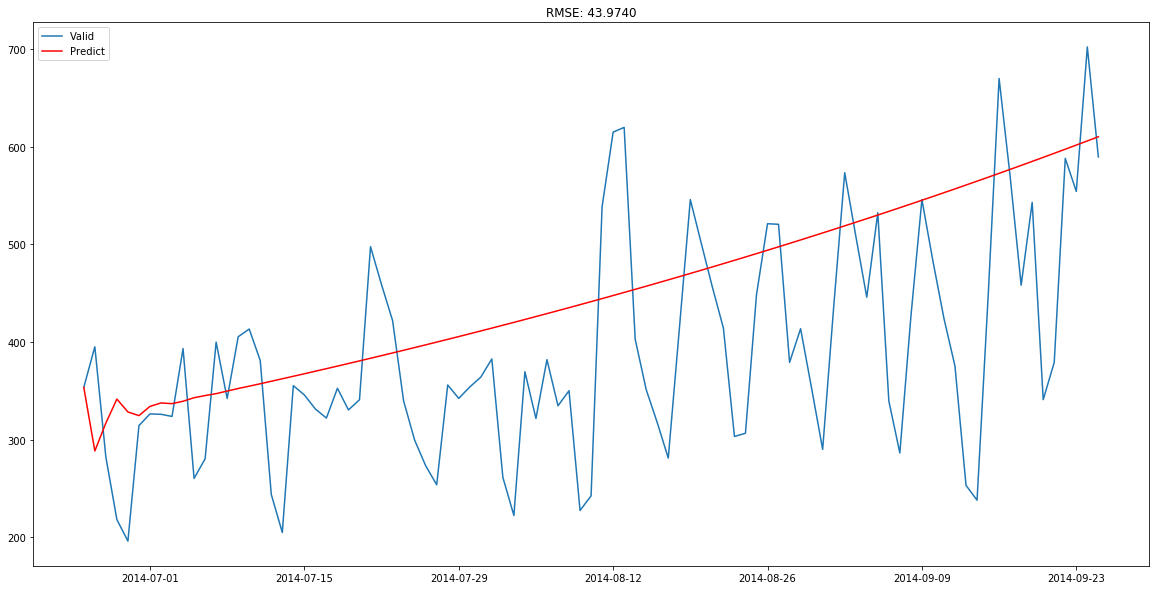
AR\_predict = np.exp(AR\_predict1)

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

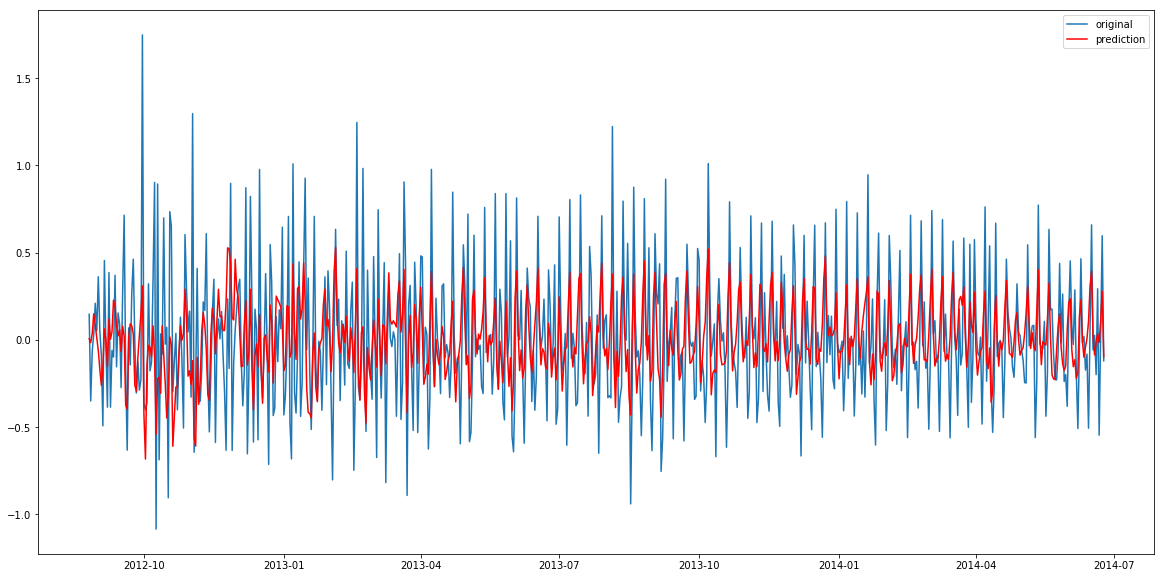
results\_MA = model.fit(disp=-1)

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)

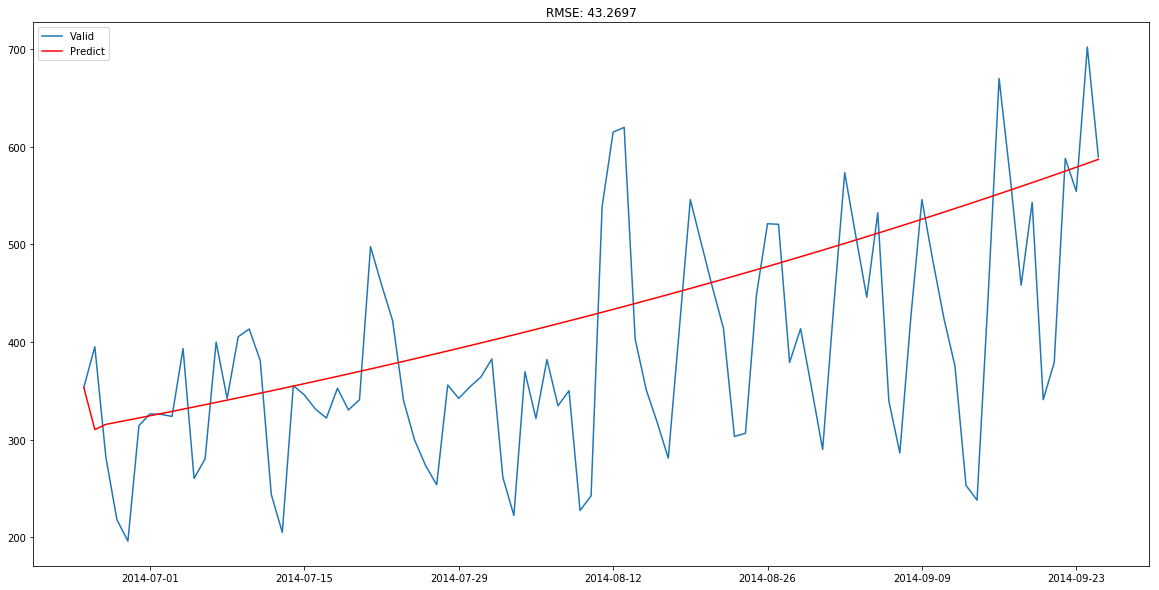
MA\_predict = np.exp(MA\_predict1)

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))

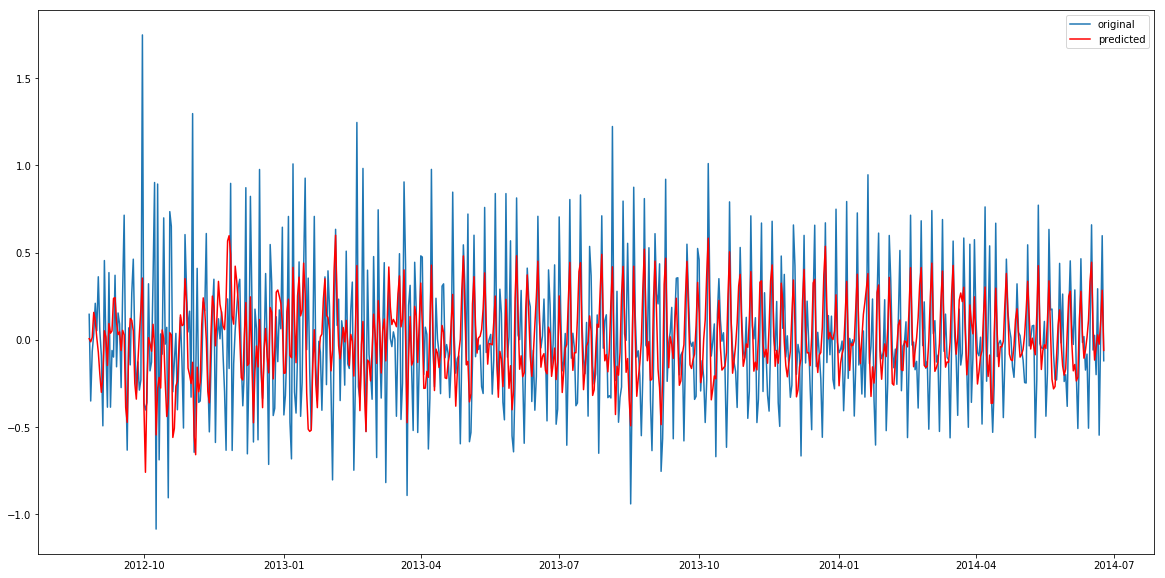
results\_ARIMA = model.fit(disp=-1)

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')

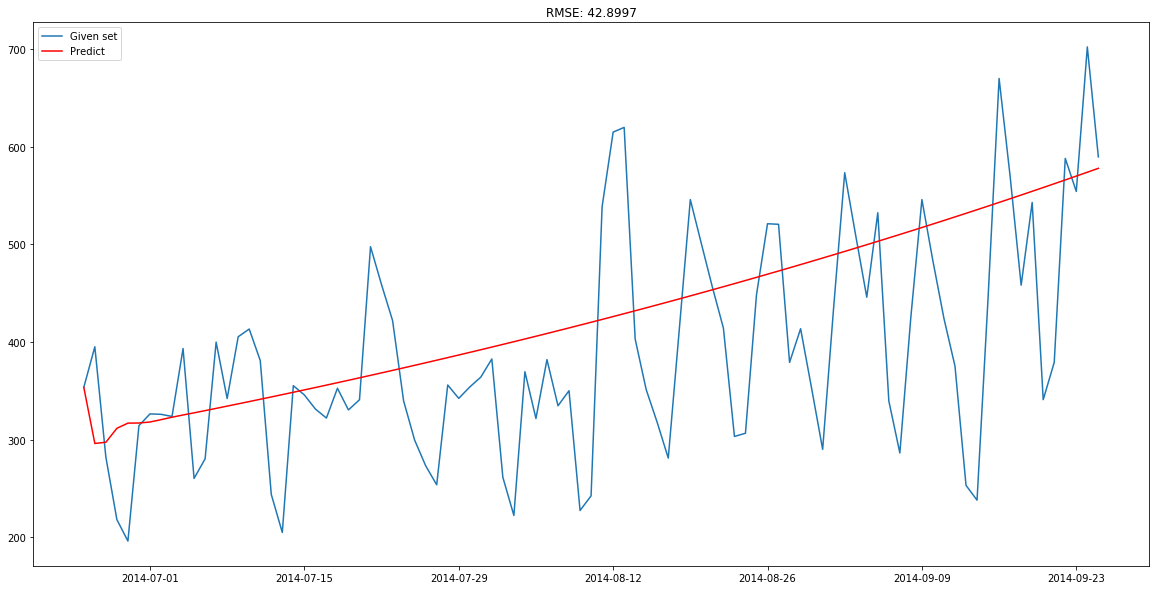
    plt.title('RMSE: %.4f'% (np.sqrt(np.dot(predict, given\_set['Count']))/given\_set.shape[0]))

    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)



**COMPLETE & CONTINUE**

**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')

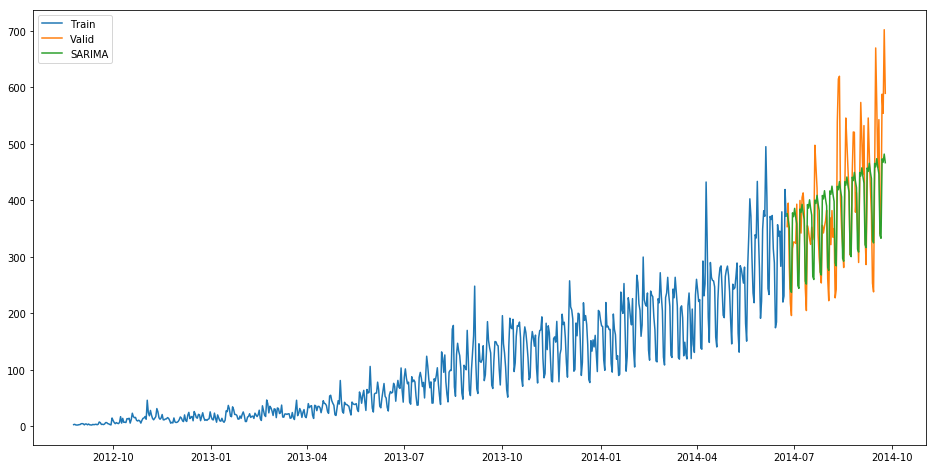
plt.plot(y\_hat\_avg['SARIMA'], label='SARIMA')

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))

print(rms)

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/>

**COMPLETE & CONTINUE**