Taxi Demand Forecasting Report

April 1, 2019

note: We attach the code along with the response, please feel free to only read the response.

1 Dataset

1.1 Data preprocessing and exploratory analysis

We are provided the dataset of list of timestamps recording taxi requests in a certain region from 2010-1-1 to 2010-8-28. (We intend to hide the dataset in this report.)

We import data using pandas, aggregate number of logins by 15 mins intervals, get time series named ts. It uses time intervals as index and number of logins as value.

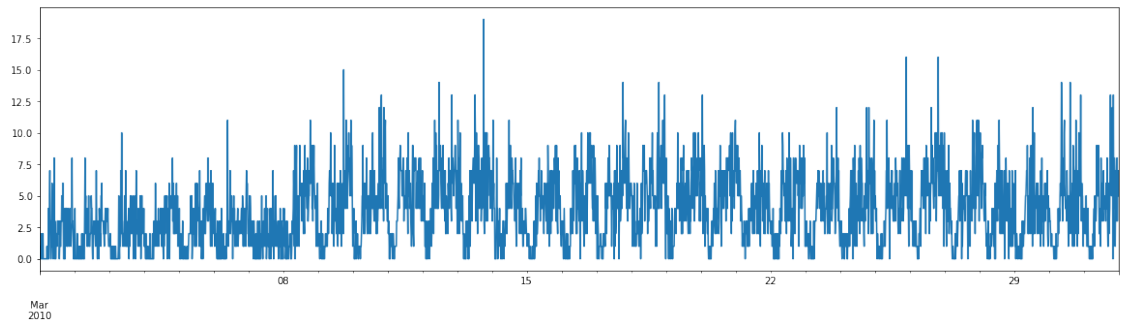
```
In [1]: #import data, aggregate number of logins by 15 mins intervals, get time series ts
        import pandas as pd
        import json
        import datetime
        import traces
        import matplotlib.pyplot as plt
        df=pd.io.json.read_json(url,typ='series',date_unit='s')
        dts=pd.Series(1.,index=pd.to_datetime(df['login_time']))
        ts=dts.resample('15T').sum()
```

```
In [2]: #for example in the last hour ts looks like this, there are 4,4,4,2 logins
        ts['2010-8-28 14:00:00':]
```

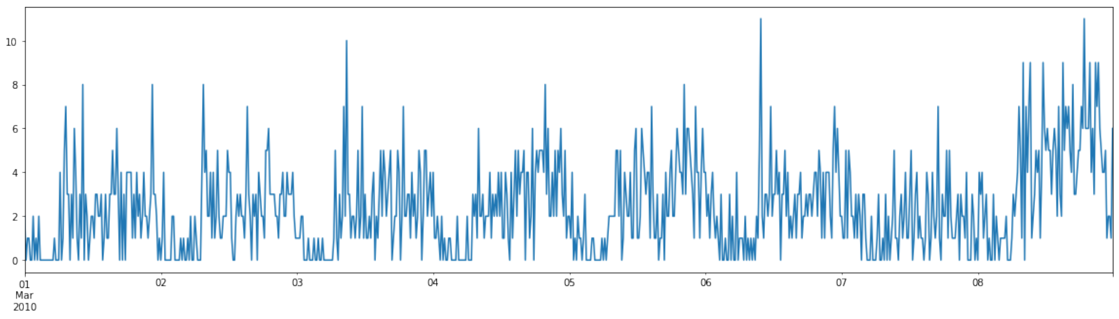
```
Out[2]: 2010-08-28 14:00:00    4.0
        2010-08-28 14:15:00    4.0
        2010-08-28 14:30:00    4.0
        2010-08-28 14:45:00    2.0
        Freq: 15T, dtype: float64
```

```
In [3]: #for example in March ts looks like this, it strongly suggests each day is a cycle, each
        plt.figure(figsize=(20, 5))
        ts['2010-03-01':'2010-03-31'].plot()
```

```
Out[3]: <matplotlib.axes._subplots.AxesSubplot at 0x123085828>
```



```
In [34]: plt.figure(figsize=(20, 5))
         ts['2010-03-01':'2010-03-8'].plot()

Out[34]: <matplotlib.axes._subplots.AxesSubplot at 0x1c2aa0b4a8>
```



The plot strongly suggests a daily cycle and even a weekly cycle. But due to the stochastic behavior of the raw data, there are too many ups and downs, so we conduct a smoothing in the next step.

1.2 Stationary test and smoothing

After testing the series is stationary, we apply a rolling mean on the raw data with a 12 hours window and call it cts (continuous time series) because the behavior in a half-day is relatively small to the overall pattern. Now the daily cycle is better revealed as in [37].

At 00:00 there is some amount of demand, from people who work at late night or in night events. Later, the demand is decreasing as people go home. Around 4 am, most activities are finished and people go to sleep, so there is a minimum demand. (I have personally experienced no Uber when I need a ride for my 7 am flight at 5 am.) Since then, a new day begins and the request increases. It stays at a relatively high volume during day time all the way to 24:00.

The specific behavior during a day largely varies because of events, weather, holiday, which is hidden behind the data. But it also depends on the day of the week explicitly. From plot [3] we can tell Monday and Sunday have the lowest demand, while the days in between are higher.

One more interesting observation is the demand in the first week of February and March, the last 3 days of May, first 5 days of July and first 5 days of August are unusually low, which we aren't able to explain yet.

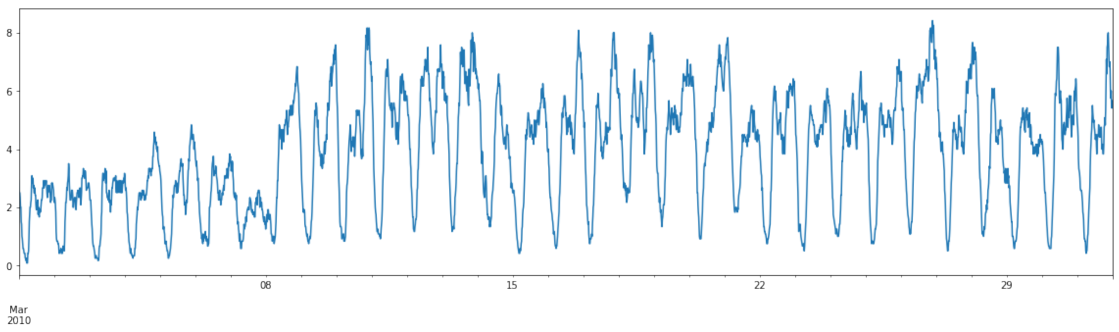
```
In [4]: #test stationary of ts
        from statsmodels.tsa.stattools import adfuller
        dftest=adfuller(ts)
        dfoutput = pd.Series(dftest[0:4], index=['Test Statistic',
                                                'p-value','#Lags Used','Number of Observations
        for key,value in dftest[4].items():
                dfoutput['Critical Value (%s)'%key] = value
        print(dfoutput)
```

```
Test Statistic                   -21.467619
p-value                            0.000000
#Lags Used                        47.000000
Number of Observations Used    22956.000000
Critical Value (1%)               -3.430635
Critical Value (5%)               -2.861666
Critical Value (10%)              -2.566837
dtype: float64
```

```
In [5]: #p value is rather small, ts is stationary
```

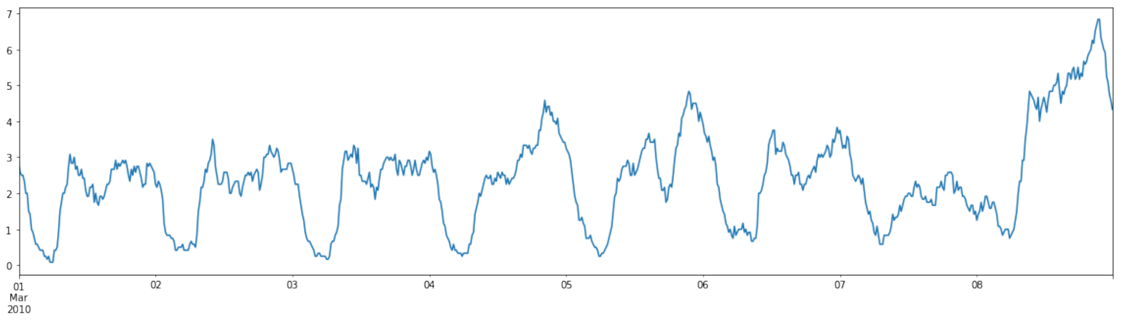
```
In [36]: #but ts has too many ups and downs, perform a smoothing by rolling mean by 12 hours win
         #get continuous time series cts
         #for example in March cts looks like this
         cts=ts.rolling(window=12).mean()
         cts.dropna(inplace=True)
         cts['2010-03-01':'2010-03-31'].plot(figsize=(20,5))
```

```
Out[36]: <matplotlib.axes._subplots.AxesSubplot at 0x1c2ecc74e0>
```



```
In [37]: cts['2010-03-01':'2010-03-08'].plot(figsize=(20,5))
```

```
Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3360b5c0>
```

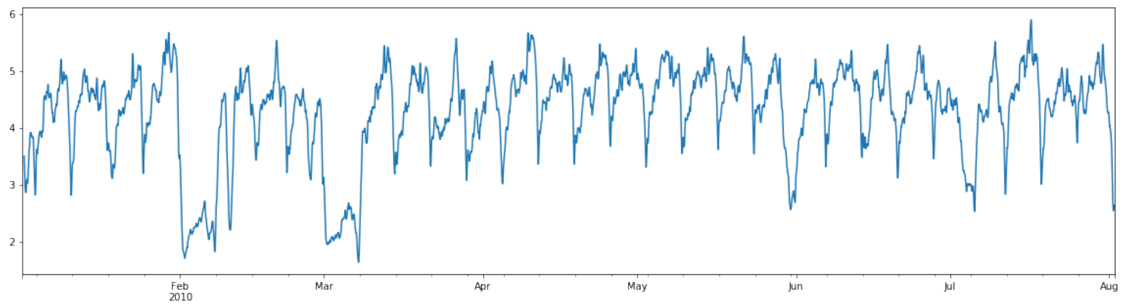


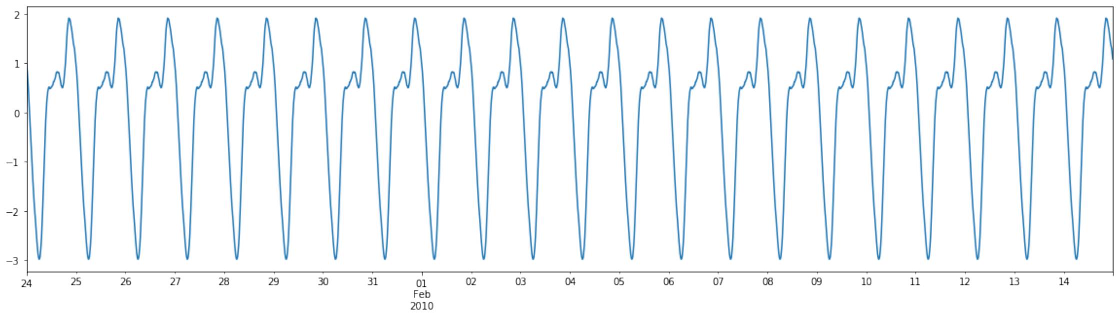
1.3 Seasonal-trend decomposition

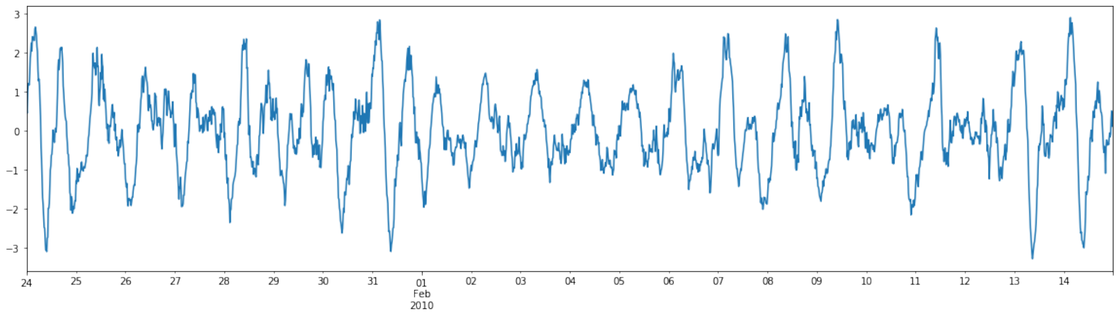
We apply a decomposition on cts into trend and seasonal part, see [7]. Graph 1, 2, 3 are trend, seasonal, residual. It confirms our previous observations, including the unusual low demand we reported in section 1.2. In fact, from the seasonal part, within each daily cycle, there is a jump in the afternoon, which also makes sense as daily activities arrives peak time in the afternoon.

```
In [7]: #considering the strong daily cycle, we decompose cts into trend + seasonal,
        #in fact it has some weekly seasonality which we first neglect here
        %matplotlib inline
        from statsmodels.tsa.seasonal import seasonal_decompose
        decomposition=seasonal_decompose(cts,freq=96)
        trend=decomposition.trend
        seasonal=decomposition.seasonal
        residual=decomposition.resid

        #trend in the whole 8 months
        plt.figure(figsize=(20, 5))
        trend['2010-1-1':'2010-8-1'].plot()
        #daily seasonality
        plt.figure(figsize=(20, 5))
        seasonal['2010-1-24':'2010-2-14'].plot()
        plt.figure(figsize=(20, 5))
        residual['2010-1-24':'2010-2-14'].plot()
```

```
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x123163358>
```







2 Modeling and Prediction

2.1 Methods

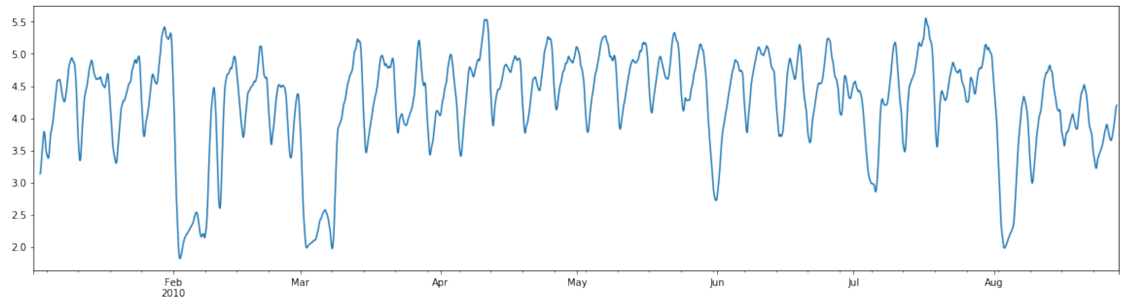
We apply two methods in this problem, the ARMA model (AutoRegressive Moving Average) and LSTM model (Long Short-term Memory). The first is common practice in time series analysis. For better performance, LSTM is introduced.

2.2 Method 1: ARMA model on trend and seasonal

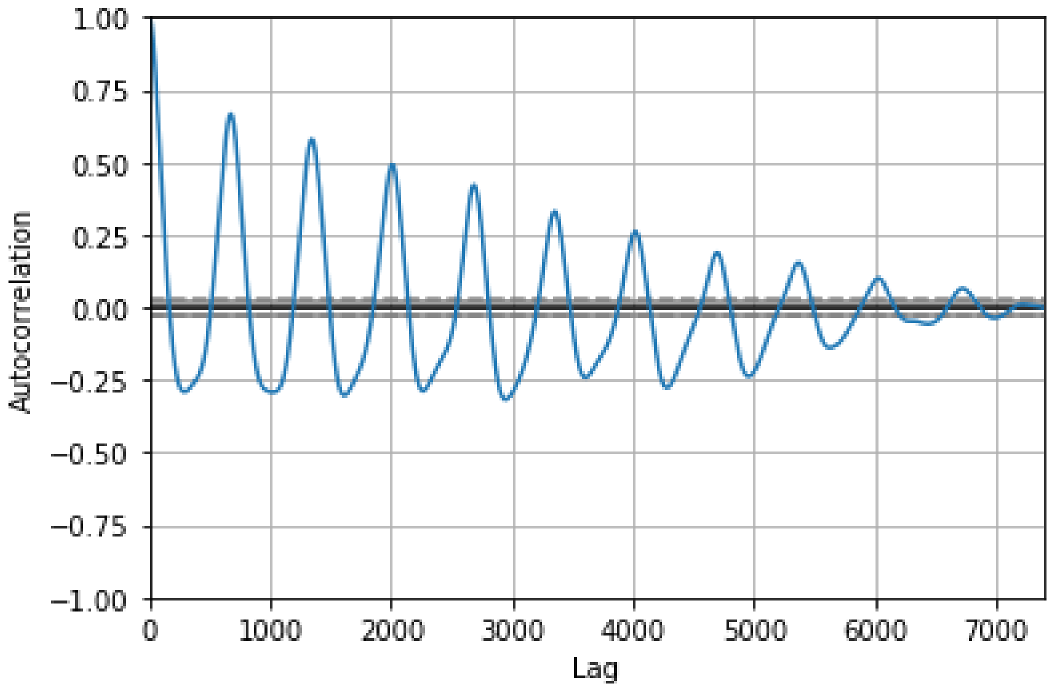
Our first strategy in this problem is to fit an ARMA model, as time series is already stationary, ARIMA is not needed. We didn't choose to fit a seasonal ARIMA model because of big computation. Instead, we fit ARMA on trend and seasonal respectively so that the best parameter could be found to model both parts.

We look at ACF, PACF graphs, fitted AR model. The residual is under 0.06 for trend and under 0.03 for seasonal. The residual of trend fit is almost a normal distribution centered at 0, and the residual of seasonal is constantly under 0.03, both convince us high accuracy of the model.

```
In [9]: #do another smoothing on trend
        ctrend=trend.rolling(window=96).mean()
        ctrend.dropna()
        ctrend.plot(figsize=(20,5))

Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x1c25e63ac8>
```

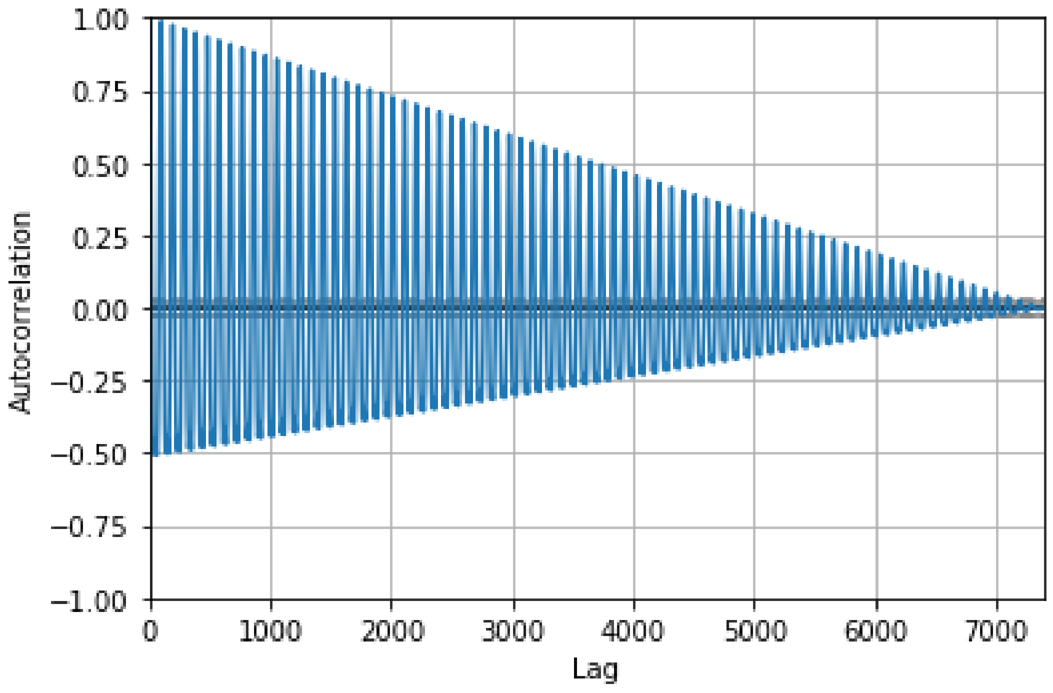


```
In [10]: from pandas.plotting import autocorrelation_plot

In [11]: autocorrelation_plot(ctrend['2010-3-10':'2010-5-25'])
         plt.figure(figsize=(20,5))
         plt.show()
```



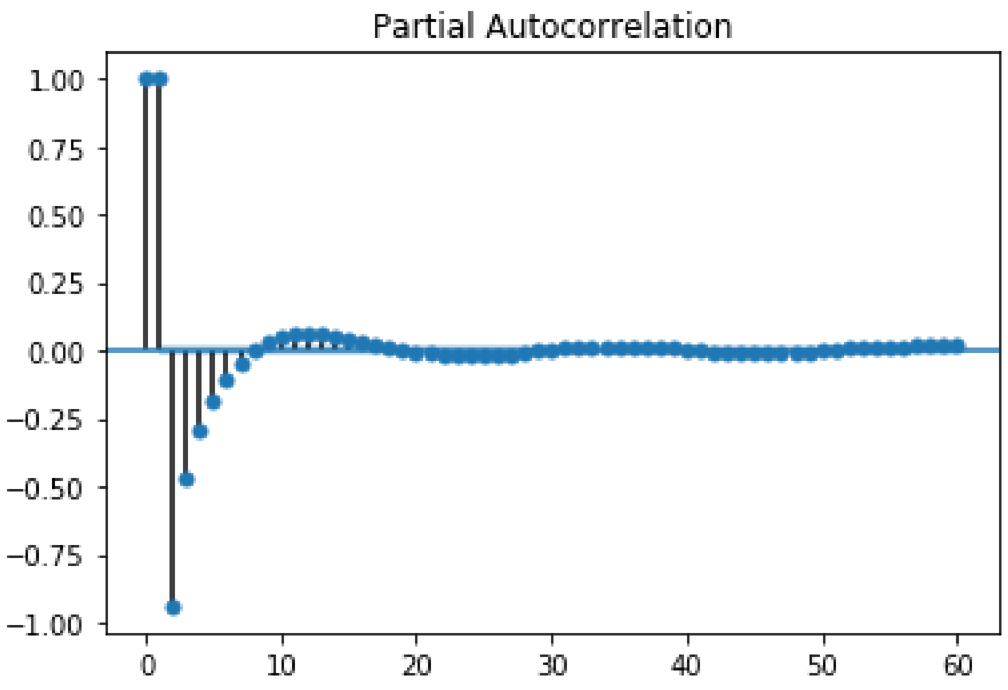
```
<Figure size 1440x360 with 0 Axes>
```

```
In [12]: autocorrelation_plot(seasonal['2010-3-10':'2010-5-25'])
         plt.figure(figsize=(20,5))
         plt.show()
```

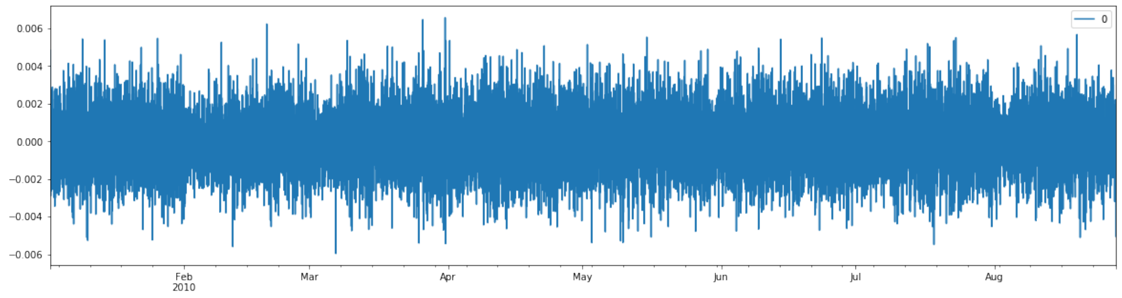


```
<Figure size 1440x360 with 0 Axes>
```

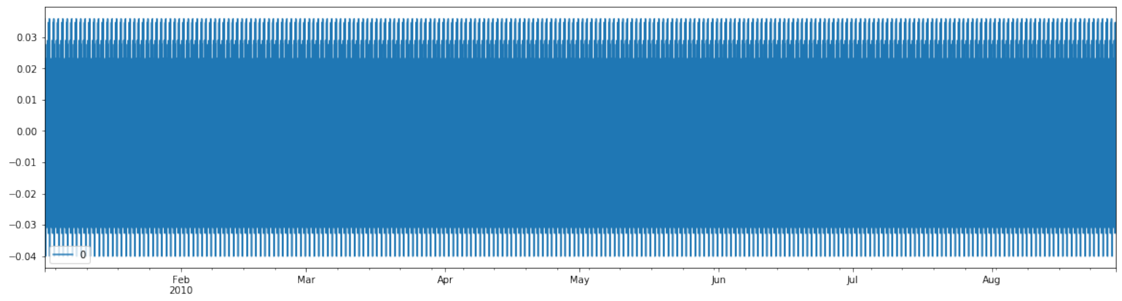
```
In [13]: from statsmodels.graphics.tsaplots import plot_pacf

In [14]: plot_pacf(ctrend['2010-3-10':'2010-5-25'],lags=60)

Out[14]:
```



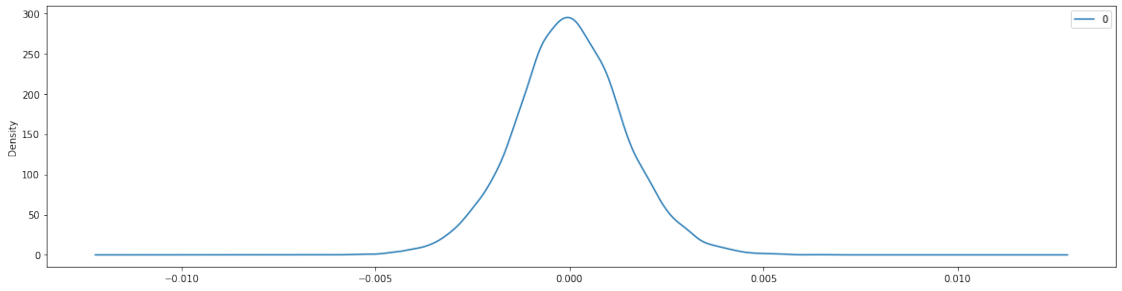
```
In [15]: #trend and seasonal are stationary, ACF decays geometrically, so it's enough to fit AR
         from statsmodels.tsa.ar_model import AR

In [16]: model = AR(trend['2010-01-01 14:45:00':'2010-08-28 02:45:00'])

In [17]: model_2=AR(seasonal)

In [18]: result=model.fit()

In [19]: result_2=model_2.fit()

In [20]: result.summary()

In [21]: # residuals are under 0.03, normal distributed with mean 0, good fitting
         from pandas import DataFrame
         residuals=DataFrame(result.resid)
         residuals.plot(figsize=(20, 5),subplots=False,layout=[5,5])

Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x1c29dc90f0>
```



```
In [22]: residuals_2=DataFrame(result_2.resid)
         residuals_2.plot(figsize=(20, 5),subplots=False,layout=[5,5])

Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x1238569b0>
```



```
In [23]: residuals.plot(kind='kde',figsize=(20,5))

Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x121f7dd30>
```



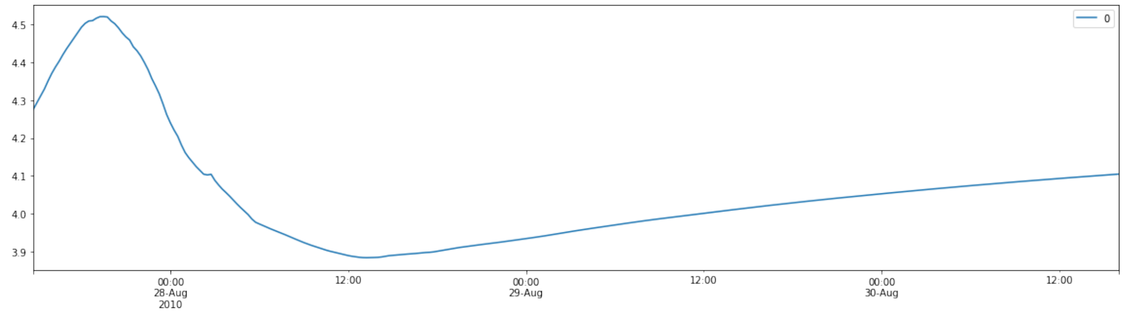
2.3 Total prediction

Using the models fitted, we predict on the number of logins for the next two days after the last time interval, [24] for trend and [25] for seasonal, and their sum is our prediction. Result is listed in [32], our prediction is 2010-08-28 15:00:00 4.712671, 2010-08-28 15:15:00 4.720942, 2010-08-28 15:30:00 4.737874, 2010-08-28 15:45:00 4.748609. As a sum of a slightly increasing trend and a flat seasonal component, this prediction is understandable.

```
In [24]: #predict trend and seasonal respectively then add together
         predi=result.predict(start='2010-08-27 14:45:00', end='2010-08-30 16:00:00')
         predictions=DataFrame(predi)
         predictions.plot(figsize=(20, 5),subplots=False,layout=[5,5])
```

```
/Users/beingcshen/anaconda3/lib/python3.6/site-packages/statsmodels/tsa/base/tsa_model.py:336: F
  freq=base_index.freq)
```

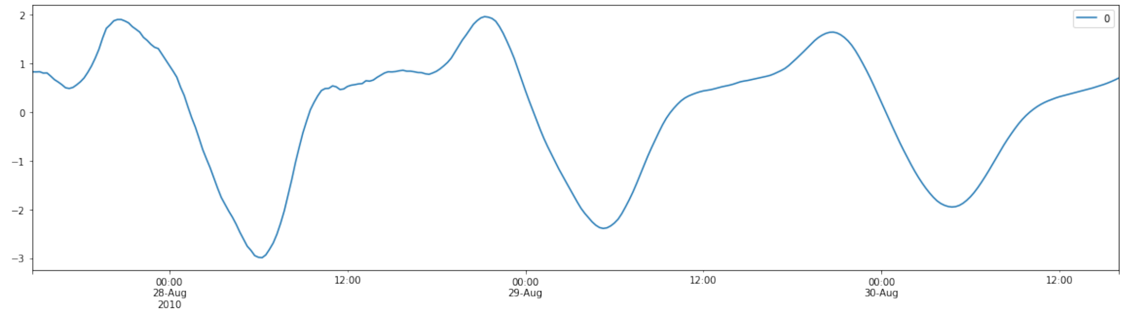
```
Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x122210f28>
```



```
In [25]: predi_s=result_2.predict(start='2010-08-27 14:45:00', end='2010-08-30 16:00:00')
         predictions_s=DataFrame(predi_s)
         predictions_s.plot(figsize=(20, 5),subplots=False,layout=[5,5])
```

```
/Users/beingcshen/anaconda3/lib/python3.6/site-packages/statsmodels/tsa/base/tsa_model.py:336: F
  freq=base_index.freq)
```

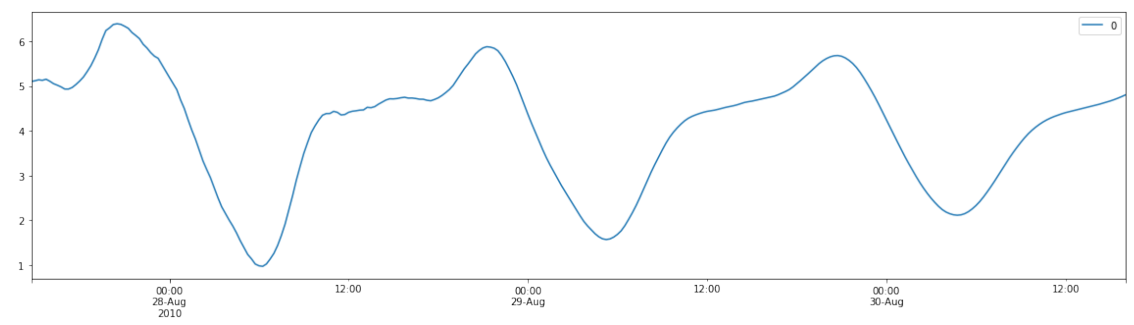
```
Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x1221b3d68>
```



```
In [26]: prediction_tot = predictions + predictions_s
```

```
In [27]: #total prediction around 2010-8-28
         prediction_tot.plot(figsize=(20,5))
```

```
Out[27]: <matplotlib.axes._subplots.AxesSubplot at 0x1c264ebc50>
```



```
In [32]: prediction_tot['2010-8-28 15:00:00':'2010-8-28 16:00:00']

Out[32]:                                0
         2010-08-28 15:00:00  4.712671
         2010-08-28 15:15:00  4.720942
         2010-08-28 15:30:00  4.737874
         2010-08-28 15:45:00  4.748609
         2010-08-28 16:00:00  4.730792
```

2.4 RNN Long Short-Term Model

One disadvantage of ARMA model is that it forgets the long term behavior easily. This is best illustrated from the prediction of the trend, the graph almost stays like a straight line one hour after the end point. As the nature of ARMA is just local linear regression, this defect can hardly be improved, so in this section we develop the long short-term memory from neural network. It aims to learn from the long-term behavior and capture the repeating short-term patterns, which is what we desire.

We first split the existing data into train and test set, perform a model fit and evaluate its accuracy, then fit another model on the complete dataset and predict for the future one hour.

We construct a neural network with one-day look back (since we focus on daily cycle) with 10 epochs. The prediction on the test set is the blue curve in [22], while the real data is orange in [22]. They are very close! Mean square error is 0.44, in acceptable range compared to 10.0 variation. This is satisfactory accuracy, we next conduct it on the complete data set.

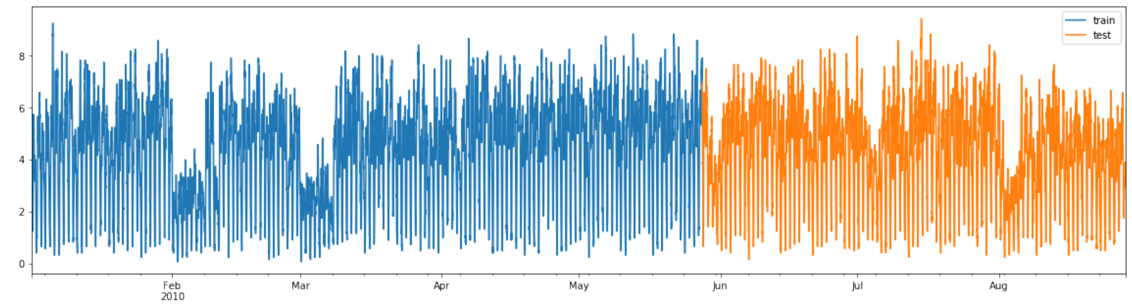
We use the average value 3.50 to extend the data set and predict the next one hour, prediction curve of the last day from 2010-8-27 16:00:00 to 2010-8-28 16:00:00 is shown in [27].

Our final prediction for next hour in the future is 2010-8-28 15:00:00 3.91, 2010-8-28 15:15:00 3.98, 2010-8-28 15:30:00 3.61, 2010-8-28 16:00:00 3.63.

This model fully captures the daily cycle, we can also take look back to be 7 days, which will take into account the weekly pattern.

```
In [5]: #train test split
        split_date = pd.Timestamp('2010-5-28')

        train = cts.loc[:split_date]
        test = cts.loc[split_date:]
        ax = train.plot()
        test.plot(ax=ax,figsize=(20,5))
        plt.legend(['train', 'test'])
```

```
Out[5]: <matplotlib.legend.Legend at 0x11fdb2f28>
```



```
In [8]: #construct correct data format for LSTM
        def create_dataset(dataset, look_back=1):
            dataX, dataY = [], []
            for i in range(len(dataset)-look_back-1):
                a = dataset[i:(i+look_back)]
                dataX.append(a)
                dataY.append(dataset[i + look_back])
                return np.array(dataX), np.array(dataY)
```

```
In [9]: look_back = 96  #a daily cycle contains 96 15minutes
        trainX,trainY = create_dataset(train, look_back)
        testX, testY = create_dataset(test, look_back)
```

```
In [10]: trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
         testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

```
In [15]: import math
         from keras.models import Sequential
         from keras.layers import Dense
         from keras.layers import LSTM
         from sklearn.preprocessing import MinMaxScaler
         from sklearn.metrics import mean_squared_error
```

```
In [16]: #train model
         model = Sequential()
```

```
In [17]: model.add(LSTM(4, input_shape=(1, look_back)))
```

```
In [18]: model.add(Dense(1))
```

```
In [19]: model.compile(loss='mean_squared_error', optimizer='adam')
         model.fit(trainX, trainY, epochs=10, batch_size=1, verbose=2)
```

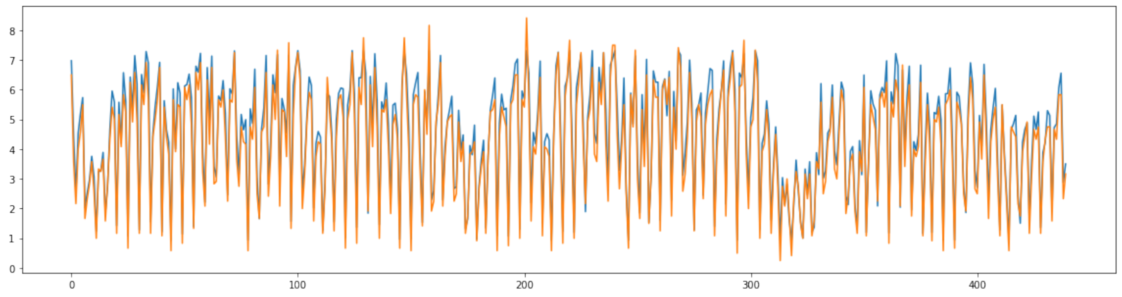
```
Epoch 1/10
 - 28s - loss: 1.8304
Epoch 2/10
 - 26s - loss: 0.6043
Epoch 3/10
 - 26s - loss: 0.2687
Epoch 4/10
 - 26s - loss: 0.1926
Epoch 5/10
 - 26s - loss: 0.1729
Epoch 6/10
 - 26s - loss: 0.1685
Epoch 7/10
 - 26s - loss: 0.1572
Epoch 8/10
 - 26s - loss: 0.1518
Epoch 9/10
 - 26s - loss: 0.1394
Epoch 10/10
 - 97s - loss: 0.1462
```

```
Out[19]: <keras.callbacks.History at 0x1a30a269b0>
```

```
In [20]: # predict on test set
         prediction = model.predict(testX)
```

```
In [22]: plt.figure(figsize=(20,5))
         plt.plot(prediction[::20])
         plt.plot(testY[::20])
```

```
Out[22]: [<matplotlib.lines.Line2D at 0x1a31671550>]
```



```
In [23]: trainPredict= model.predict(trainX)
         # calculate root mean squared error of the prediction on the test set
         from sklearn.metrics import mean_squared_error
         trainScore = math.sqrt(mean_squared_error(trainY, trainPredict))
         print('Train Score: %.2f RMSE' % (trainScore))
         testScore = math.sqrt(mean_squared_error(testY, prediction))
         print('Test Score: %.2f RMSE' % (testScore))
```

```
Train Score: 0.44 RMSE
Test Score: 0.44 RMSE
```

```
In [24]: #train LSTM using total time series
         ctsX, ctsY= create_dataset(cts,96)
         ctsX= np.reshape(ctsX, (ctsX.shape[0],1, ctsX.shape[1]))
         model.fit(ctsX, ctsY,epochs=10, batch_size=1, verbose=2)
```

```
Epoch 1/10
 - 42s - loss: 0.1344
Epoch 2/10
 - 43s - loss: 0.1317
Epoch 3/10
 - 42s - loss: 0.1361
Epoch 4/10
 - 44s - loss: 0.1290
Epoch 5/10
 - 47s - loss: 0.1278
Epoch 6/10
 - 45s - loss: 0.1273
Epoch 7/10
 - 42s - loss: 0.1268
Epoch 8/10
 - 45s - loss: 0.1266
Epoch 9/10
 - 42s - loss: 0.1338
Epoch 10/10
 - 43s - loss: 0.1264
```

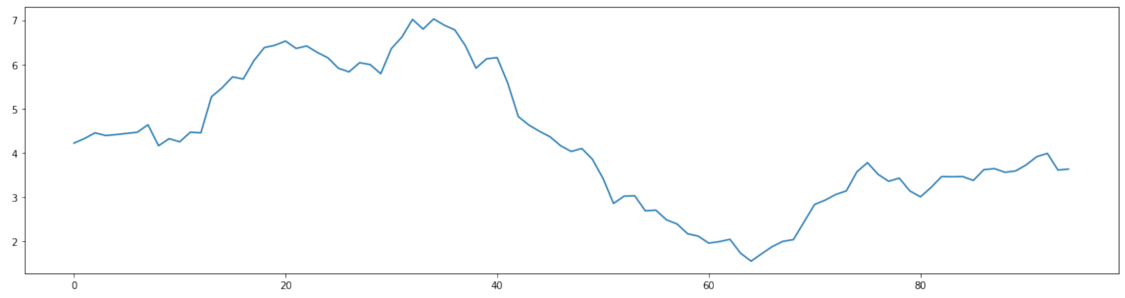
```
Out[24]: <keras.callbacks.History at 0x11a76a588>
```

```
In [25]: prediction=model.predict(ctsX)
```

```
In [26]: #using 3.5 to extend the total time series by one hour and predict
         future= pd.date_range('2010-08-28 15:00:00', periods=4, freq='15T')
         futurets = pd.Series(3.5, index=future)
         cfts=cts.append(futurets)
         cftsX, cftsY=create_dataset(cfts,96)
         cftsX=np.reshape(cftsX,(cftsX.shape[0],1,cftsX.shape[1]))
```

```
In [27]: cfpredict=model.predict(cftsX)
         plt.figure(figsize=(20,5))
         plt.plot(cfpredict[-96:-1])
```

```
Out[27]: [<matplotlib.lines.Line2D at 0x1a315f2080>]
```



```
In [28]: cfpredict[-5:-1]

Out[28]: array([[3.9136057],
               [3.986833 ],
               [3.609972 ],
               [3.6320686]], dtype=float32)
```