PROJECT

New York City Taxi Fare Prediction

Cao Van Minh (caovanminh94@gmail.com)
Ngo Van Truong Manh (nvtruongmanh@gmail.com)

TEAM 6

Table of Content

A. ASSUMPTION

Add your own subtitle here.

PROJECT: New York City Taxi Fare Prediction

Our mission:

In this notebook, we will perform a time series analysis (TSA) using some modeling techniques, from Naive Approach to ARIMA model. The purpose is that we want to learn what is the difference between a time series analysis and another method (for example: linear regression)? How to do a time series analysis?

Our scenario:

There is a transportation company at NYC. BOD of this company want to expande their business into Taxi service. We are in analysis team. Our tasks is to make an analysis report, which will help BOD make decision.

In [16]:

```python
# Outlier values
# Scatter plot
train_temp = train.drop(columns=['pickup_datetime'])
train_temp.plot(figsize=(15,8), style='k.')
plt.show()
```

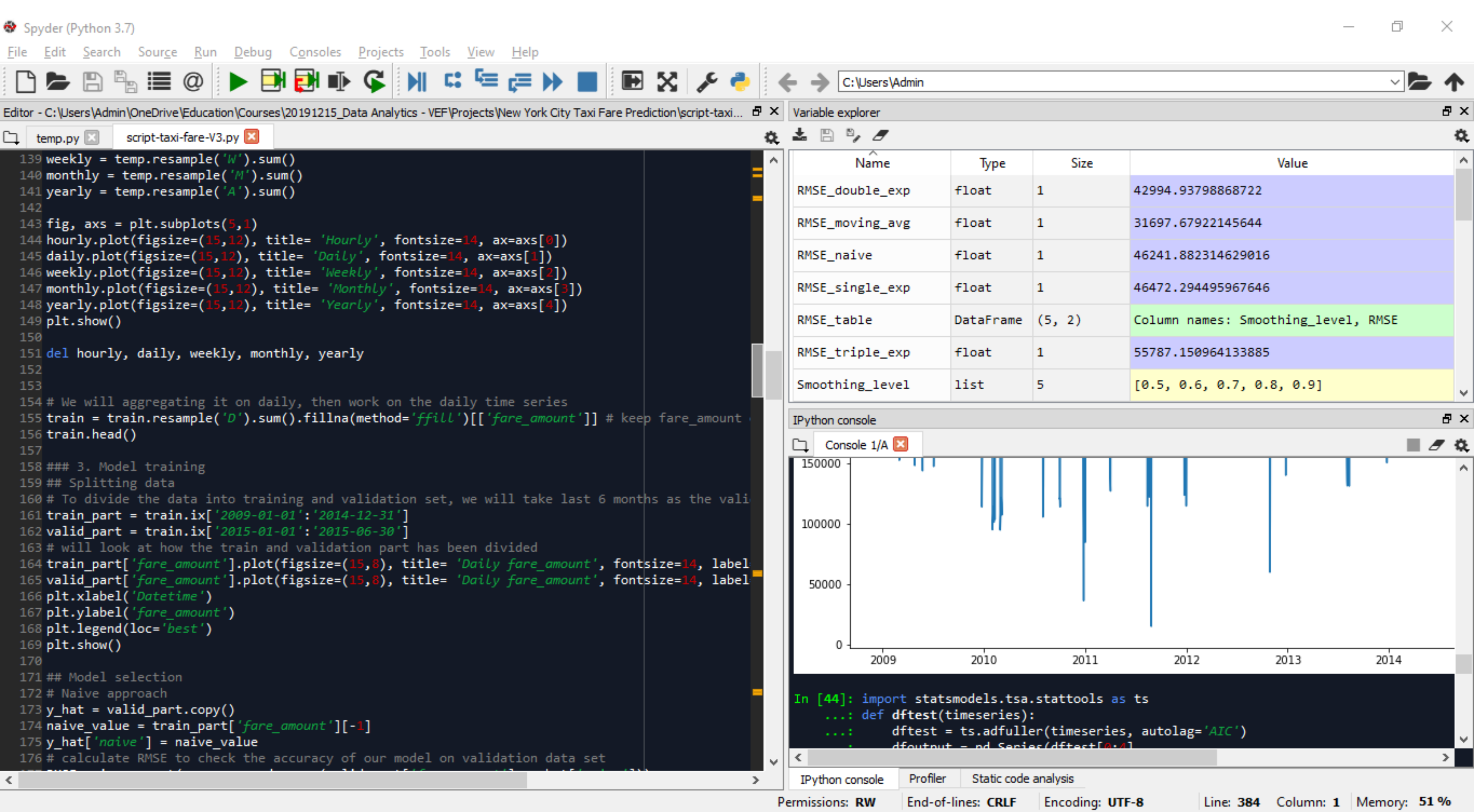
MEMORY ERROR

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
~\Anaconda3\lib\site-packages\pandas\plotting\_matplotlib\converter.py in _convert_1d(values, unit, axis)
    296             if isinstance(values, Index):
--> 297                 values = _dt_to_float_ordinal(values)
    298             else:

~\Anaconda3\lib\site-packages\pandas\plotting\_matplotlib\converter.py in _dt_to_float_ordinal(dt)
    244     if isinstance(dt, (np.ndarray, Index, ABCSeries)) and is_datetime64_ns_dtype(dt):
--> 245         base = dates.epoch2num(dt.asi8 / 1.0e9)
    246     else:

MemoryError: 

During handling of the above exception, another exception occurred:

MemoryError                               Traceback (most recent call last)
~\Anaconda3\lib\site-packages\matplotlib\axis.py in convert_units(self, x)
   1549         try:
```

In [17]:

```python
# Create group of Taxi Fare
train_temp['fare_group'] = pd.cut(train_temp['fare_amount'], [-301, 0, 100, 200, 300, 400, 500, 93970], right=False)
train_count = train_temp['fare_group'].value_counts(sort=False)
train_pct = train_temp['fare_group'].value_counts(sort=False, normalize=True)
train_crosstab = pd.concat([train_count, train_pct], axis=1)
train_crosstab.columns = ['Counts', 'Percentage']
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-17-e56d06945d48> in <module>
```


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\Admin
Editor - C:\Users\Admin\OneDrive\Education\Courses\20191215_Data Analytics - VEF\Projects\New York City Taxi Fare Prediction\script-taxi...
temp.py
script-taxi-fare-V3.py
weekly = temp.resample('W').sum()
monthly = temp.resample('M').sum()
yearly = temp.resample('A').sum()
fig, axs = plt.subplots(5,1)
hourly.plot(figsize=(15,12), title= 'Hourly', fontsize=14, ax=axs[0])
daily.plot(figsize=(15,12), title= 'Daily', fontsize=14, ax=axs[1])
weekly.plot(figsize=(15,12), title= 'Weekly', fontsize=14, ax=axs[2])
monthly.plot(figsize=(15,12), title= 'Monthly', fontsize=14, ax=axs[3])
yearly.plot(figsize=(15,12), title= 'Yearly', fontsize=14, ax=axs[4])
plt.show()
del hourly, daily, weekly, monthly, yearly
# We will aggregating it on daily, then work on the daily time series
train = train.resample('D').sum().fillna(method='ffill')[['fare_amount']] # keep fare_amount
train.head()
3. Model training
Splitting data
# To divide the data into training and validation set, we will take last 6 months as the vali
train_part = train.ix['2009-01-01':'2014-12-31']
valid_part = train.ix['2015-01-01':'2015-06-30']
# will look at how the train and validation part has been divided
train_part['fare_amount'].plot(figsize=(15,8), title= 'Daily fare_amount', fontsize=14, label
valid_part['fare_amount'].plot(figsize=(15,8), title= 'Daily fare_amount', fontsize=14, label
plt.xlabel('Datetime')
plt.ylabel('fare_amount')
plt.legend(loc='best')
plt.show()
Model selection
# Naive approach
y_hat = valid_part.copy()
naive_value = train_part['fare_amount'][-1]
y_hat['naive'] = naive_value
calculate RMSE to check the accuracy of our model on validation data set
Variable explorer
Name | Type | Size | Value
RMSE_double_exp | float | 1 | 42994.93798868722
RMSE_moving_avg | float | 1 | 31697.67922145644
RMSE_naive | float | 1 | 46241.882314629016
RMSE_single_exp | float | 1 | 46472.294495967646
RMSE_table | DataFrame | (5, 2) | Column names: Smoothing_level, RMSE
RMSE_triple_exp | float | 1 | 55787.150964133885
Smoothing_level | list | 5 | [0.5, 0.6, 0.7, 0.8, 0.9]
IPython console
Console 1/A
150000
100000
50000
0
2009
2010
2011
2012
2013
2014
In [44]: import statsmodels.tsa.stattools as ts
...: def dftest(timeseries):
...: dftest = ts.adfuller(timeseries, autolag='AIC')
IPython console
Profiler
Static code analysis
Permissions: RW
End-of-lines: CRLF
Encoding: UTF-8
Line: 384
Column: 1
Memory: 51 %

B. EDA

Loading libraries

```
In [1]: import pandas as pd                    # package for data frame analysis
   ...: import numpy as np                     # package for mathematical calculation
   ...: import matplotlib.pyplot as plt     # package for graphical plot
   ...: from datetime import datetime    # To access datetime
   ...: from sklearn.metrics import mean_squared_error
   ...: from math import sqrt
   ...: from statsmodels.tsa.api import SimpleExpSmoothing, Holt, ExponentialSmoothing
   ...: from statsmodels.tsa.stattools import adfuller, acf, pacf
```

B. EDA

Understanding the large csv file

```
In [2]: trainpath = r'C:\Users\Admin\datasets\nyc_taxi_fare_train.csv'
   ...: # How many rows are there in this csv file
   ...: with open(trainpath) as file:
   ...:     n_rows = len(file.readlines())
   ...:
   ...: print (f'Exact number of rows: {n_rows}')
Exact number of rows: 55423857
```

```
In [3]: df_tmp = pd.read_csv(trainpath, nrows=5)
   ...: df_tmp.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 8 columns):
key                  5 non-null object
fare_amount          5 non-null float64
pickup_datetime      5 non-null object
pickup_longitude     5 non-null float64
pickup_latitude      5 non-null float64
dropoff_longitude    5 non-null float64
dropoff_latitude     5 non-null float64
passenger_count      5 non-null int64
dtypes: float64(5), int64(1), object(2)
memory usage: 448.0+ bytes
```

Based on our assumption:

1. We don't need these cols for our TSA: **key, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, passenger_count**
2. Choose the optimization data type

B. EDA

Understanding the large csv file

```
In [57]: pd.set_option('display.max_columns', 500)

In [58]: df_tmp.head()
Out[58]:
                             key  fare_amount          pickup_datetime  \
0    2009-06-15 17:26:21.0000001          4.5  2009-06-15 17:26:21 UTC
1    2010-01-05 16:52:16.0000002         16.9  2010-01-05 16:52:16 UTC
2   2011-08-18 00:35:00.00000049          5.7  2011-08-18 00:35:00 UTC
3    2012-04-21 04:30:42.0000001          7.7  2012-04-21 04:30:42 UTC
4  2010-03-09 07:51:00.000000135          5.3  2010-03-09 07:51:00 UTC

   pickup_longitude  pickup_latitude  dropoff_longitude  dropoff_latitude  \
0        -73.844311        40.721319         -73.841610         40.712278
1        -74.016048        40.711303         -73.979268         40.782004
2        -73.982738        40.761270         -73.991242         40.750562
3        -73.987130        40.733143         -73.991567         40.758092
4        -73.968095        40.768008         -73.956655         40.783762

   passenger_count
0                1
1                1
2                2
3                1
4                1
```

Based on our assumption:

1. We don't need these cols for our TSA: **key, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, passenger_count**
2. Choose the optimization data type

B. EDA

Loading dataset

```
In [5]: trainpath = r'C:\Users\Admin\datasets\nyc_taxi_fare_train.csv'
   ...: traintypes = {'fare_amount': 'float32',
   ...:               'pickup_datetime': 'str'}
   ...: cols = list(traintypes.keys())
   ...: train = pd.read_csv(trainpath, usecols=cols, dtype=traintypes)
   ...: train.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 55423856 entries, 0 to 55423855
Data columns (total 2 columns):
fare_amount        float32
pickup_datetime    object
dtypes: float32(1), object(1)
memory usage: 634.3+ MB
```

Based on our assumption:

1. We don't need these cols for our TSA: **key, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, passenger_count**
2. Choose the optimization data type

B. EDA

Data Cleaning

```
In [7]: train.index
Out[7]: RangeIndex(start=0, stop=55423856, step=1)
```

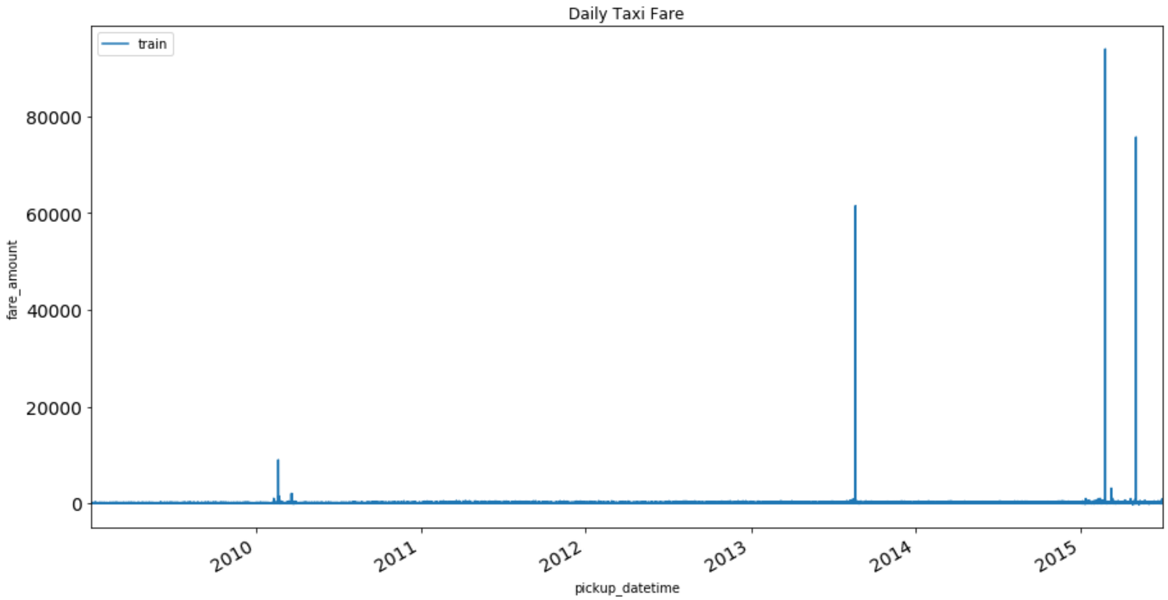
We'll convert RangeIndex into DateTimeIndex to do TSA

```
In [8]: train['pickup_datetime'] = train['pickup_datetime'].str.replace(" UTC","")
   ...: train['pickup_datetime'] = pd.to_datetime(train['pickup_datetime'], format='%Y-%m-%d %H:%M:%S')
   ...: # Set index using pickup_datetime col
   ...: train.set_index('pickup_datetime', inplace=True, drop=False)
   ...: train.index
Out[8]:
DatetimeIndex(['2009-06-15 17:26:21', '2010-01-05 16:52:16',
               '2011-08-18 00:35:00', '2012-04-21 04:30:42',
               '2010-03-09 07:51:00', '2011-01-06 09:50:45',
               '2012-11-20 20:35:00', '2012-01-04 17:22:00',
               '2012-12-03 13:10:00', '2009-09-02 01:11:00',
               ...
               '2010-05-28 07:49:50', '2011-09-16 00:46:43',
               '2013-05-24 00:13:36', '2014-03-04 22:25:01',
               '2015-03-22 16:37:27', '2014-03-15 03:28:00',
               '2009-03-24 20:46:20', '2011-04-02 22:04:24',
               '2011-10-26 05:57:51', '2014-12-12 11:33:00'],
              dtype='datetime64[ns]', name='pickup_datetime', length=55423856, freq=None)
```

B. EDA

Outlier Values

```
# First of all, get an overview, how fare_amount it was?
train['fare_amount'].plot(figsize=(15,8), title= 'Daily Taxi Fare', fontsize=14, label='fare_amount')
plt.xlabel("pickup_datetime")
plt.ylabel("fare_amount")
plt.legend(loc='best')
plt.show()
```

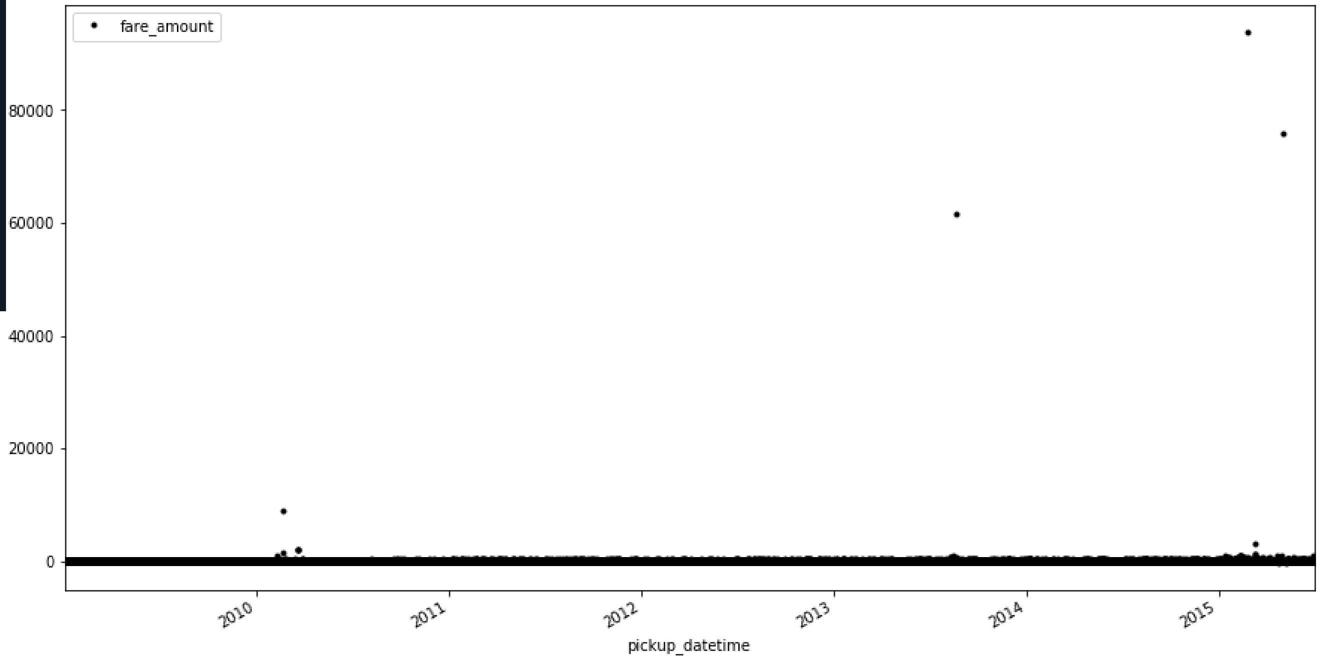


B. EDA

Outlier Values

```
# Scatter plot
train_temp = train.drop(columns=['pickup_datetime'])
train_temp.plot(figsize=(15,8), style='k.')
plt.show()
```

```
In [63]: train['fare_amount'].describe()
Out[63]:
count    5.542386e+07
mean     8.077921e+00
std      2.055127e+01
min     -3.000000e+02
25%      6.000000e+00
50%      8.500000e+00
75%      1.250000e+01
max      9.396336e+04
Name: fare_amount, dtype: float64
```



B. EDA

Outlier Values

```
In [10]: train_temp = train.drop(columns=['pickup_datetime'])

In [11]: train_temp['fare_group'] = pd.cut(train_temp['fare_amount'],
    ...:                  [-301, 0, 100, 200, 300, 400, 500, 93970], right=False)
    ...: train_count = train_temp['fare_group'].value_counts(sort=False)
    ...: train_pct = train_temp['fare_group'].value_counts(sort=False, normalize=True)
    ...: train_crosstab = pd.concat([train_count, train_pct], axis=1)
    ...: train_crosstab.columns = ['Counts', 'Percentage']
    ...: train_crosstab
Out[11]:
```

	Counts	Percentage
[-301, 0)	2454	0.000044
[0, 100)	55398503	0.999543
[100, 200)	20934	0.000378
[200, 300)	1548	0.000028
[300, 400)	222	0.000004
[400, 500)	138	0.000002
[500, 93970)	57	0.000001

Handling outlier values: remove below 0 and above 100

B. EDA

Missing Values

```
In [13]: train.isnull().sum()
Out[13]:
fare_amount        0
pickup_datetime    0
dtype: int64

In [14]: train = train[(train['fare_amount'] > 0) & (train['fare_amount'] < 100)]
```

Handling outlier values: remove below 0 and above 100

B. EDA

Data Transformation

```
In [15]: train['year'] = train['pickup_datetime'].dt.year
    ...: train['month'] = train['pickup_datetime'].dt.month
    ...: train['day'] = train['pickup_datetime'].dt.day
    ...: train['hour'] = train['pickup_datetime'].dt.hour
    ...: train['day_of_week'] = train['pickup_datetime'].dt.dayofweek
    ...: # Make a weekend var
    ...: def is_weekend(row):
    ...:     if row.dayofweek == 5 or row.dayofweek == 6:
    ...:         return 1
    ...:     else:
    ...:         return 0
    ...:
    ...: train['weekend'] = train['pickup_datetime'].apply(is_weekend)
```

Extract Year, Month, Day, Hour; Day of week, weekend from the pickup_datetime

```
In [26]: train.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 55397123 entries, 2009-06-15 17:26:21 to 2014-12-12 11:33:00
Data columns (total 8 columns):
fare_amount        float32
pickup_datetime    datetime64[ns]
year               int64
month              int64
day                int64
hour               int64
day_of_week        int64
weekend            int64
dtypes: datetime64[ns](1), float32(1), int64(6)
memory usage: 3.5 GB
```

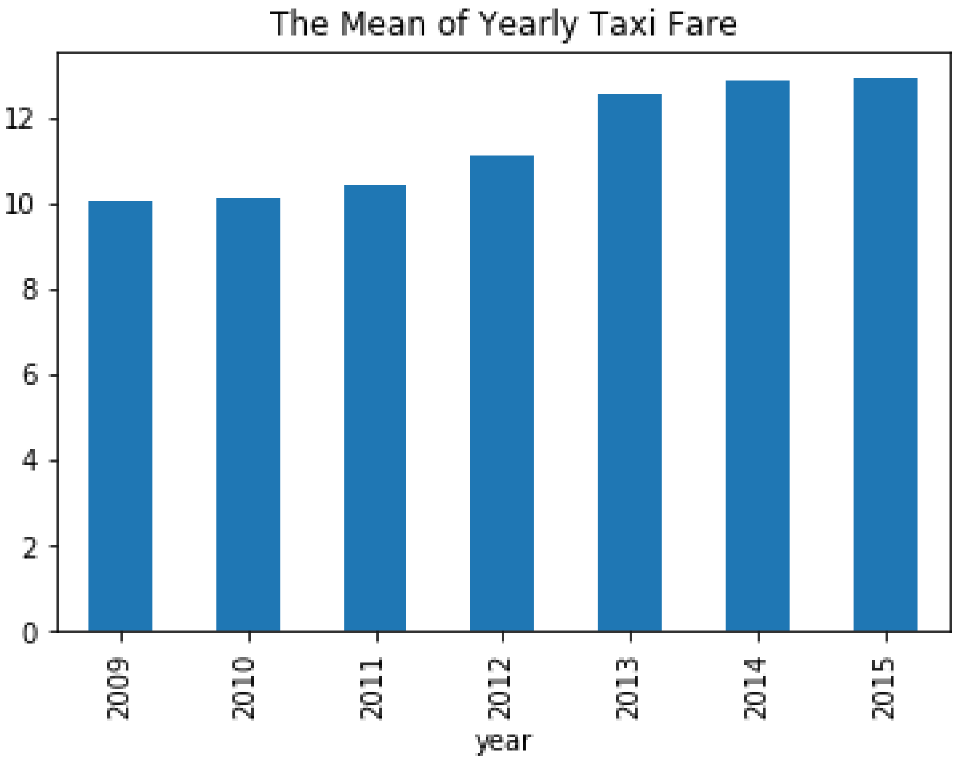
Let's deep dive into new feature

B. EDA

Data Transformation

```
In [16]: train.groupby('year')['fare_amount'].mean().plot.bar(title= 'The Mean of Yearly Taxi Fare')
Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x2ceab1bed08>
```

fare_amount increase as the years pass by

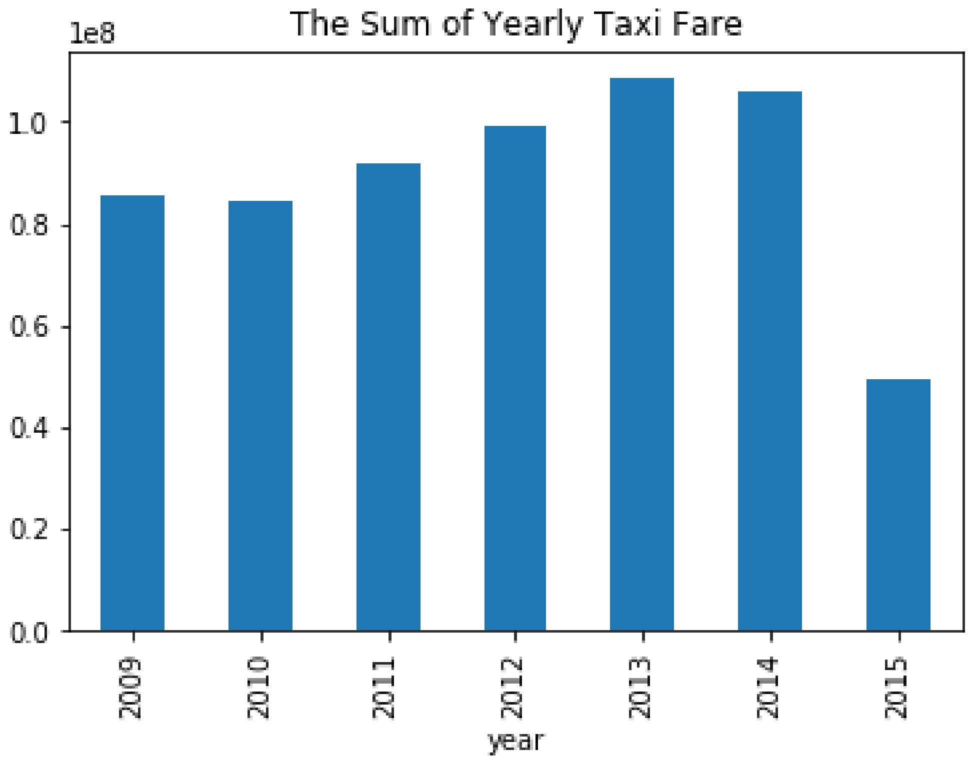


B. EDA

Data Transformation

```
In [17]: train.groupby('year')['fare_amount'].sum().plot.bar(title= 'The Sum of Yearly Taxi Fare')
Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x2cd86abcc88>
```

fare_amount increase as the years pass by

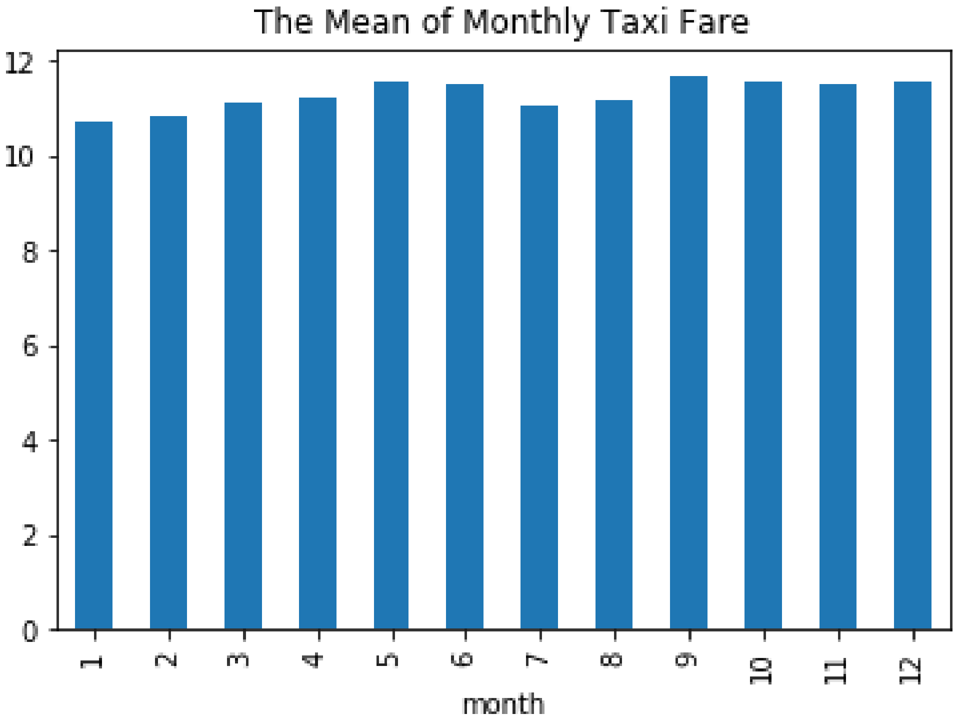


B. EDA

Data Transformation

```
In [18]: train.groupby('month')['fare_amount'].mean().plot.bar(title='The Mean of Monthly Taxi Fare')
Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x2cd9b83e5c8>
```

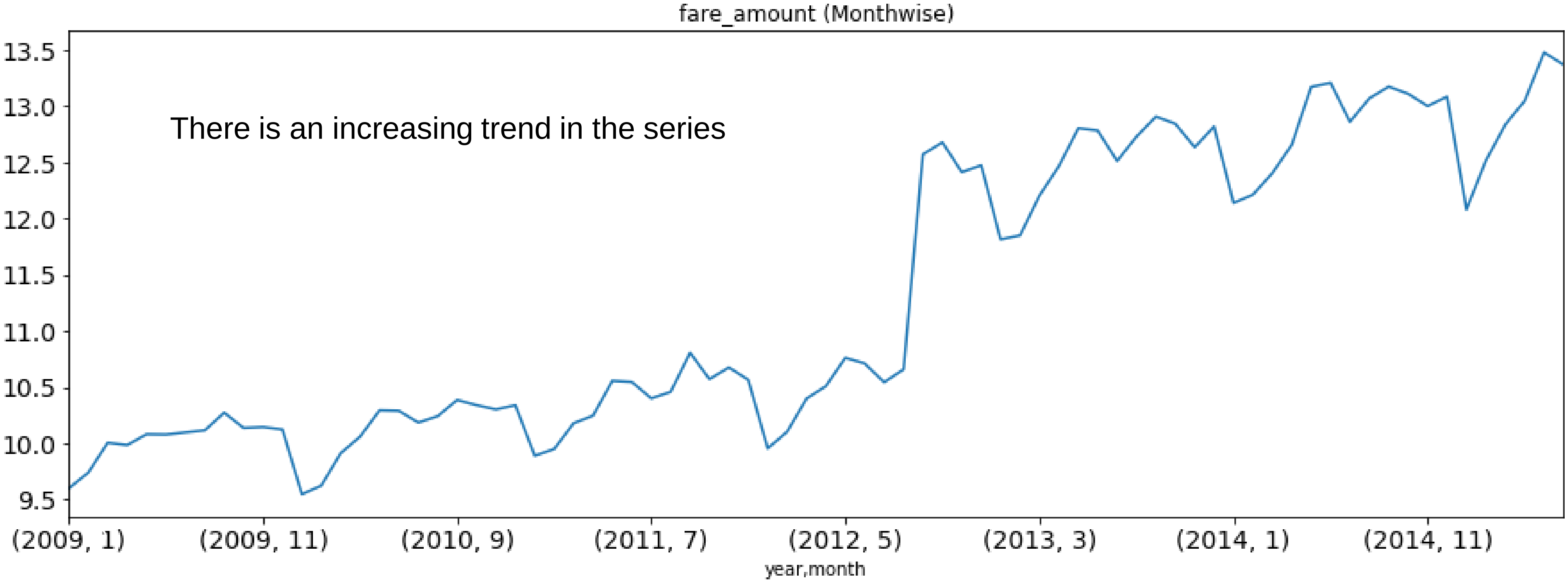
fare_amount increasing from JAN to MAY, then decreasing, and increasing from SEP to OCT again



B. EDA

Data Transformation

```
In [19]: temp = train.groupby(['year', 'month'])['fare_amount'].mean()

In [20]: temp.plot(figsize=(15,5), title= 'fare_amount (Monthwise)', fontsize=14)
Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x2cd99e59748>
```



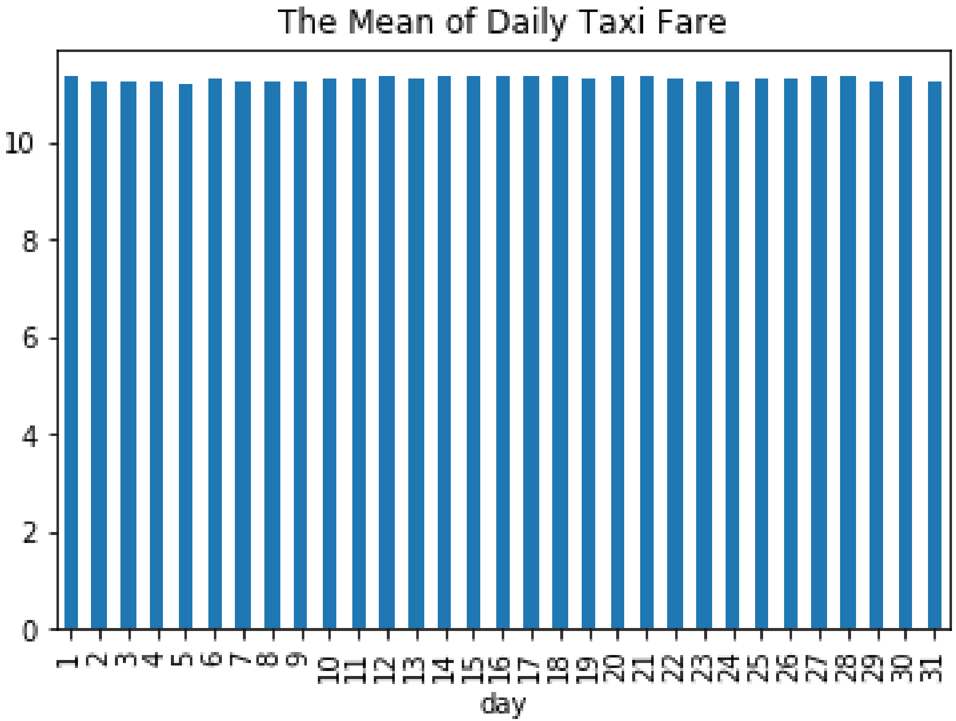
There is an increasing trend in the series

B. EDA

Data Transformation

```
In [21]: train.groupby('day')['fare_amount'].mean().plot.bar(title='The Mean of Daily Taxi Fare')
Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x2cda1bd5d48>
```

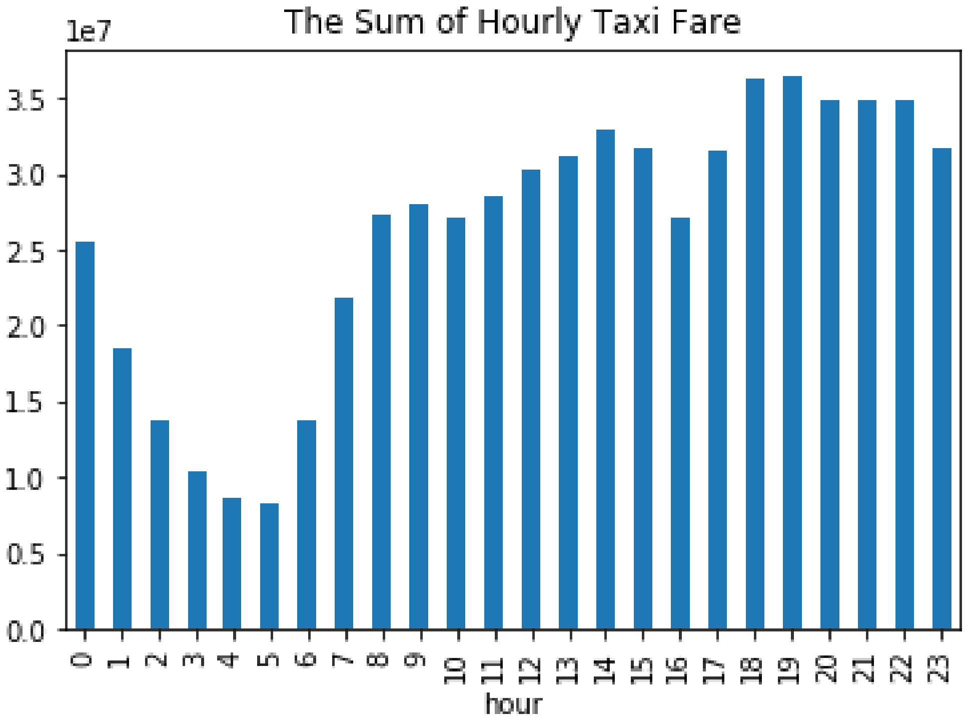
We are not getting much insights from day wise fare_amount



B. EDA

Data Transformation

```
In [23]: train.groupby('hour')['fare_amount'].sum().plot.bar(title='The Sum of Hourly Taxi Fare') # Khung gio cao diem
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x2cdb59bdbc8>
```



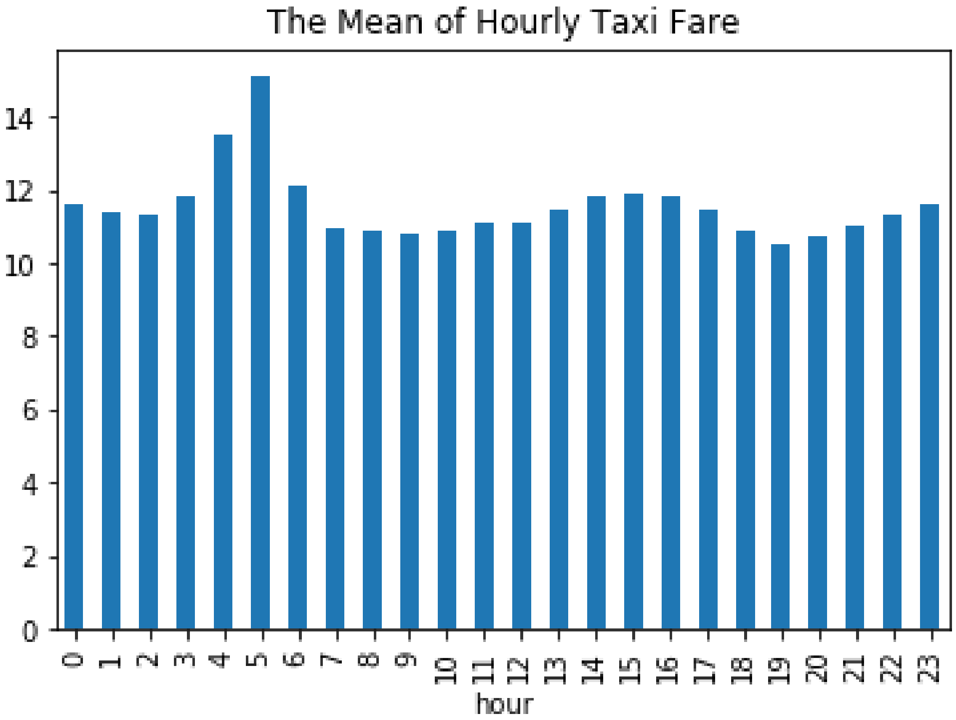
fare_amount got the peak at 7PM, and then decreasing trend till 5AM
after that fare_amount increasing again and peak again 2PM;
But, this plot show that there is a high price at 5AM Low price at 7PM

B. EDA

Data Transformation

```
In [22]: train.groupby('hour')['fare_amount'].mean().plot.bar(title='The Mean of Hourly Taxi Fare') # Gia cao
Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x2cda98e9488>
```

fare_amount got the peak at 7PM, and then decreasing trend till 5AM
after that fare_amount increasing again and peak again 2PM;
But, this plot show that there is a high price at 5AM Low price at 7PM

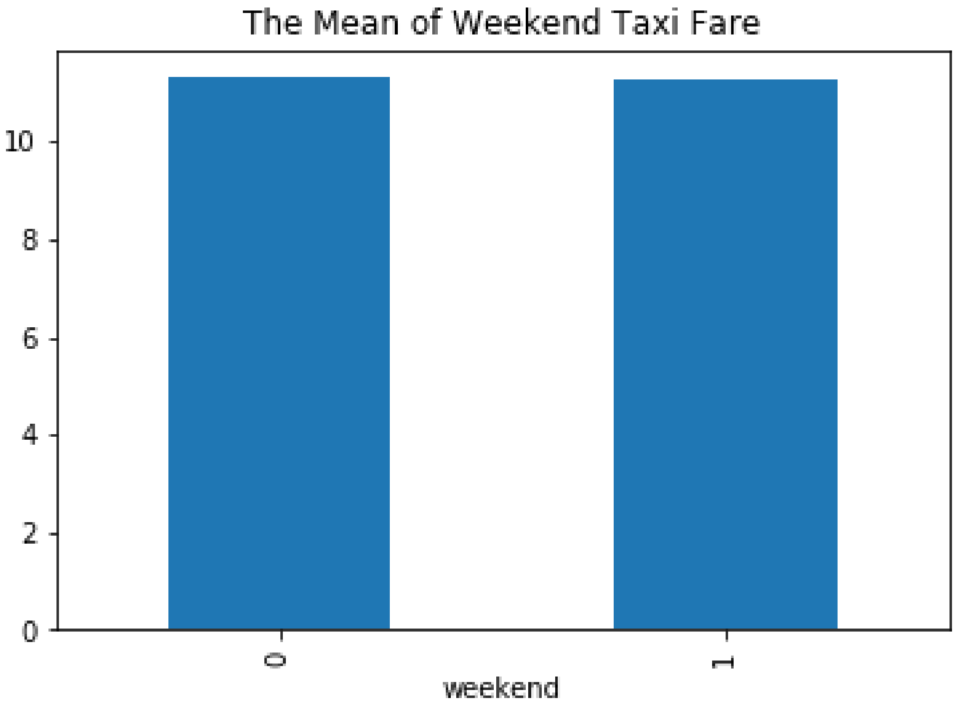


B. EDA

Data Transformation

```
In [24]: train.groupby('weekend')['fare_amount'].mean().plot.bar(title='The Mean of Weekend Taxi Fare')
Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x2cdbe694108>
```

there is no difference between weekday and weekend

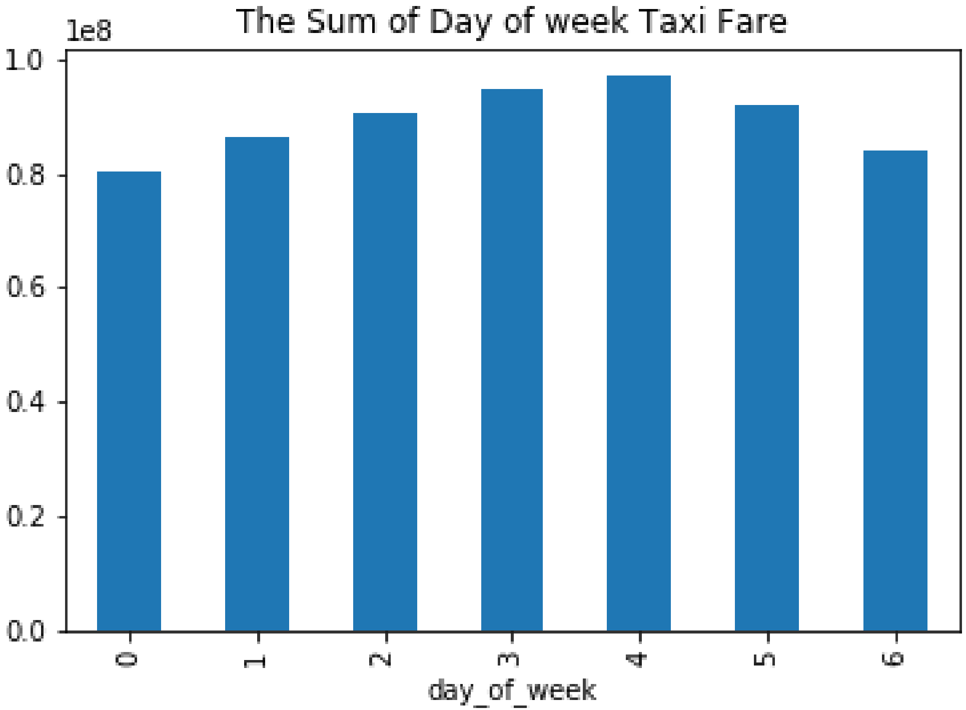


B. EDA

Data Transformation

```
In [25]: train.groupby('day_of_week')['fare_amount'].sum().plot.bar(title='The Sum of Day of week Taxi Fare')
Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x2cdbebfe548>
```

fare_amount is less for Saturday, Sunday as compared to the other days of the week.
It peaked at Friday



B. EDA

Data Transformation

```
In [28]: train = train.sort_index() # sort index
    ...: temp = train['fare_amount']
    ...: hourly = temp.resample('H').sum()
    ...: daily = temp.resample('D').sum()
    ...: weekly = temp.resample('W').sum()
    ...: monthly = temp.resample('M').sum()
    ...: yearly = temp.resample('A').sum()

In [29]: fig, axs = plt.subplots(5,1)
    ...: hourly.plot(figsize=(15,12), title= 'Hourly', fontsize=14, ax=axs[0])
    ...: daily.plot(figsize=(15,12), title= 'Daily', fontsize=14, ax=axs[1])
    ...: weekly.plot(figsize=(15,12), title= 'Weekly', fontsize=14, ax=axs[2])
    ...: monthly.plot(figsize=(15,12), title= 'Monthly', fontsize=14, ax=axs[3])
    ...: yearly.plot(figsize=(15,12), title= 'Yearly', fontsize=14, ax=axs[4])
    ...: plt.show()
```

Let's look at the hourly, daily, weekly, monthly time series

resample('H', 'D', 'W', 'M').sum() : Aggregate the hourly time series to daily, weekly, monthly time series to reduce the noise and make it more stable

We will aggregating it on daily, then work on the daily time series

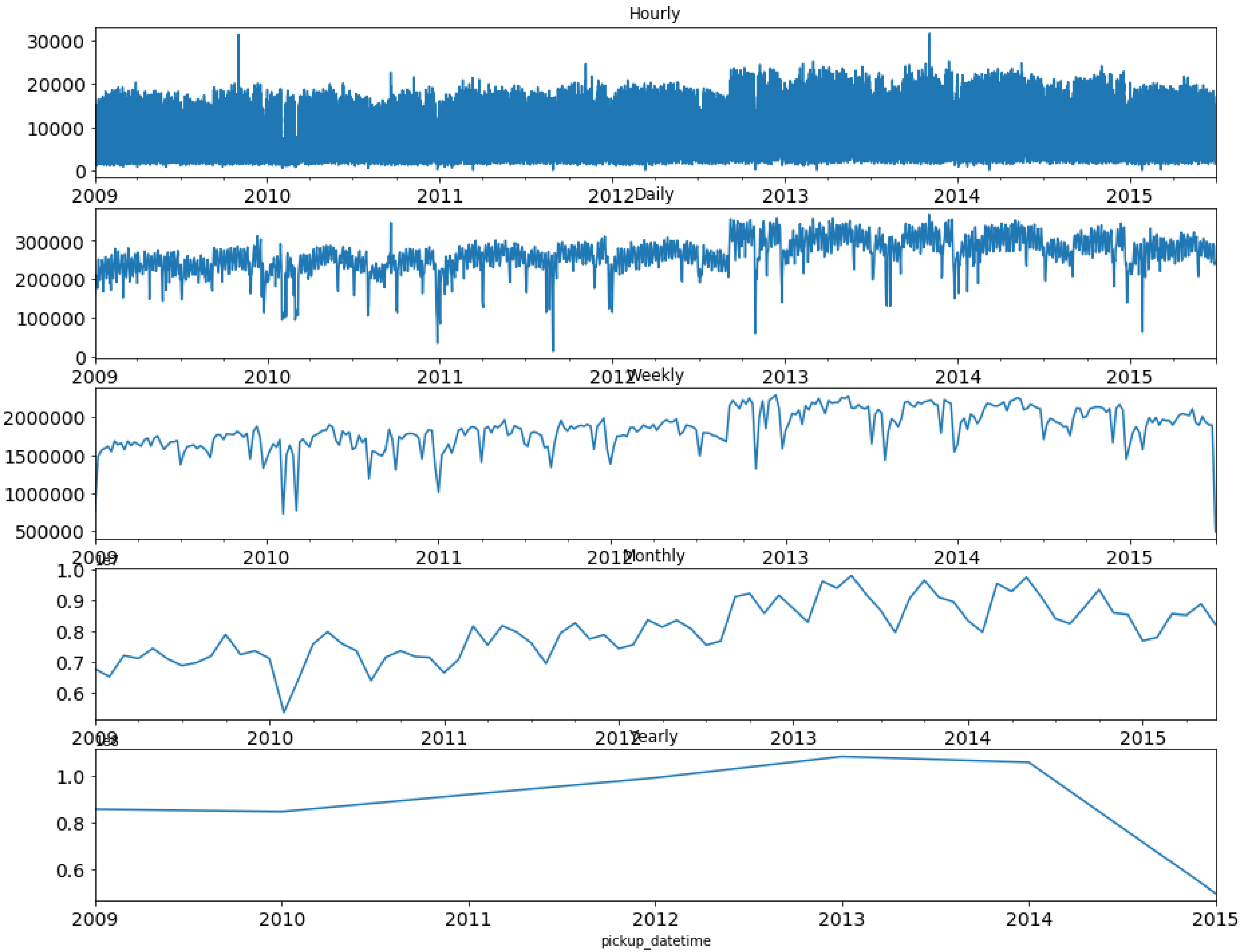
B. EDA

Data Transformation

Let's look at the hourly, daily, weekly, monthly time series

resample('H', 'D', 'W', 'M').sum() : Aggregate the hourly time series to daily, weekly, monthly time series to reduce the noise and make it more stable

We will aggregating it on daily, then work on the daily time series



B. EDA

Data Transformation

```
In [31]: train = train.resample('D').sum().fillna(method='ffill')[['fare_amount']] # keep fare_amount only
    ...:

In [32]: train.head()
Out[32]:
                 fare_amount
pickup_datetime
2009-01-01     163501.296875
2009-01-02     184812.187500
2009-01-03     211485.343750
2009-01-04     191543.656250
2009-01-05     190128.015625
```

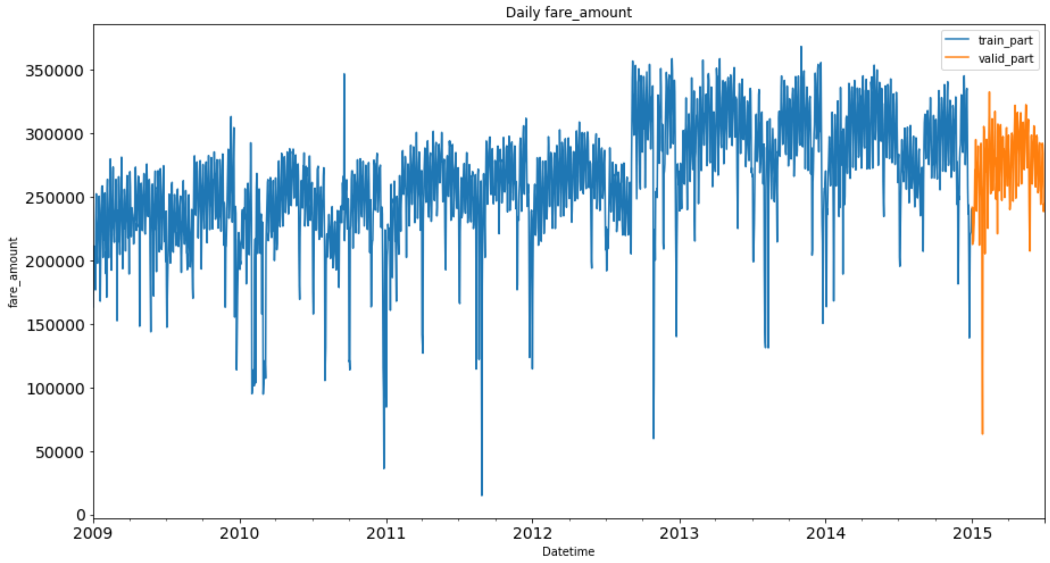
Let's look at the hourly, daily, weekly, monthly time series
resample('H', 'D', 'W', 'M').sum() : Aggregate the hourly time series to daily, weekly, monthly time series to reduce the noise and make it more stable
We will aggregating it on daily, then work on the daily time series

C. MODEL SELECTION

Splitting the data into training and validation part

```
In [33]: train_part = train.ix['2009-01-01':'2014-12-31']
    ...: valid_part = train.ix['2015-01-01':'2015-06-30']
    ...: # will look at how the train and validation part has been divided
    ...: train_part['fare_amount'].plot(figsize=(15,8), title= 'Daily fare_amount', fontsize=14, label='train_part')
    ...: valid_part['fare_amount'].plot(figsize=(15,8), title= 'Daily fare_amount', fontsize=14, label='valid_part')
    ...: plt.xlabel('Datetime')
    ...: plt.ylabel('fare_amount')
    ...: plt.legend(loc='best')
    ...: plt.show()
```

We will take the last 6 months as the validation data and rest for training data
6 months as the trend, seasonality will be the most in them



C. MODEL SELECTION

The difference between TSA and other techniques

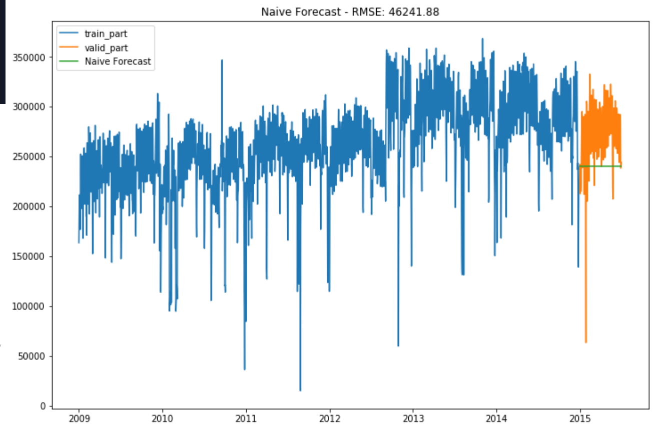
What differentiates a time series from **regular regression problem data** is that the **observations are time dependent and, along with an increasing or decreasing trend**, many time series exhibit seasonal trends. The **TSA technique can seek to model these trends in data over time and then bring these trends into the future forecast**.

We will look at **various models** now to forecast the time series; then select the best model

C. MODEL SELECTION

Naïve approach

```
In [34]: y_hat = valid_part.copy()
    ...: naive_value = train_part['fare_amount'][-1]
    ...: y_hat['naive'] = naive_value

In [35]: y_hat = valid_part.copy()
    ...: naive_value = train_part['fare_amount'][-1]
    ...: y_hat['naive'] = naive_value
    ...: # calculate RMSE to check the accuracy of our model on validation data set
    ...: RMSE_naive = sqrt(mean_squared_error(valid_part['fare_amount'], y_hat['naive']))
    ...:
    ...: plt.figure(figsize=(12,8))
    ...: plt.plot(train_part.index, train_part['fare_amount'], label='train_part')
    ...: plt.plot(valid_part.index, valid_part['fare_amount'], label='valid_part')
    ...: plt.plot(y_hat.index, y_hat['naive'], label='Naive Forecast')
    ...: plt.legend(loc='best')
    ...: plt.title('Naive Forecast - RMSE: {:.2f}'.format(RMSE_naive))
    ...: plt.show()
```



assumption is that the next expected point is equal to the last observed point.
Calc RMSE
Issue: This technique doesn't predict based on old values

C. MODEL SELECTION

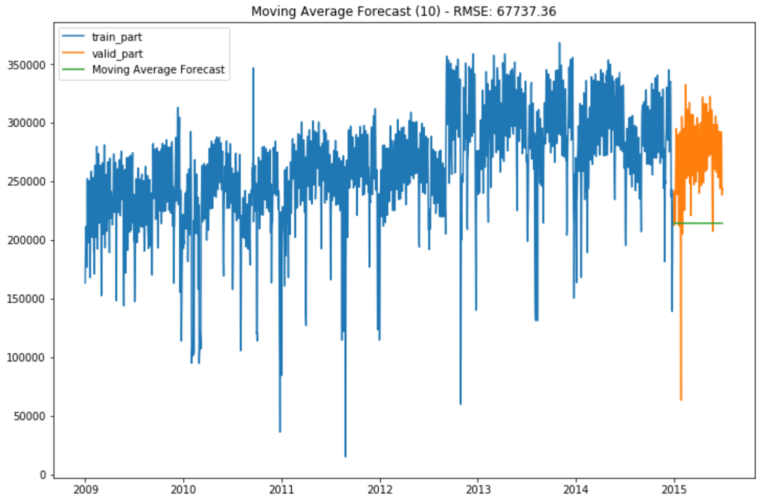
Moving Average

```
In [36]: y_hat = valid_part.copy()
    ...: y_hat['moving_avg_forecast'] = train_part['fare_amount'].rolling(10).mean().iloc[-1] # average of last 10 observations.
    ...: RMSE_moving_avg = sqrt(mean_squared_error(valid_part['fare_amount'], y_hat['moving_avg_forecast']))
    ...:
    ...: plt.figure(figsize=(12,8))
    ...: plt.plot(train_part.index, train_part['fare_amount'], label='train_part')
    ...: plt.plot(valid_part.index, valid_part['fare_amount'], label='valid_part')
    ...: plt.plot(y_hat.index, y_hat['moving_avg_forecast'], label='Moving Average Forecast')
    ...: plt.legend(loc='best')
    ...: plt.title('Moving Average Forecast (10) - RMSE: {:.2f}'.format(RMSE_moving_avg))
    ...: plt.show()
```

We will take the average of fare_amount for last few time periods only

We took the average of last 10 upto 60 observations and predicted based on that. Then we find the lowest RMSE value.

Issue: This technique only work well on stable time series, it doesn't bring the trend into model.



C. MODEL SELECTION

Moving Average

```
In [37]: y_hat = valid_part.copy()
    ...: rolling_para = []
    ...: RMSE = []
    ...: for i in range(10,60,1):
    ...:     y_hat['moving_avg_forecast'] = train_part['fare_amount'].rolling(i).mean().iloc[-1]
    ...:     rmse = sqrt(mean_squared_error(valid_part['fare_amount'], y_hat['moving_avg_forecast']))
    ...:     rolling_para.append(i)
    ...:     RMSE.append(rmse)
    ...:
    ...: RMSE_table = pd.DataFrame(list(zip(rolling_para, RMSE)), columns=['rolling_para', 'RMSE'])
    ...: RMSE_table[RMSE_table['RMSE'] == RMSE_table['RMSE'].min()]
Out[37]:
    rolling_para          RMSE
22            32  31697.679221
```

We will take the average of fare_amount for last few time periods only

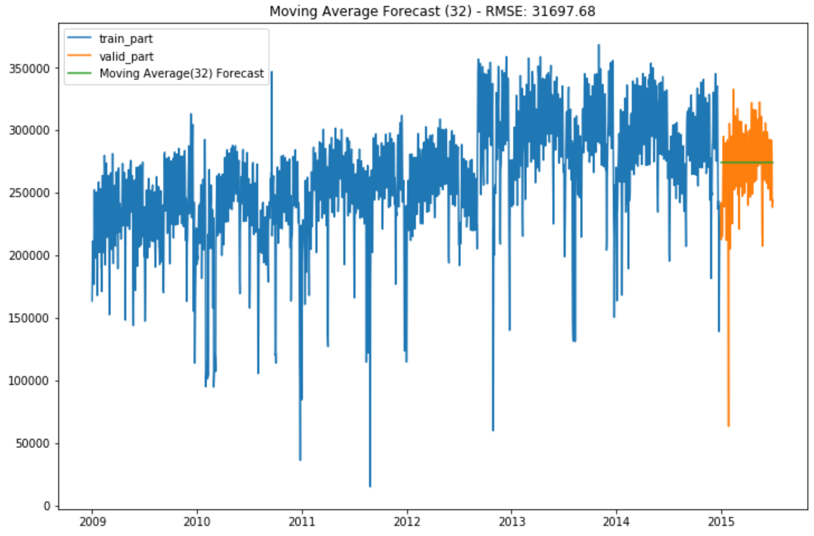
We took the average of last 10 upto 60 observations and predicted based on that. Then we find the lowest RMSE value.

Issue: This technique only work well on stable time series, it doesn't bring the trend into model.

C. MODEL SELECTION

Moving Average

```
In [38]: y_hat['moving_avg_forecast'] = train_part['fare_amount'].rolling(32).mean().iloc[-1] # average of last 32 observations.
    ...: RMSE_moving_avg = sqrt(mean_squared_error(valid_part['fare_amount'], y_hat['moving_avg_forecast']))
    ...:
    ...: plt.figure(figsize=(12,8))
    ...: plt.plot(train_part.index, train_part['fare_amount'], label='train_part')
    ...: plt.plot(valid_part.index, valid_part['fare_amount'], label='valid_part')
    ...: plt.plot(y_hat.index, y_hat['moving_avg_forecast'], label='Moving Average(32) Forecast')
    ...: plt.legend(loc='best')
    ...: plt.title('Moving Average Forecast (32) - RMSE: {:.2f}'.format(RMSE_moving_avg))
    ...: plt.show()
```



We will take the average of fare_amount for last few time periods only

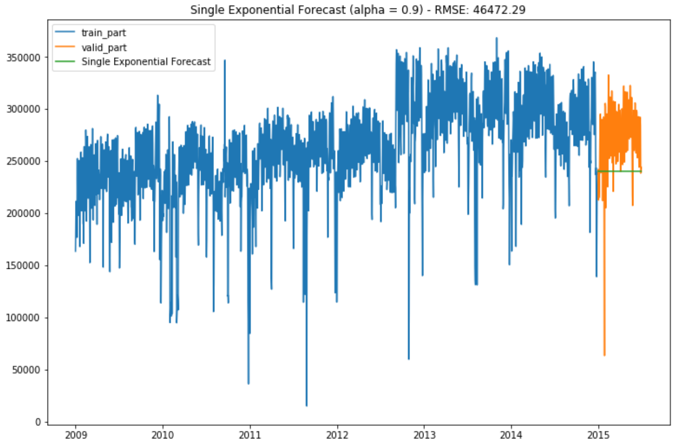
We took the average of last 10 upto 60 observations and predicted based on that. Then we find the lowest RMSE value.

Issue: This technique only work well on stable time series, it doesn't bring the trend into model.

C. MODEL SELECTION

Single Exponential Smoothing

```
In [39]: y_hat = valid_part.copy()
    ...: single_exp = SimpleExpSmoothing(np.asarray(train_part['fare_amount'])).fit(smoothing_level=0.9,
    ...:                                  optimized=False)
    ...: y_hat['single_exp'] = single_exp.forecast(len(valid_part))
    ...: RMSE_single_exp = sqrt(mean_squared_error(valid_part['fare_amount'], y_hat['single_exp']))

In [40]: plt.figure(figsize=(12,8))
    ...: plt.plot(train_part.index, train_part['fare_amount'], label='train_part')
    ...: plt.plot(valid_part.index, valid_part['fare_amount'], label='valid_part')
    ...: plt.plot(y_hat.index, y_hat['single_exp'], label='Single Exponential Forecast')
    ...: plt.legend(loc='best')
    ...: plt.title('Single Exponential Forecast (alpha = 0.9) - RMSE: {:.2f}'.format(RMSE_single_exp))
    ...: plt.show()
```



The idea of this method is that the predictions are made by assigning larger weight to the recent values and lesser weight to the old values

We took smoothing_level from 0.5 - 0.9 and predicted based on each of these model, then looking for lowest RSME value

Issue: This method is better than Moving Average, maybe. But there is no trend or seasonality

C. MODEL SELECTION

Single Exponential Smoothing

```
In [41]: Smoothing_level = []
    ...: RMSE = []
    ...: for i in [0.5, 0.6, 0.7, 0.8, 0.9]:
    ...:     fit = SimpleExpSmoothing(np.asarray(train_part['fare_amount'])).fit(smoothing_level=i,
    ...:                                 optimized=False)
    ...:     y_hat['single_exp'] = fit.forecast(len(valid_part))
    ...:     rmse = sqrt(mean_squared_error(valid_part['fare_amount'], y_hat['single_exp']))
    ...:     Smoothing_level.append(i)
    ...:     RMSE.append(rmse)
    ...:
    ...: RMSE_table = pd.DataFrame(list(zip(Smoothing_level, RMSE)), columns=['Smoothing_level', 'RMSE'])
    ...: RMSE_table[RMSE_table['RMSE'] == RMSE_table['RMSE'].min()]
Out[41]:
   Smoothing_level          RMSE
4              0.9  46472.294496
```

The idea of this method is that the predictions are made by assigning larger weight to the recent values and lesser weight to the old values

We took smoothing_level from 0.5 - 0.9 and predicted based on each of these model, then looking for lowest RSME value

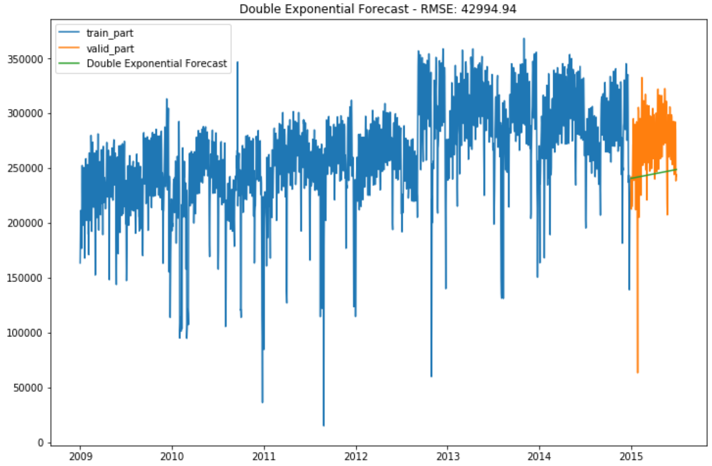
Issue: This method is better than Moving Average, maybe. But there is no trend or seasonality

C. MODEL SELECTION

Double Exponential Smoothing (also known as Holt's Linear Trend Model)

```
In [42]: y_hat = valid_part.copy()
    ...: double_exp = Holt(np.asarray(train_part['fare_amount'])).fit(optimized=True)
    ...: y_hat['double_exp'] = double_exp.forecast(len(valid_part))
    ...: RMSE_double_exp = sqrt(mean_squared_error(valid_part['fare_amount'], y_hat['double_exp']))
    ...:
    ...: plt.figure(figsize=(12,8))
    ...: plt.plot(train_part.index, train_part['fare_amount'], label='train_part')
    ...: plt.plot(valid_part.index, valid_part['fare_amount'], label='valid_part')
    ...: plt.plot(y_hat.index, y_hat['double_exp'], label='Double Exponential Forecast')
    ...: plt.legend(loc='best')
    ...: plt.title('Double Exponential Forecast - RMSE: {:.2f}'.format(RMSE_double_exp))
    ...: plt.show()
```

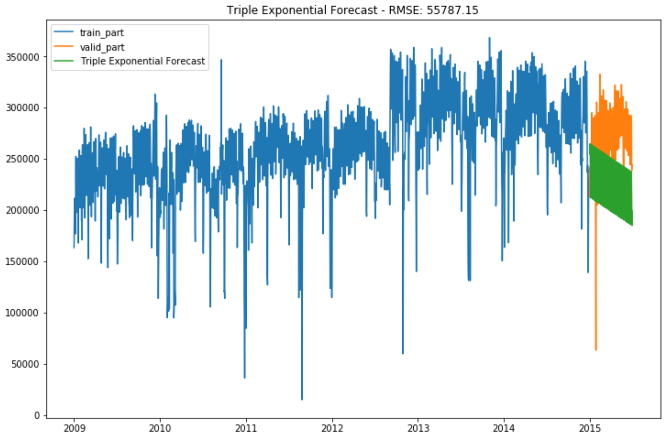
Issue: This model can pickup on trend, but no seasonality



C. MODEL SELECTION

Triple Exponential Smoothing (also known as Holt winter's model)

```
In [43]: y_hat = valid_part.copy()
    ...: triple_exp = ExponentialSmoothing(np.asarray(train_part['fare_amount']),
    ...:                                   trend="additive",
    ...:                                   seasonal="additive",
    ...:                                   seasonal_periods=7).fit(optimized=True)
    ...: y_hat['triple_exp'] = triple_exp.forecast(len(valid_part))
    ...: RMSE_triple_exp = sqrt(mean_squared_error(valid_part['fare_amount'], y_hat['triple_exp']))
    ...:
    ...: plt.figure(figsize=(12,8))
    ...: plt.plot(train_part.index, train_part['fare_amount'], label='train_part')
    ...: plt.plot(valid_part.index, valid_part['fare_amount'], label='valid_part')
    ...: plt.plot(y_hat.index, y_hat['triple_exp'], label='Triple Exponential Forecast')
    ...: plt.legend(loc='best')
    ...: plt.title('Triple Exponential Forecast - RMSE: {:.2f}'.format(RMSE_triple_exp))
    ...: plt.show()
```



The idea behind Holt's Winter is to apply exponential smoothing to the seasonal components in addition to level and trend

C. MODEL SELECTION

ARIMA model

(also known as the Box-Jenkins approach)

=> 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?

ARIMA **model**

Check stationary Dickey-Fuller Test

```
In [11]: import statsmodels.tsa.stattools as ts
    ...: def dftest(timeseries, t=30):
    ...:     dftest = ts.adfuller(timeseries, autolag='AIC')
    ...:     dfoutput = pd.Series(dftest[0:4],
    ...:                          index=['Test Statistic','p-value','Lags Used','Observations Used'])
    ...:     for key,value in dftest[4].items():
    ...:         dfoutput['Critical Value (%s)'%key] = value
    ...:     print(dfoutput)
    ...:     #Determing rolling statistics
    ...:     rolmean = timeseries.rolling(window=t).mean()
    ...:     rolstd = timeseries.rolling(window=t).std()
    ...:     #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 and Standard Deviation')
    ...:     plt.grid()
    ...:     plt.show(block=False)
```

The null hypothesis is that the TS is non-stationary; Alternative hypothesis: the series is stationary
We generally say that the series is stationary if the p-value is less than 0.05
Or If the ‘Test Statistic’ is less than the ‘Critical Value’, we can reject the null hypothesis and say that the series is stationary
Dickey-Fuller test with difference window size (30, 15, 7 days)

ARIMA model

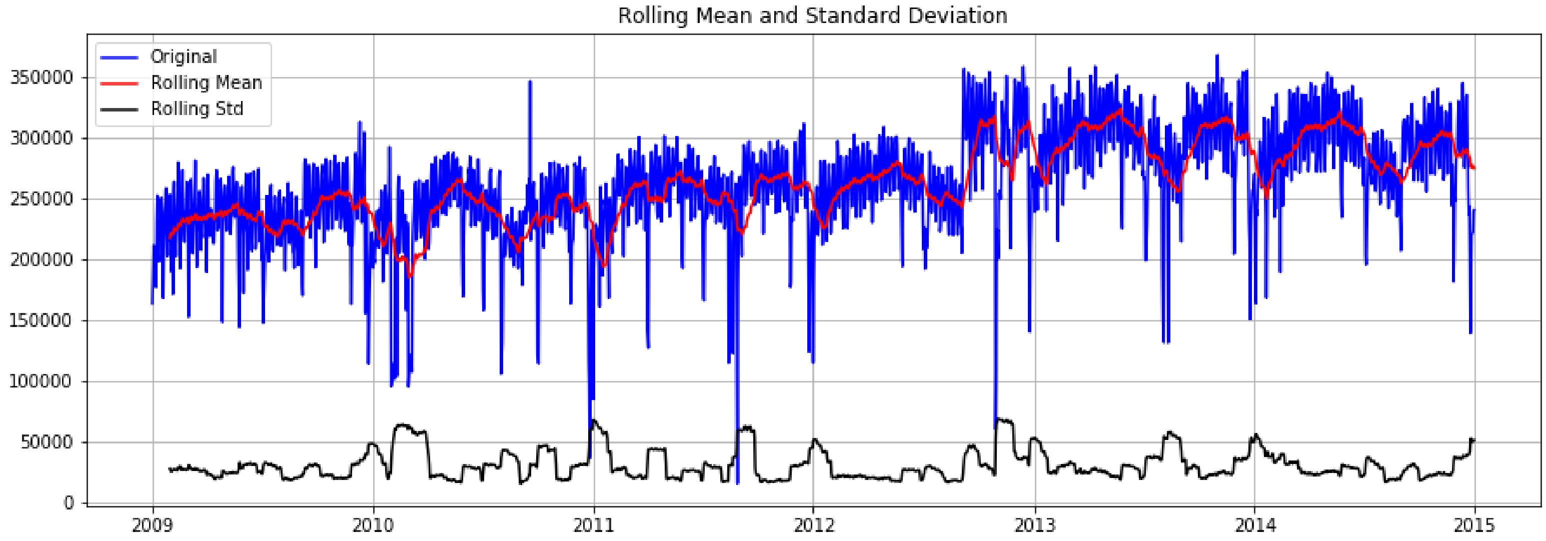
Check stationary Dickey-Fuller Test

```
In [47]: dftest(train_part['fare_amount'], 30)
Test Statistic                -3.166025
p-value                        0.022037
Lags Used                     26.000000
Observations Used           2164.000000
Critical Value (1%)           -3.433375
Critical Value (5%)           -2.862877
Critical Value (10%)          -2.567482
dtype: float64
```

We generally say that the series is stationary if the p-value is less than 0.05

Or If the 'Test Statistic' is less than the 'Critical Value', we can reject the null hypothesis and say that the series is stationary

Dickey-Fuller test with difference window size (30, 15, 7 days)



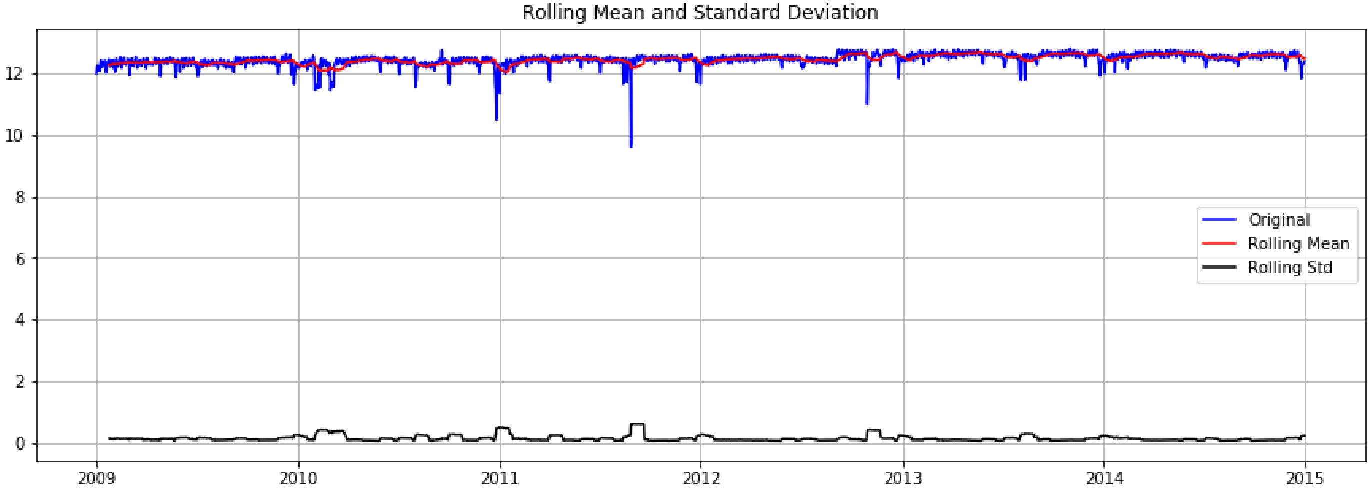
ARIMA **model**

Make a Time Series Stationary

```
In [18]: train_part_log = np.log(train_part['fare_amount'])

In [19]: dftest(train_part_log, 24)
Test Statistic                  -3.802526
p-value                          0.002882
Lags Used                       26.000000
Observations Used             2164.000000
Critical Value (1%)             -3.433375
Critical Value (5%)             -2.862877
Critical Value (10%)            -2.567482
dtype: float64
```

There is 4 methods of checking Stationary: **Transformation**; **Differencing**; Removing trend by using **Moving Average Smoothing** or **Exponentially Weighted Moving Average**; **Decomposition**

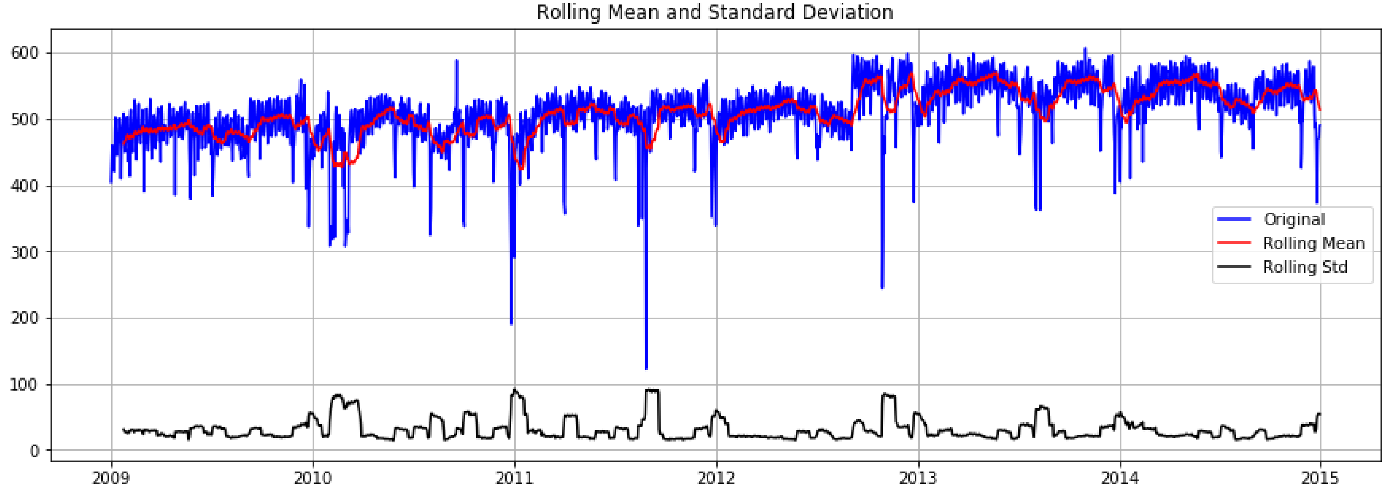
But we'll using Transformation; Differencing (cuz: the other method is hard to be scaled back to the original values)



ARIMA **model**

Make a Time Series Stationary

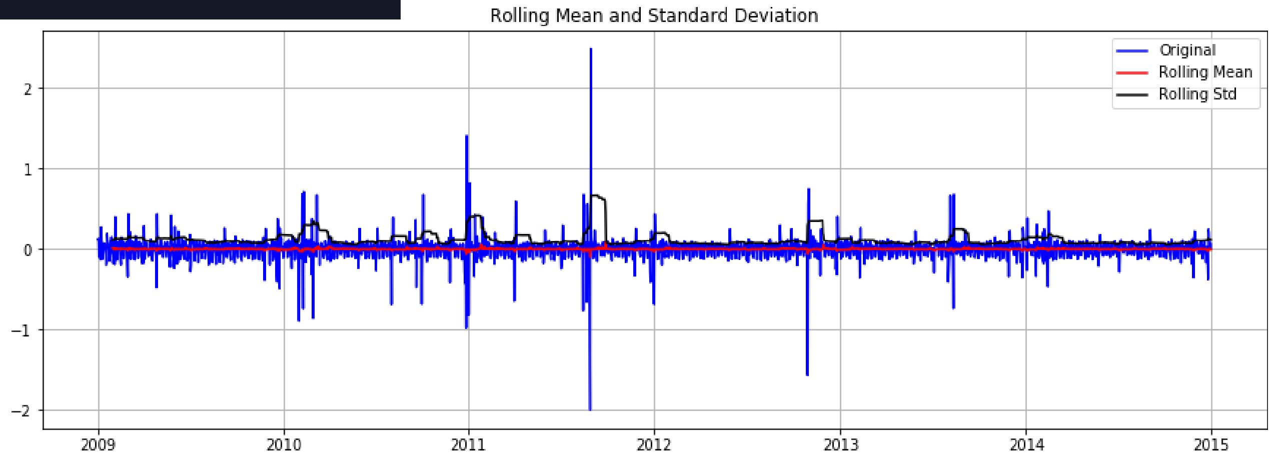
```
In [20]: train_part_sqrt = np.sqrt(train_part['fare_amount'])

In [21]: dftest(train_part_sqrt, 24)
Test Statistic                -3.393908
p-value                        0.011168
Lags Used                     26.000000
Observations Used           2164.000000
Critical Value (1%)           -3.433375
Critical Value (5%)           -2.862877
Critical Value (10%)          -2.567482
dtype: float64
```



ARIA model

Differencing on the Log Transformation of the Time Series

```
In [22]: train_part_log_diff = train_part_log - train_part_log.shift(1)

In [23]: train_part_log_diff.dropna(inplace=True)

In [24]: dftest(train_part_log_diff)
Test Statistic             -1.717438e+01
p-value                     6.710614e-30
Lags Used                   2.600000e+01
Observations Used           2.163000e+03
Critical Value (1%)        -3.433377e+00
Critical Value (5%)        -2.862877e+00
Critical Value (10%)       -2.567482e+00
dtype: float64
```

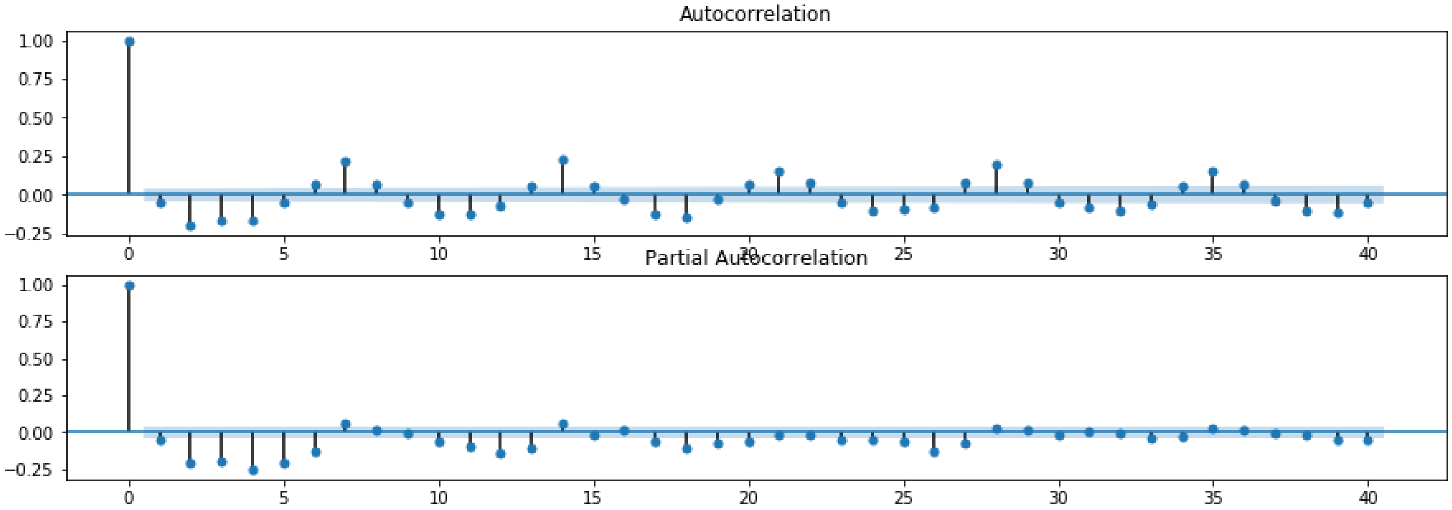


ARIMA **model**

Deciding Upon the Parameters for Modeling | ACF and PACF plot

```
In [25]: ax1 = plt.subplot(211)
    ...: fig = sm.graphics.tsa.plot_acf(train_part_log_diff.squeeze(), lags=40, ax=ax1)
    ...: ax2 = plt.subplot(212)
    ...: fig = sm.graphics.tsa.plot_pacf(train_part_log_diff, lags=40, ax=ax2)
```

We wasn't sure that what is the parameter of the ACF and PACF



ARIMA **model**

Finding parameters of the ACF and PACF

```
In [26]: para_set = [(1, 1, 1), (2, 1, 1), (1, 1, 2), (2, 1, 0), (0, 1, 2)]
    ...: data_value = train_part_log
    ...: for i in para_set:
    ...:     model = ARIMA(data_value, order=i)
    ...:     results_ARIMA = model.fit(disp=-1)
    ...:     # Measuring the quality of the model using AIC
    ...:     AIC = results_ARIMA.aic
    ...:     print('ARIMA {} - AIC: {}'.format(i, AIC))
    ...:
    ...:
ARIMA (1, 1, 1) - AIC: -2514.039253369927
ARIMA (2, 1, 1) - AIC: -2567.817398555848
ARIMA (1, 1, 2) - AIC: -2561.3745087410734
ARIMA (2, 1, 0) - AIC: -2213.240014179304
ARIMA (0, 1, 2) - AIC: -2436.8956954862624
```

Finding the optimal values for the ARIMA(p,d,q) model by try and error method (training on train_part)

Measuring the quality of the model: AIC

ARIMA **model**

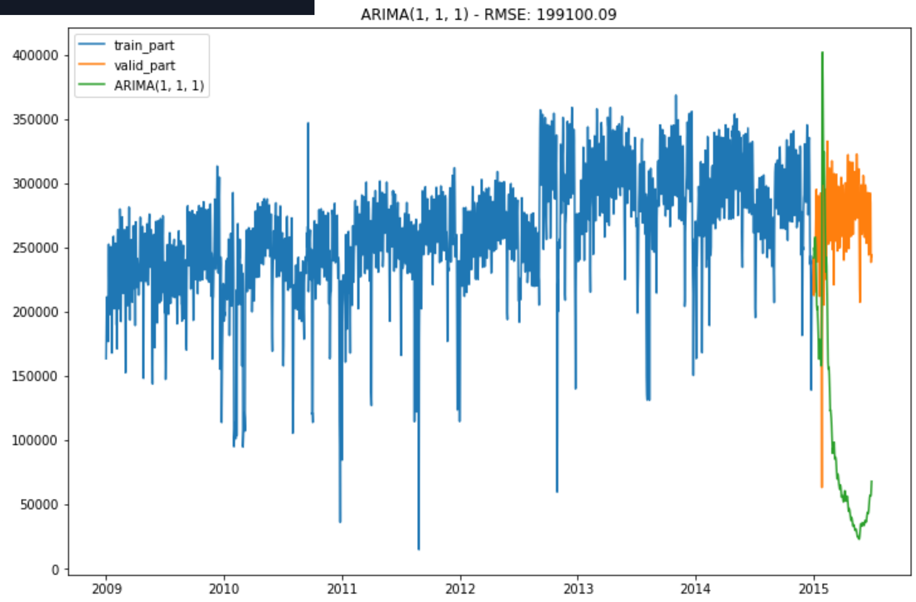
Scaling back the Forecast values

```
In [48]: valid_part_log = np.log(valid_part['fare_amount'])
    ...: model = ARIMA(valid_part_log, order=(1, 1, 1))
    ...: results_ARIMA_111 = model.fit(disp=-1)
    ...: predict_valid_ARIMA_diff = pd.Series(results_ARIMA_111.fittedvalues, copy=True)
    ...: predict_valid_ARIMA_diff_cumsum = predict_valid_ARIMA_diff.cumsum()
    ...: # assumed the first value of the log transformed time series to be the base value
    ...: predict_valid_ARIMA_log = pd.Series(valid_part_log.iloc[0], index=valid_part_log.index)
    ...: predict_valid_ARIMA_log = predict_valid_ARIMA_log.add(predict_valid_ARIMA_diff_cumsum,
    ...:                                                   fill_value=0)
    ...: predict_valid_ARIMA = np.exp(predict_valid_ARIMA_log)
    ...: RMSE_ARIMA = RMSE(valid_part['fare_amount'], predict_valid_ARIMA)
    ...:
```

ARIMA **model**

Scaling back the Forecast values

```
In [49]: plt.figure(figsize=(12,8))
    ...: plt.plot(train_part.index, train_part['fare_amount'], label='train_part')
    ...: plt.plot(valid_part.index, valid_part['fare_amount'], label='valid_part')
    ...: plt.plot(predict_valid_ARIMA.index, predict_valid_ARIMA, label='ARIMA(1, 1, 1)')
    ...: plt.legend(loc='best')
    ...: plt.title('ARIMA(1, 1, 1) - RMSE: {:.2f}'.format(RMSE_ARIMA))
    ...: plt.show()
    ...:
```



CONCLUTIONS

This is just the beginning

1. What's next?

dataset	work on the **daily** time series	work on the **monthly** time series	work on **the short daily** time series
Model	RMSE	RMSE	RMSE
Naive approach	46,241.88	**455,445.98**	
Moving Average	**31,697.68**	522,812.17	
Single Exponential Smoothing	46,772.29	461,734.86	
Double Exponential Smoothing	42,994.94	875,668.88	
Triple Exponential Smoothing	55,787.15	797,000.81	
ARIMA model	199,100.09		

2. Come back to our first assumption…

Thanks for Watching

Xin góp ý, câu hỏi?

end ■