NYC_CAB_MODELS

June 29, 2019

In [1]:
```
#Importing Libraries
import pandas as pd #pandas to create small dataframes
import numpy as np #Do aritmetic operations on arrays

import os
from datetime import datetime
import pickle

# Visualization related packages
import matplotlib.pylab as plt
import seaborn as sns #Plots
sns.set()

# model related packages
import xgboost as xgb
from xgboost import XGBRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression

# Evaluation metric related packages
from sklearn.metrics import mean_squared_error
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_absolute_error

from sklearn.preprocessing import OneHotEncoder
import math

from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import TimeSeriesSplit

from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

# packages for presenting the results
from  prettytable import  PrettyTable
```

```
from tsfresh.feature_extraction import feature_calculators
```

1 Configs

```
In [2]: print(datetime.now() ,' Started')
        base_dir = './data'
        kmeans_path = './model/NYC_kmeans_model.pickle'
        test_size = 0.3 # train, test partitioning threshold

        # set sample size here for testing the code
        sample_size = -1 # set -1 if we want to use full dataset
```

```
2019-06-28 09:18:17.825627  Started
```

2 UTIL Functions

```
In [3]: def compute_mape(actual, predicted):

            # compute MAPE error
            try:
                mape = mean_absolute_error(actual, predicted) / np.mean(actual) * 100.0
                mape = round(mape, 4)
            except ZeroDivisionError:
                print('Division by zero error in MAPE')
                mape = np.inf
            except:
                print('Exception in computing MAPE value')
                mape = np.inf

            return mape
```

```
In [4]: def evaluate_model(actual_values_list, predicted_values_list):

            # get MSE, MAPE error metrics
            mse = round(mean_squared_error(actual_values_list, predicted_values_list), 4)
            mape = round(compute_mape(actual_values_list, predicted_values_list), 4)

            return (mse, mape,)
```

3 Data

3.0.1 2015 Data

```
In [5]: # Read Jan, Feb & March data
        df_jan_2015 = pd.read_csv(os.path.join(base_dir, 'ts_data_jan_2015.csv'), index_col=Fals
        df_feb_2015 = pd.read_csv(os.path.join(base_dir, 'ts_data_feb_2015.csv'), index_col=Fals
```

```
df_mar_2015 = pd.read_csv(os.path.join(base_dir, 'ts_data_mar_2015.csv'), index_col=Fals

shape_info = df_jan_2015.shape + df_feb_2015.shape + df_mar_2015.shape

print('Shape of data frames Jan:(%d,%d), Feb:(%d,%d) & March:(%d,%d)'%shape_info)

# concatenate three months data
df_2015 = pd.concat([df_jan_2015, df_feb_2015, df_mar_2015], axis=1)
df_2015.columns = range(0, df_2015.shape[1])
print(df_2015.shape)
df_2015.head()
```

```
Shape of data frames Jan:(30,4464), Feb:(30,4032) & March:(30,4464)
(30, 12960)
```

Out[5]:

	0	1	2	3	4	5	6	7	8	9	\
0	15	15	114	141	144	200	145	184	190	225	
1	54	54	228	266	248	262	278	270	314	275	
2	15	15	47	42	47	21	15	33	16	27	
3	18	18	26	18	14	5	13	19	7	16	
4	58	58	240	220	157	145	130	189	149	133	

	...	12950	12951	12952	12953	12954	12955	12956	12957	12958	12959
0	...	60.0	52.0	48.0	66.0	54.0	36.0	68.0	46.0	29.0	18.0
1	...	116.0	163.0	79.0	132.0	116.0	120.0	125.0	149.0	119.0	118.0
2	...	87.0	71.0	90.0	98.0	76.0	55.0	98.0	72.0	40.0	67.0
3	...	87.0	104.0	139.0	84.0	86.0	68.0	79.0	61.0	50.0	42.0
4	...	178.0	174.0	214.0	208.0	118.0	166.0	164.0	185.0	135.0	135.0

[5 rows x 12960 columns]

3.0.2 2016 Data

In [6]:
```
# Read Jan, Feb & March data
df_jan_2016 = pd.read_csv(os.path.join(base_dir, 'ts_data_jan_2016.csv'), index_col=Fals
df_feb_2016 = pd.read_csv(os.path.join(base_dir, 'ts_data_feb_2016.csv'), index_col=Fals
df_mar_2016 = pd.read_csv(os.path.join(base_dir, 'ts_data_mar_2016.csv'), index_col=Fals

shape_info = df_jan_2016.shape + df_feb_2016.shape + df_mar_2016.shape

print('Shape of data frames Jan:(%d,%d), Feb:(%d,%d) & March:(%d,%d)'%shape_info)

# concatenate three months data
df_2016 = pd.concat([df_jan_2016, df_feb_2016, df_mar_2016], axis=1)
df_2016.columns = range(0,df_2016.shape[1])
print(df_2016.shape)
df_2016.head()
```

```
Shape of data frames Jan:(30,4464), Feb:(30,4176) & March:(30,4464)
(30, 13104)
```

Out[6]:

	0	1	2	3	4	5	6	7	8	9	\
0	0	32	110	145	160	176	157	155	156	161	
1	0	96	204	228	257	222	228	268	260	236	
2	0	28	27	37	21	22	18	27	17	15	
3	0	9	33	16	8	11	4	8	11	20	
4	0	79	171	168	125	163	138	172	153	135	

	...	13094	13095	13096	13097	13098	13099	13100	13101	13102	13103
0	...	92	96	92	74	59	88	76	76	73	65
1	...	193	163	171	165	170	178	180	166	132	155
2	...	76	94	93	92	74	80	67	51	63	66
3	...	133	123	135	139	137	131	127	148	84	94
4	...	160	187	164	171	176	194	190	160	168	144

[5 rows x 13104 columns]

2016 is a leap year and FEB has 29 days
We can discard the last day of March 2016 to make the dataset sizes equal

```
In [7]: num_days = min(df_2015.shape[1], df_2016.shape[1])
        df_2015 = df_2015.iloc[:,0:num_days]
        df_2016 = df_2016.iloc[:,0:num_days]
        print('Shape of DF 2015: ', df_2015.shape, 'Shape of DF 2016: ',df_2016.shape)
```

```
Shape of DF 2015:  (30, 12960) Shape of DF 2016:  (30, 12960)
```

```
In [8]: print('Minimum pickup value in df_2015 DF', df_2015.min().min())
        print('Minimum pickup value in df_2016 DF', df_2016.min().min())
```

```
Minimum pickup value in df_2015 DF 1.0
Minimum pickup value in df_2016 DF 0
```

```
In [9]: print('Maximum pickup value in df_2015 DF', df_2015.max().max())
        print('Maximum pickup value in df_2016 DF', df_2016.max().max())
```

```
Maximum pickup value in df_2015 DF 681.0
Maximum pickup value in df_2016 DF 645
```

4 A) Time Series Baseline Models

4.0.1 Train, Test Split & Optional Sample Data

```
In [10]: if sample_size > 0:
             print('Sample is taken !!!')
```

```
        # sample of
        df_2015 = df_2015.iloc[0:sample_size, :]
        df_2016 = df_2016.iloc[0:sample_size, :]

    # train test split

    # partition data to train, test
    split_point = int((1-test_size) * df_2016.shape[1])
    print('Split point :', split_point)

    # split to train & test data (2015)
    df_2015_train = df_2015.iloc[:, 0:split_point]
    df_2015_train.columns = range(0, df_2015_train.shape[1])
    df_2015_test = df_2015.iloc[:, split_point:]
    df_2015_test.columns = range(0, df_2015_test.shape[1])
    print('Data Frmae 2015 train shapes :', df_2015_train.shape,)
    print('Data Frmae 2015 test shapes :', df_2015_test.shape,)

    # split to train & test data (2016)
    df_2016_train = df_2016.iloc[:, 0:split_point]
    df_2016_train.columns = range(0, df_2016_train.shape[1])
    df_2016_test = df_2016.iloc[:, split_point:]
    df_2016_test.columns = range(0, df_2016_test.shape[1])
    print('Data Frmae 2016 train shapes :', df_2016_train.shape,)
    print('Data Frmae 2016 test shapes :', df_2016_test.shape,)

    print('\n'*2)

    ####### Preparing ratio DF
    ratio_df_train = df_2016_train / df_2015_train
    # replace 0/0 with 1.0
    print('Any invalid entry in the ratio df train:',
          ratio_df_train.isin([np.inf, -np.inf, None]).any().any())
    ratio_df_train = ratio_df_train.fillna(0.0)


    ratio_df_test = df_2016_test / df_2015_test
    # replace 0/0 with 1.0
    print('Any inf entry in the ratio df test:',
          ratio_df_test.isin([np.inf, -np.inf, None]).any().any())
    ratio_df_test = ratio_df_test.fillna(0.0)

    print('Ratio DF Train data frame size :', ratio_df_train.shape)
    print('Ratio DF Test data frame size :', ratio_df_test.shape)
```

```
Split point : 9072
Data Frmae 2015 train shapes : (30, 9072)
Data Frmae 2015 test shapes : (30, 3888)
```

```
Data Frmae 2016 train shapes : (30, 9072)
Data Frmae 2016 test shapes : (30, 3888)
```

```
Any invalid entry in the ratio df train: False
Any inf entry in the ratio df test: False
Ratio DF Train data frame size : (30, 9072)
Ratio DF Test data frame size : (30, 3888)
```

4.1 Baseline Model Functions

```
In [11]: def Simple_MA_Prediction(value_array, window_size):

             # if size less than 2 return as it is
             if len(value_array) < 2:
                 return value_array

             # initialize predicted value list as empty
             predicted_values = list()

             # # process each element in the value list
             for index, value in enumerate(value_array):

                # case 1: We have already made atleast window_size predictions
                 if index >=  window_size:
                     predicted_value = np.mean(value_array[index - window_size : index])

                 # case 2: We have just started prediction
                 else:
                     if index == 0:
                         predicted_value = value
                     else:
                         predicted_value = np.mean(value_array[0 : index])

                 # update the list
                 predicted_values.append(predicted_value)

             # round the values to integers
             predicted_values = np.array([int(round(item)) for item in predicted_values])


             return np.array(predicted_values)
```

```
In [12]: def WMA_Predictions(value_array, window_size):

             # if size less than 2 return as it is
```

```
            if len(value_array) < 2:
                return value_array

            # set the denominator
            denominator = (( window_size) * ( window_size + 1)) / 2

            # weights array
            window_weight_array = np.array(range(1, window_size + 1)) / denominator

            # initialize the predicted value with first element of value list
            predicted_values = list()

            # process each element in the value list
            for index, val in enumerate(value_array):

                # case 1: we have already made atleast window_size predictions
                if index >=  window_size:
                    predicted_value = np.mean(window_weight_array * value_array[index - window_

                # case 2: we have just started prediction
                else:

                    if index == 0:
                        predicted_value = val
                    else:
                        # set the denominator
                        denominator = (index * (index + 1)) / 2
                        # weights array
                        temp_weight_array = np.array(range(1, index + 1)) / denominator

                        predicted_value = np.mean(temp_weight_array * value_array[0 : index])

                # update the list
                predicted_values.append(predicted_value)

            # round the values to integers
            predicted_values = np.array([int(round(item)) for item in predicted_values])

            return predicted_values
```

In [13]:
```
def exp_weighted_MA_Predictions(value_array, alpha):

            # if size less than 2 return as it is
            if len(value_array) < 2:
                return value_array

            # initialize as empty
            predicted_values = list()
```

```
        # predict for every time step
        for index, value in enumerate(value_array):

            if index > 0:
                predicted = alpha * value_array[index-1] + (1-alpha) * predicted_values[-1]
            else:
                predicted = value_array[0]

            # update list
            predicted_values.append(predicted)

        # round the values to integers
        predicted_values = np.array([int(round(item)) for item in predicted_values])

        return predicted_values
```

In [14]:
```
def timeseries_model_prediction(df, window_size_sim, window_size_weight, alpha):

    # initialize three lists for saving three types of time series prediction
    simple_avg_prediction_list = list()
    weighted_avg_prediction_list = list()
    exp_weighted_avg_prediction_list = list()

    # predict one row at a time
    for index, row in df.iterrows():

        # get the row as numpy array
        inp_array = row.values

        # get simple moving average prediction
        simp_avg_pred = Simple_MA_Prediction(inp_array, window_size_sim)

        # get weighted moving average prediction
        weighted_avg_pred = WMA_Predictions(inp_array, window_size_weight)

        # get exponentially weighted moving average prediction
        exp_weighted_avg_pred = exp_weighted_MA_Predictions(inp_array, alpha)

        # update the lists
        simple_avg_prediction_list.append(simp_avg_pred)
        weighted_avg_prediction_list.append(weighted_avg_pred)
        exp_weighted_avg_prediction_list.append(exp_weighted_avg_pred)

    # prepare three types of prediction dfs
    simple_avg_prediction_df = pd.DataFrame(simple_avg_prediction_list)
    weighted_avg_prediction_df = pd.DataFrame(weighted_avg_prediction_list)
    exp_weighted_avg_prediction_df = pd.DataFrame(exp_weighted_avg_prediction_list)
```

```
            ret_tuple = (simple_avg_prediction_df, weighted_avg_prediction_df,
                         exp_weighted_avg_prediction_df,)


            return ret_tuple

In [15]: def find_best_hyperparam(df):

            print(datetime.now() ,' Hyperparam Tuning of time series model started')

            # try 10 different window sizes from 1 to 10
            window_size_list = list(range(1, 11))

            # try differnt alpah values from 0.3 to 0.99
            alpha_val_list = [0.20, 0.30, 0.40, 0.50, 0.65,
                              0.75, 0.82, 0.90, 0.95, 0.99]

            hyp_info_list = list()

            # evaluate each hyp value
            for window_size, alpha in zip(window_size_list, alpha_val_list):

                # predict using this hyperparam
                pred_dfs = timeseries_model_prediction(df, window_size, window_size, alpha)
                # separate out the prediction dfs
                smp_pred_df, weight_pred_df, exp_pred_df = pred_dfs

                # get actual values list
                actual_values_list = df.values.flatten()

                # get predicted values list
                predicted_values_list_sma = smp_pred_df.values.flatten()
                predicted_values_list_wma = weight_pred_df.values.flatten()
                predicted_values_list_exp = exp_pred_df.values.flatten()

                # evaluate three models on this hyperparam
                sim_mse, sim_mape = evaluate_model(actual_values_list, predicted_values_list_sm
                wm_mse, wm_mape = evaluate_model(actual_values_list, predicted_values_list_wma)
                exp_mse, exp_mape = evaluate_model(actual_values_list, predicted_values_list_ex

                # update hyp info list
                hyp_info_list.append((window_size, alpha, sim_mse, sim_mape,
                                      wm_mse, wm_mape, exp_mse, exp_mape,))

            # create the evaluation df
            eval_df = pd.DataFrame(hyp_info_list, columns=['Window', 'Alpha', 'SIM_MSE',
                                                           'SIM_MAPE', 'WM_MSE', 'WM_MAPE',
```

```
                                                    'EXP_MSE', 'EXP_MAPE'])

            # best index
            sim_model_best_index = eval_df['SIM_MAPE'].idxmin()
            weight_model_best_index = eval_df['WM_MAPE'].idxmin()
            exp_model_best_index = eval_df['EXP_MAPE'].idxmin()

            print("""Hyper params (Window  for simple, weighted MA,
                     Alpha for Exponentailly weighted model) scores df: \n\n\n""", eval_df)

            # get the best hyperparam based on the MAPE lowest value
            sim_best_window = eval_df.loc[sim_model_best_index, 'Window']
            weight_best_window = eval_df.loc[weight_model_best_index, 'Window']
            exp_best_alpha = eval_df.loc[exp_model_best_index, 'Alpha']

            print('\n'*2)

            print('Best Window Size (Hyperparam) for Simple Moiving Average: ', sim_best_window
            print('Best Window Size (Hyperparam) for Weighted Moiving Average: ', weight_best_w
            print('Best Alpha (Hyperparam) for Exp. Weighted Moiving Average: ', exp_best_alpha


            best_mse_sim = eval_df.loc[sim_model_best_index, 'SIM_MSE']
            best_mape_sim = eval_df.loc[sim_model_best_index, 'SIM_MAPE']

            best_mse_wma = eval_df.loc[weight_model_best_index, 'WM_MSE']
            best_mape_wma = eval_df.loc[weight_model_best_index, 'WM_MAPE']

            best_mse_ewma = eval_df.loc[exp_model_best_index, 'EXP_MSE']
            best_mape_ewma = eval_df.loc[exp_model_best_index, 'EXP_MAPE']


            print('\n'*2)
            print('Simple Moiving Average : Best MSE : %f, MAPE : %f'%(best_mse_sim,
                                                                       best_mape_sim,))
            print('Weighted Moiving Average : Best MSE : %f, MAPE : %f'%(best_mse_wma,
                                                                         best_mape_wma,))
            print('Exp. Weighted Moiving Average : Best MSE :%f MAPE : %f'%(best_mse_ewma,
                                                                           best_mape_ewma,))


            # prepare a return tuple
            ret_tuple = (sim_best_window, weight_best_window, exp_best_alpha,)

            print(datetime.now() ,' Hyperparam Tuning of time series model completed')

            return ret_tuple

In [16]: def evaluate_time_series_model(actual_df, pred_smp_df, pred_weight_df, pred_exp_df):
```