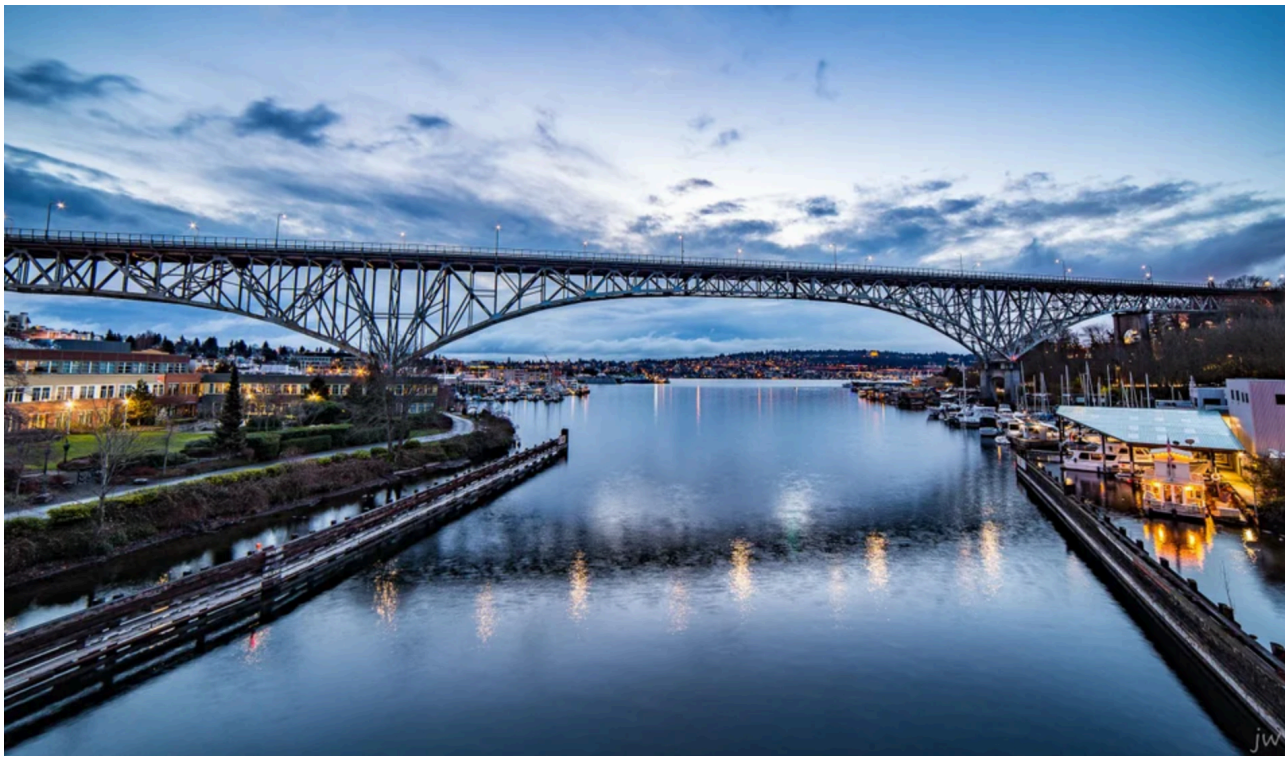
Fremont Bike Flow Daily Prediction

INFO 7390 Advances in Data Science & Architecture

2018.12.14



Team 10

Shunan Gao
Li Cai
Ying Wang

Problem Overview:

The **Fremont Bridge** is a double-leaf bascule bridge that spans the Fremont Cut in Seattle, Washington. The bridge, which connects Fremont Avenue North and 4th Avenue North, connects the neighborhoods of Fremont and Queen Anne.The bike counter installed at the Northwest corner of the Fremont Bridge on Oct 13, 2012.

Goals:

To make the prediction of the future day's bike flow. For a future day, based on weather forecast of the Seattle and the information we can extract from time to predict the future hourly bike flow. We think this information is pretty useful for the government and people who is planning to ride a bike cross the bridge.

1. Build a model use the weather data and time series information to predict the hourly bike flow from both side. West and East —-Two model

2. Select the best model and deploy it through a web application with User Friendly interface.

3. For each user input, plot two graph to show the prediction

4. Running the instance through the cloud environment Amazon EC2

Data:

Transportation : Fremont Bridge - Hourly Bicycle Counts by the month of October 2012 to present

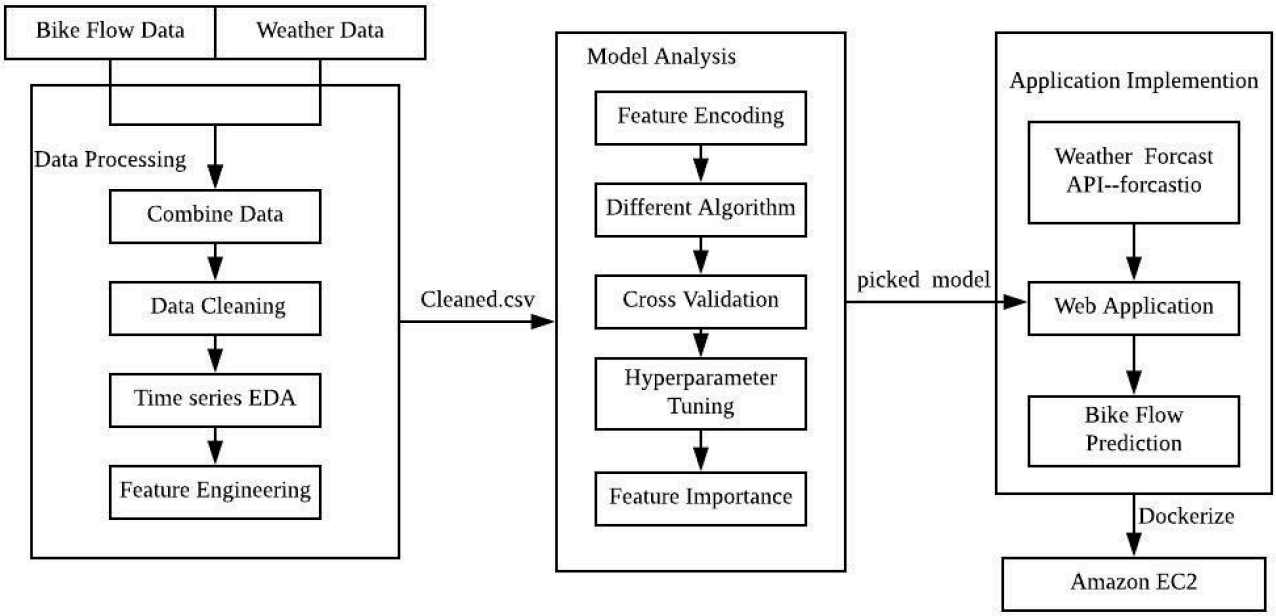
- The Fremont Bridge Bicycle Counter records the number of bikes that cross the bridge using the pedestrian/bicycle pathways. Inductive loops on the east and west pathways count the passing of bicycles regardless of travel direction. The data consists of a date/time field: Date, east pathway count field: Fremont Bridge NB, and west pathway count field: Fremont Bridge SB.

Historical Hourly Weather Data 2012-2017

- The weather data set we use is from kaggle. It contains Hourly weather data for 30 US & Canadian Cities + 6 Israeli Cities. Which has temperature, humidity, pressure, wind direction and wind speed. It's totally free and the it's hourly which is pretty ideal!

For our final project, We will use the 5 years data (from 2012-11-30 to 2017-11-30) to analysis and build the model

Pipeline Design:



Our design looks like this. It mainly contain 3 part.

- The first part is pre processing the data, we used a jupyter notebook to show how the data combined and cleaned, after that we will plot some graph to visualize the data and try to analysis each features in the data.

- The second part is model building. we will use some technique we have learned from the class to select the model we want. For different algorithms, we analysis the model performance and do the hyper parameter tuning and grid search to train the model. we will pipeline the process to show how the model was build and selected.

- The third part is web application implementation. We will use flask package to deploy our model in a Html, the user only need to enter a future day. And the application will fetch the weather forecast and preprocessing the data for the model input and the result will be shown in graph.

Implementation Details:

Preparing the data:

The first thing we need to do is to combine the weather and bike flow data set together:

In [2]:
```
#pick 5 year data
temperature = temperature.loc['2012-11-30 00:00:00':'2017-11-30 00:00:00']
pressure = pressure.loc['2012-11-30 00:00:00':'2017-11-30 00:00:00']
humidity = humidity.loc['2012-11-30 00:00:00':'2017-11-30 00:00:00']
weather_description = weather_description.loc['2012-11-30 00:00:00':'2017-11-30 00:00:00']
wind_direction = wind_direction.loc['2012-11-30 00:00:00':'2017-11-30 00:00:00']
wind_speed = wind_speed.loc['2012-11-30 00:00:00':'2017-11-30 00:00:00']
#pick Seattle
temperature = temperature.filter(items=['datetime','Seattle'])
pressure = pressure.filter(items=['datetime','Seattle'])
humidity = humidity.filter(items=['datetime','Seattle'])
weather_description = weather_description.filter(items=['datetime','Seattle'])
wind_direction = wind_direction.filter(items=['datetime','Seattle'])
wind_speed = wind_speed.filter(items=['datetime','Seattle'])
#Combine them all
weather = (temperature.merge(humidity,on='datetime').merge(pressure,on='datetime').
           merge(weather_description,on='datetime').merge(wind_direction,on='datetime').
          merge(wind_speed,on='datetime')).reset_index()
weather.columns = ['datetime','temperature','humidity','pressure','description','wind_direction','wind_speed']
weather['temperature'] = weather['temperature'] - 273.15
weather.head()
```

Out[2]:

	datetime	temperature	humidity	pressure	description	wind_direction	wind_speed
0	2012-11-30 00:00:00	7.42	87.0	1003.0	light rain	0.0	0.0
1	2012-11-30 01:00:00	7.78	87.0	1003.0	light rain	0.0	1.0
2	2012-11-30 02:00:00	8.13	87.0	1003.0	light rain	160.0	2.0
3	2012-11-30 03:00:00	8.24	64.0	1025.0	sky is clear	270.0	10.0
4	2012-11-30 04:00:00	8.66	68.0	1019.0	overcast clouds	309.0	5.0

We picked Seattle weather data and combine it with our bike flow data. Because the

Select the corresponding bike flow data and combine them

In [3]:
```
bike_flow = bike_flow.loc['11/30/2012 12:00:00 AM':'11/30/2017 12:00:00 AM']
bike_flow = bike_flow.reset_index()
```

In [4]:
```
df = pd.DataFrame()
df['datetime'] = weather['datetime']
df['east_side'] = bike_flow['Fremont Bridge East Sidewalk']
df['west_side'] = bike_flow['Fremont Bridge West Sidewalk']
df['temperature'] = weather['temperature']
df['humidity'] = weather['humidity']
df['pressure'] = weather['pressure']
df['description'] = weather['description']
df['wind_direction'] = weather['wind_direction']
df['wind_speed'] = weather['wind_speed']
df.head()
```

Out[4]:

	datetime	east_side	west_side	temperature	humidity	pressure	description	wind_direction	wind_speed
0	2012-11-30 00:00:00	2.0	5.0	7.42	87.0	1003.0	light rain	0.0	0.0
1	2012-11-30 01:00:00	1.0	0.0	7.78	87.0	1003.0	light rain	0.0	1.0
2	2012-11-30 02:00:00	1.0	0.0	8.13	87.0	1003.0	light rain	160.0	2.0
3	2012-11-30 03:00:00	1.0	0.0	8.24	64.0	1025.0	sky is clear	270.0	10.0
4	2012-11-30 04:00:00	2.0	6.0	8.66	68.0	1019.0	overcast clouds	309.0	5.0

weather data is ending at 2017 and we think 2012-2017 the 5 years data set is enough so in our project we will only use the date from 2012-11-30 to 2017-11-30.

After combining the dataset we also want to add more columns to have a better view of the data. We think from ‘date time ‘ column we can extract more useful information.

```
df = df.assign(Day_of_Week = dayOfWeek)
df['Day_of_Week'] = df['Day_of_Week'].astype('category')
df['Day_of_Week'].cat.categories = ["Mon", "Tue", "Wed", "Thu", "Fri","Sat","Sun"])
df = df.assign(Hour = hour)
df['Hour'] = df['Hour'].astype('category')
df.head()
```

st_side	west_side	temperature	humidity	pressure	description	wind_direction	wind_speed	Work_Status	Day_of_Week	Hour
2.0	5.0	7.42	87.0	1003.0	light rain	0.0	0.0	1.0	Fri	0.0
1.0	0.0	7.78	87.0	1003.0	light rain	0.0	1.0	1.0	Fri	1.0
1.0	0.0	8.13	87.0	1003.0	light rain	160.0	2.0	1.0	Fri	2.0
1.0	0.0	8.24	64.0	1025.0	sky is clear	270.0	10.0	1.0	Fri	3.0
2.0	6.0	8.66	68.0	1019.0	overcast clouds	309.0	5.0	1.0	Fri	4.0

We use a holidays package to see if the day is holiday and we assume that the working status of that day will influence the bike flow and different weekday the bike flow maybe different and for each day different hour may perform differently too.

```
In [8]: df.isnull().sum()
```

```
Out[8]: datetime          0
        east_side         8
        west_side         8
        temperature       2
        humidity        262
        pressure         11
        description       0
        wind_direction    0
        wind_speed        0
        Work_Status       0
        Day_of_Week       0
        Hour              0
        dtype: int64
```

We also find that the dataset contains some missing value and we decide to replace it with the mean of that day. We first define a new columns 'Date' and group it to get the mean value and then replace Non value with it

Filling NaN value with mean value of the Date

```
#filling missing value by the mean of that day
df1 = df.groupby([df.Date]).transform(lambda x: x.fillna(x.mean()))
```

```
df1.isnull().sum()
```

```
east_side         0
west_side         0
temperature       0
humidity          0
pressure          0
wind_direction    0
wind_speed        0
Work_Status       0
dtype: int64
```

Now that we don't have the missing value. Next step is to decide which feature should we select for further analysis. We think the date time columns is redundant so we will remove that and we will rearrange the sequence to view the dataset better.

	date	hour	east_side	west_side	temperature	humidity	pressure	description	wind_direction	wind_speed	work_status
0	2012-11-30	0.0	2.0	5.0	7.42	87.0	1003.0	light rain	0.0	0.0	1.0
1	2012-11-30	1.0	1.0	0.0	7.78	87.0	1003.0	light rain	0.0	1.0	1.0
2	2012-11-30	2.0	1.0	0.0	8.13	87.0	1003.0	light rain	160.0	2.0	1.0
3	2012-11-30	3.0	1.0	0.0	8.24	64.0	1025.0	sky is clear	270.0	10.0	1.0
4	2012-11-30	4.0	2.0	6.0	8.66	68.0	1019.0	overcast clouds	309.0	5.0	1.0

We think that the 'date' columns can not better describe the time status of the data so we will include a season columns

```
]: season = []
for i in list(range(0,43825)):
    if result.iloc[i]['date'].month in [1,2,3,4,12]:
        season.append('Winter')
    if result.iloc[i]['date'].month in [5,6]:
        season.append('Spring')
    if result.iloc[i]['date'].month in [7,8,9]:
        season.append('Summer')
    if result.iloc[i]['date'].month in [10,11]:
        season.append('Fall')
result = result.assign(season = season)
result['season'] = result['season'].astype('category')
```

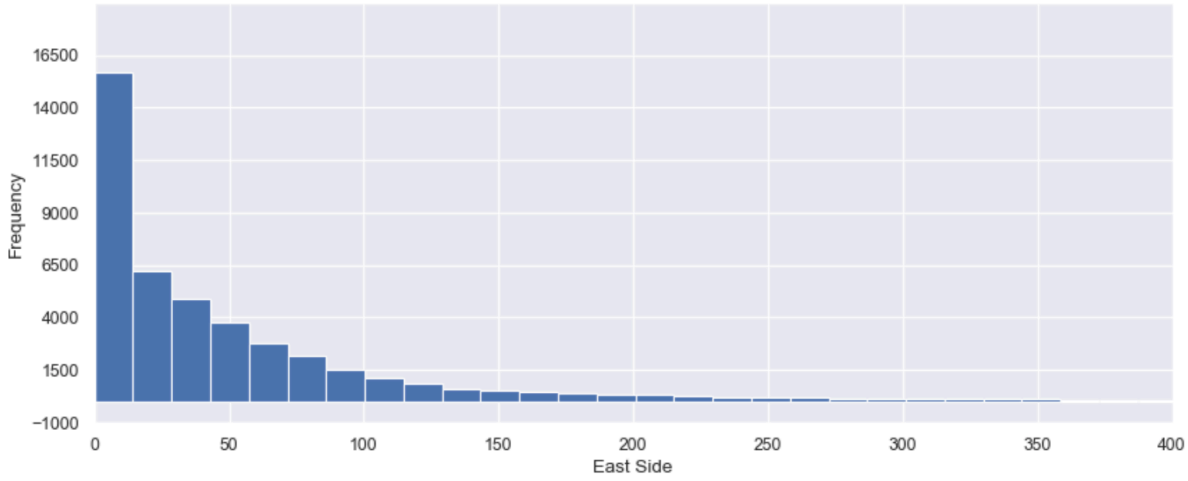
We google the how the Seattle 4 seasons divided and add the feature.
We also think for the model the 'description' feature contains the information that is not suitable for the prediction model so we will drop it . Also now that we have the season, the date columns is also no longer need the final cleaned dataset looks like this:

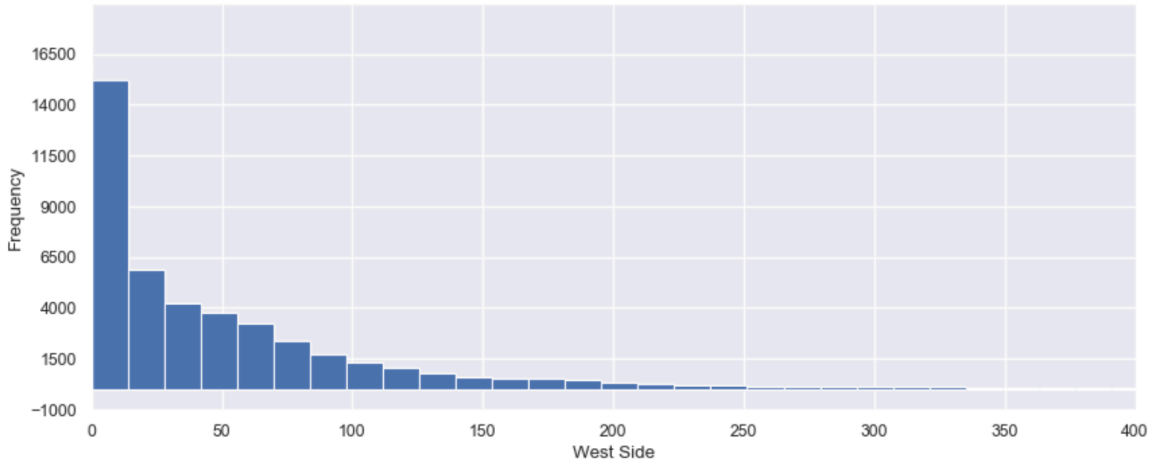
```
result.head()
```

hour	east_side	west_side	temperature	humidity	pressure	wind_direction	wind_speed	work_status	day_of_week	season
0.0	2.0	5.0	7.42	87.0	1003.0	0.0	0.0	1.0	Fri	Fall
1.0	1.0	0.0	7.78	87.0	1003.0	0.0	1.0	1.0	Fri	Fall
2.0	1.0	0.0	8.13	87.0	1003.0	160.0	2.0	1.0	Fri	Fall
3.0	1.0	0.0	8.24	64.0	1025.0	270.0	10.0	1.0	Fri	Fall
4.0	2.0	6.0	8.66	68.0	1019.0	309.0	5.0	1.0	Fri	Fall

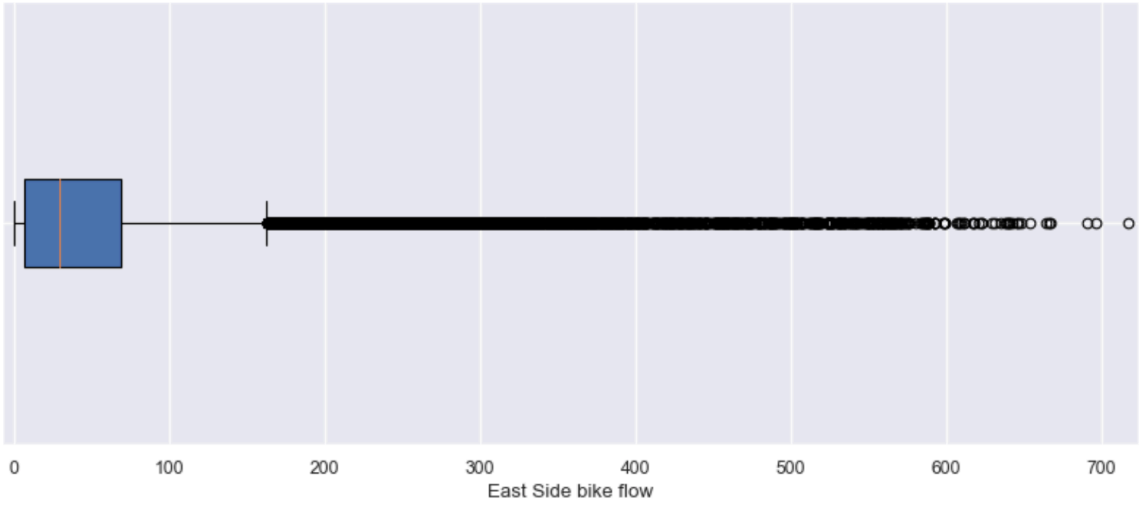
Now we will use this cleaned dataset for further analysis!

Exploratory Data Analysis:





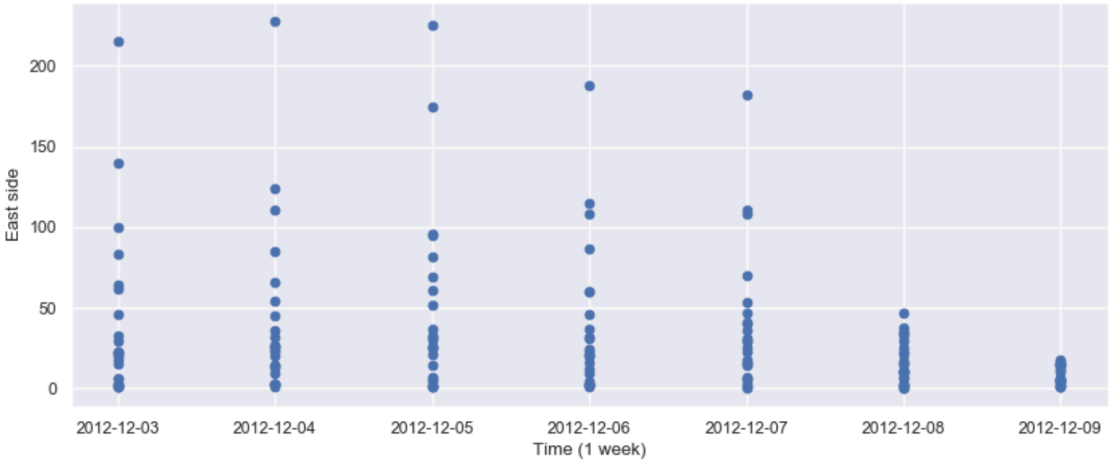
The above two graph is the distribution of the east and west side bike flow. We can see that it's normally distributed. And most of the flow number is below 200.



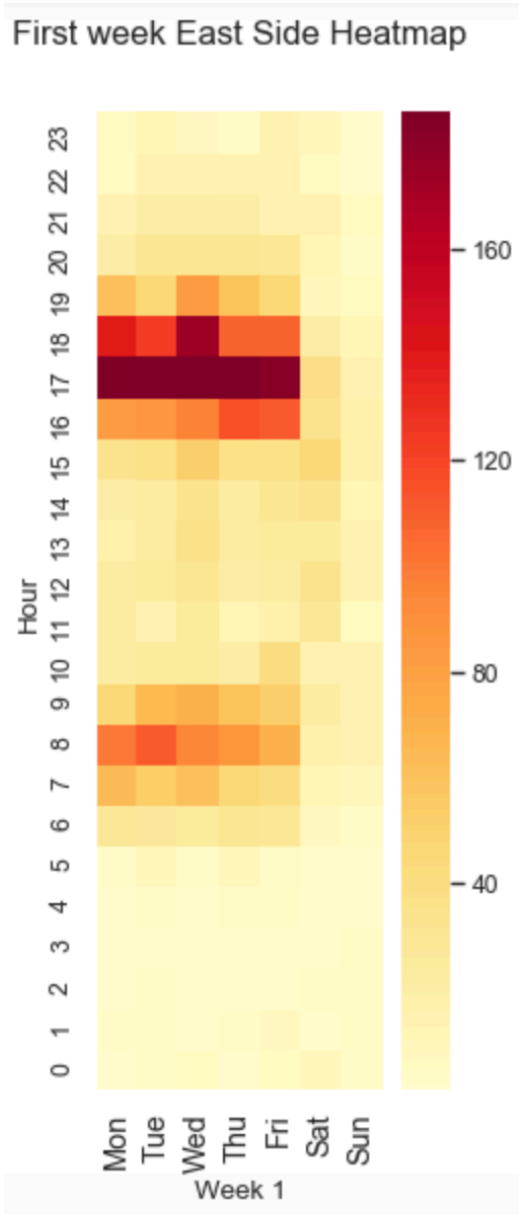
The bar plot of the east side, we can see that the outliers is pretty obvious. And we wang to see the specific percentage of that:

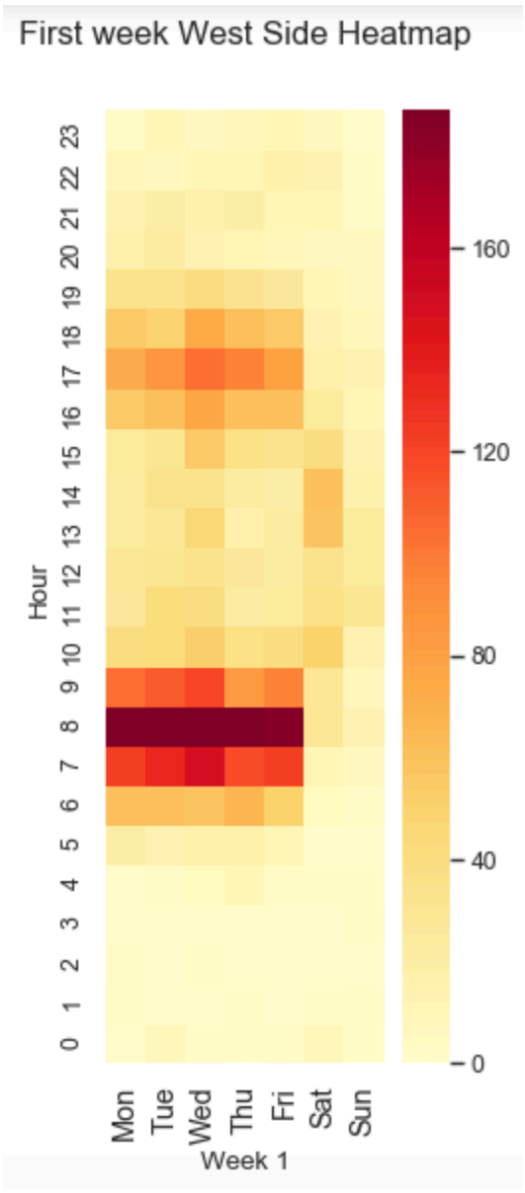
```
In [5]: print("Percentage of East Side in range of 0–200")
        print("{:.3f}%".format(
            (df[df.east_side <= 200]["east_side"].count()*100.0) / df.shape[0]))

Percentage of East Side in range of 0–200
93.953%

In [6]: print("Percentage of West Side in range of 0–200")
        print("{:.3f}%".format(
            (df[df.west_side <= 200]["west_side"].count()*100.0) / df.shape[0]))

Percentage of West Side in range of 0–200
95.306%
```

We can see that both east and west side the range is pretty high.



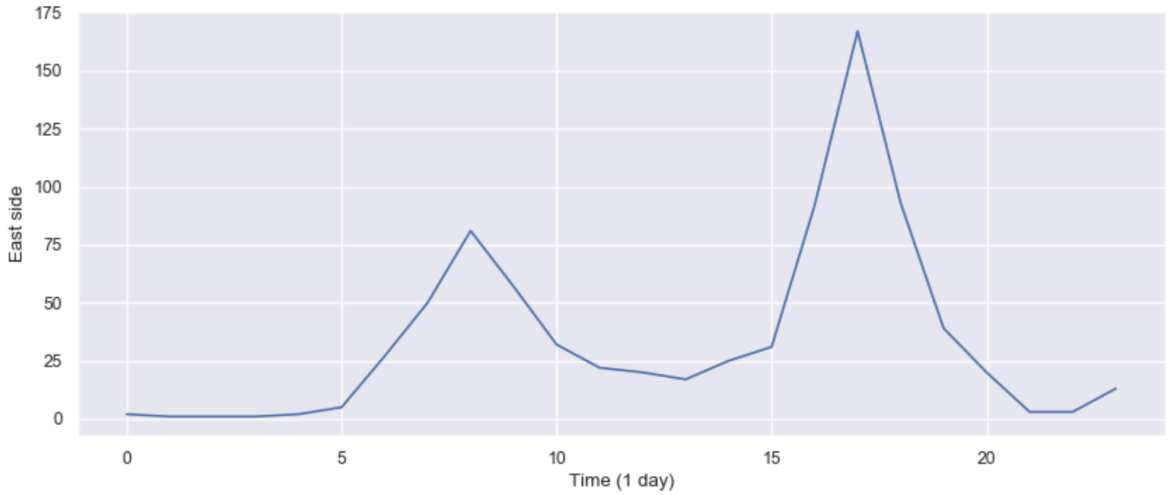
The above graph is the scatter plot of the 1st week's east side bike flow. We can see it's very obvious that the Sat and Sun day is smaller compared to other day. Then we will use a heat map to have a better of the 1st week daily behavior.





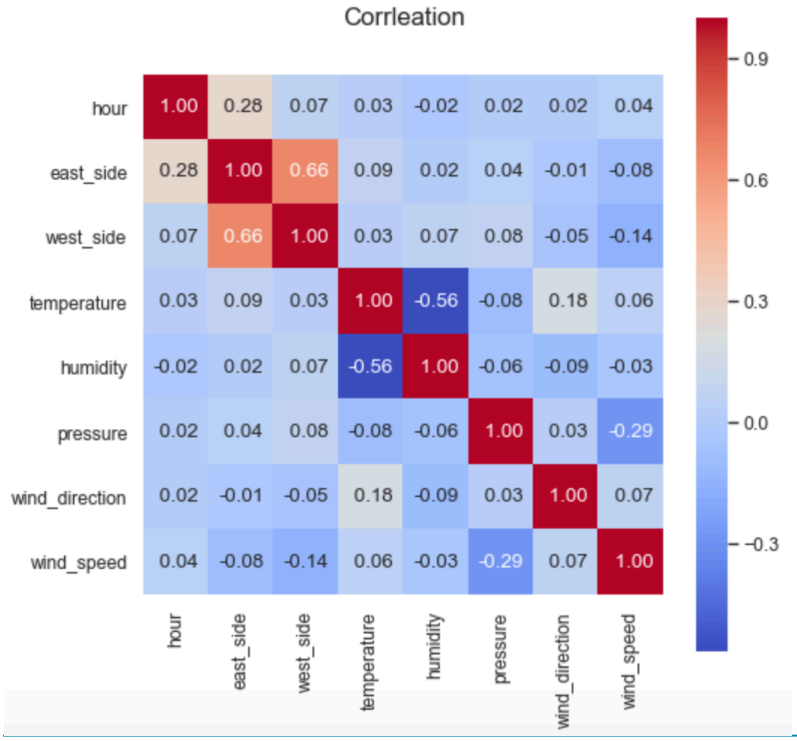
The above two graph is the first week heat map for both side. We can see that the behavior of the east and west is surprisingly **symmetric.** The east side bike flow reaches the peak around 16-18 o'clock while the west peak is around 7-9. And it will be a lower peak around 7-9 at east side while 16-18 at west side. Also weekday and weekend is pretty different. People don't have to work and the traffic will become little.

Let's see a one day hourly bike flow behavior (what we want to predict):



We took the first day(2012-11-30) as an example. We can see that because it's Friday the east bike flow have two peak value. The peak around 16-18 is higher. We may conclude that: **The people went to work at west is larger than the east and the morning rush hours are between 7 and 9, the evening rush hours are between 16-18.**

Then we will see the correlation between each feature:



We can see that there is no such pair which is strongly related. The temperature and the humidity is negative related and the east side and west side is positive related which makes sense. Because the higher temperature means the higher evaporation rate which result to low humidity. Also if the west bike flow is high the east will become higher because people need to get back to home from work!

Model Analysis:

Preparation:

See data imported properly.

```
import numpy as np
import pandas as pd
import csv

df0 = pd.read_csv('Data Preprocessing/complete2.csv')
print(df0.head())
```

```
   hour  east_side  west_side  temperature  humidity  pressure  \
0   0.0        2.0        5.0         7.42      87.0    1003.0
1   1.0        1.0        0.0         7.78      87.0    1003.0
2   2.0        1.0        0.0         8.13      87.0    1003.0
3   3.0        1.0        0.0         8.24      64.0    1025.0
4   4.0        2.0        6.0         8.66      68.0    1019.0

   wind_direction  wind_speed  work_status day_of_week season
0             0.0         0.0          1.0         Fri   Fall
1             0.0         1.0          1.0         Fri   Fall
2           160.0         2.0          1.0         Fri   Fall
3           270.0        10.0          1.0         Fri   Fall
4           309.0         5.0          1.0         Fri   Fall
```

Get dummy values.

```
Index(['temperature', 'humidity', 'pressure', 'wind_direction', 'wind_speed',
       'work_status', 'hour_0.0', 'hour_1.0', 'hour_10.0', 'hour_11.0',
       'hour_12.0', 'hour_13.0', 'hour_14.0', 'hour_15.0', 'hour_16.0',
       'hour_17.0', 'hour_18.0', 'hour_19.0', 'hour_2.0', 'hour_20.0',
       'hour_21.0', 'hour_22.0', 'hour_23.0', 'hour_3.0', 'hour_4.0',
       'hour_5.0', 'hour_6.0', 'hour_7.0', 'hour_8.0', 'hour_9.0',
       'day_of_week_Fri', 'day_of_week_Mon', 'day_of_week_Sat',
       'day_of_week_Sun', 'day_of_week_Thu', 'day_of_week_Tue',
       'day_of_week_Wed', 'season_Fall', 'season_Spring', 'season_Summer',
       'season_Winter'],
      dtype='object')
```

Define metrics

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from math import sqrt
def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    #return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    #avoid the 'a = 0' situation
    count = 0
    sum = 0
    for a, p in zip(y_true, y_pred):
        if(a!=0):
            sum+=(abs(a-p)/a)
            count+=1
    return (sum/count) * 100

def printMetrics(algorithm):
    print("RMSE: %.2f"
      % sqrt(mean_squared_error(y_test, algorithm)))
    print("MAPE: %.2f"
          % mean_absolute_percentage_error(y_test, algorithm)+'%')
    print("R2: %.2f"
          % r2_score(y_test, algorithm))
    print("MAE: %.2f"
          % mean_absolute_error(y_test, algorithm))
```

Time series Analysis:

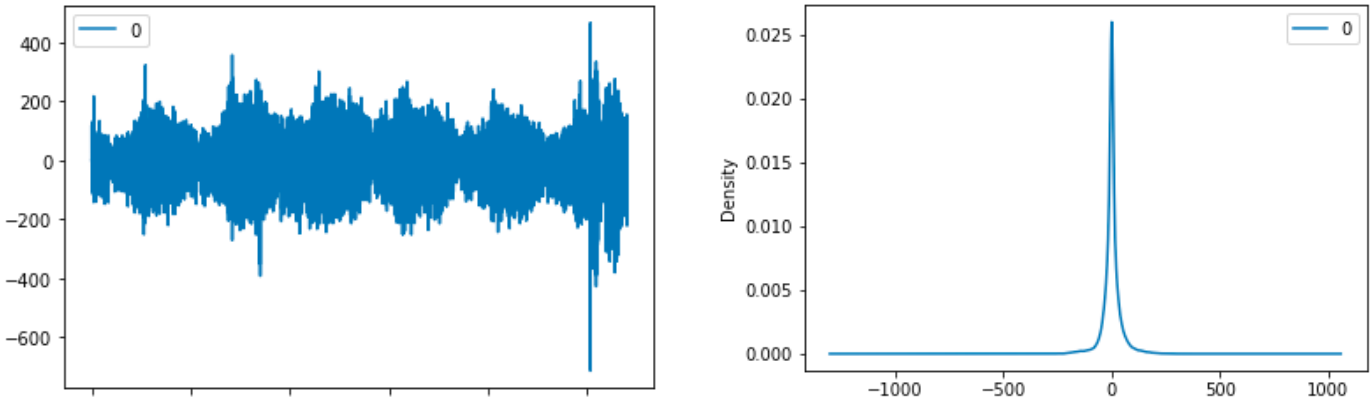
Introducing an algorithm ARIMA, and plain time series data for its usage.

```
from statsmodels.tsa.arima_model import ARIMA
import matplotlib.pyplot as plt

series = pd.read_csv('Data Preprocessing/bike_flow.csv'
print(series.head())
series = series['east']
plt.figure(figsize=(15,7))
series.plot()
plt.show()
```

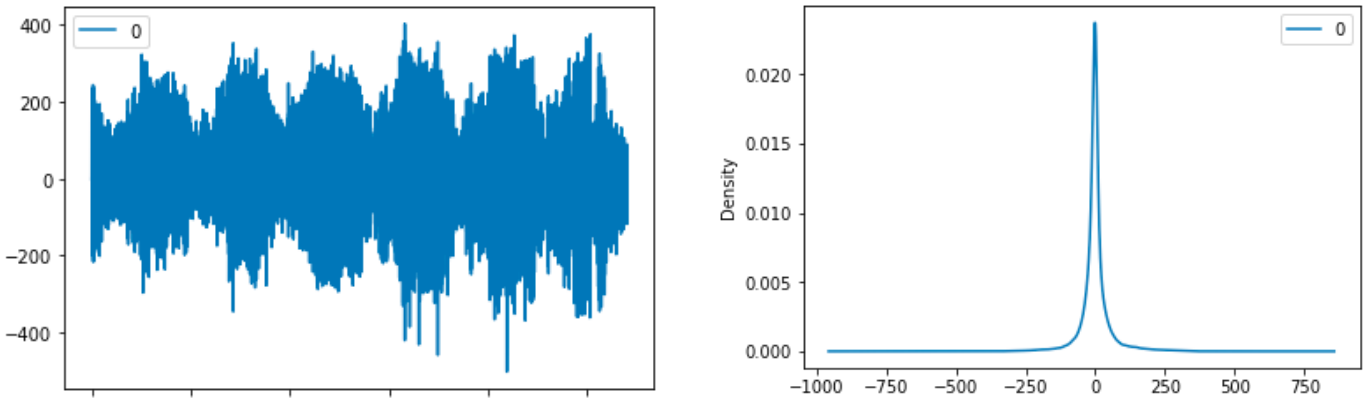
```
                      east  west
10/03/2012 12:00:00 AM   9.0   4.0
10/03/2012 01:00:00 AM   6.0   4.0
10/03/2012 02:00:00 AM   1.0   1.0
10/03/2012 03:00:00 AM   3.0   2.0
10/03/2012 04:00:00 AM   1.0   6.0

<Figure size 1500x700 with 1 Axes>
```

We shall see the information of both sides.

The residual of west side:

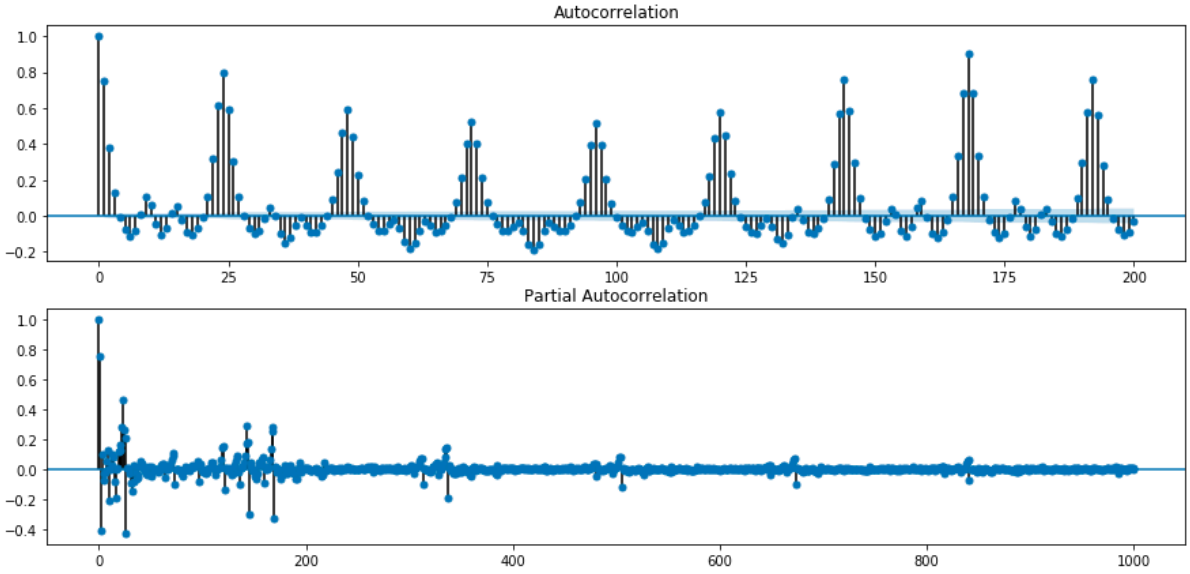


The residual of east side:

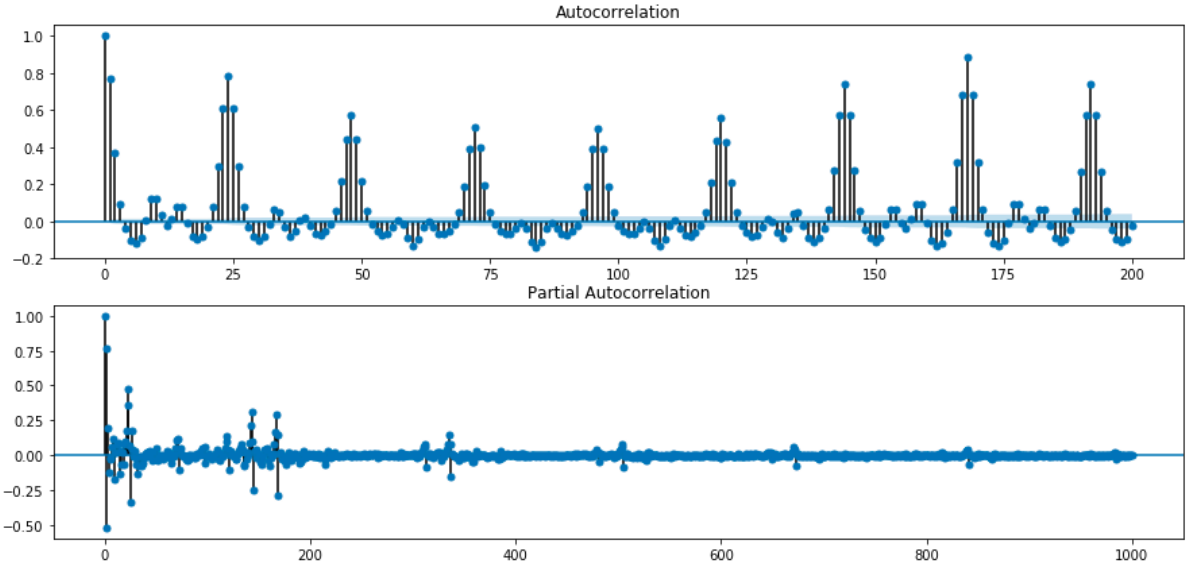


The data is stationary.

The auto-correlation and partial auto-correlation of east side:



The auto-correlation and partial auto-correlation of east side:

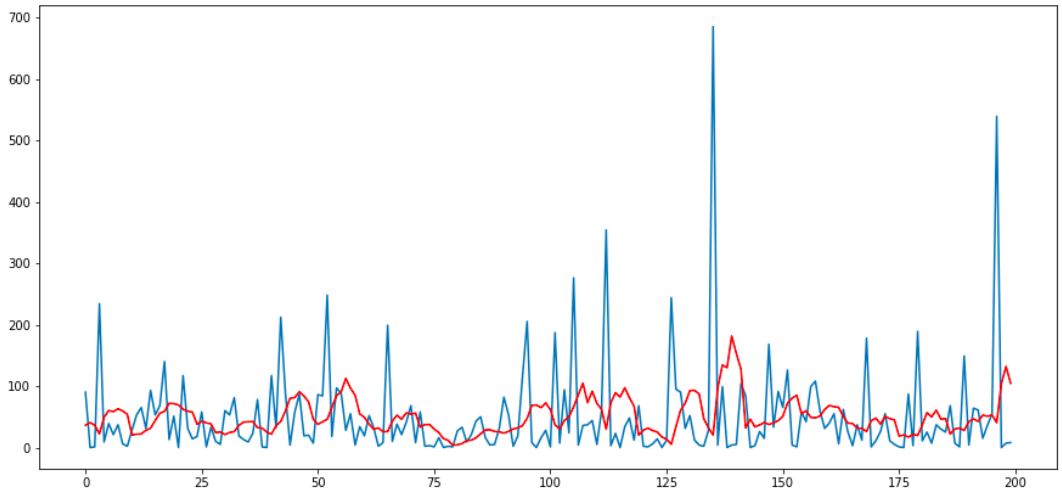


The patterns are alike. So both sides may share the same model for time series analysis.

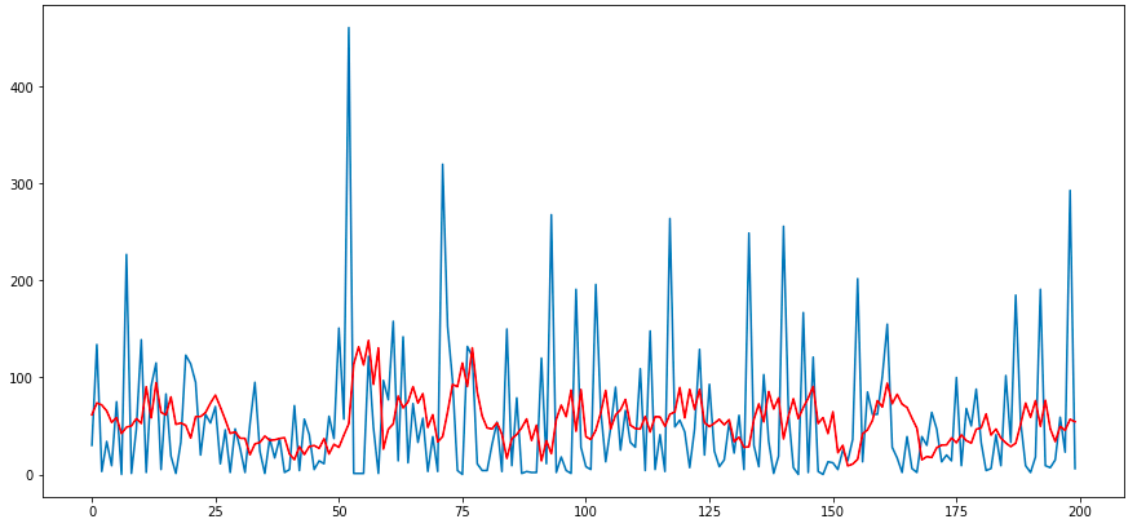
Build an ARIMA model to see the predictions.

```
#Rolling Forecast ARIMA Model
X = series.sample(1000).values
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)]
history = [x for x in train]
predictions = []
for t in range(len(test)):
    model = ARIMA(history, order=(5,1,0))
    model_fit = model.fit(disp=0)
    output = model_fit.forecast()
    yhat = output[0]
    predictions.append(yhat)
    obs = test[t]
    history.append(obs)
```

The result for east side:



The result for west side:



The performances are poor.

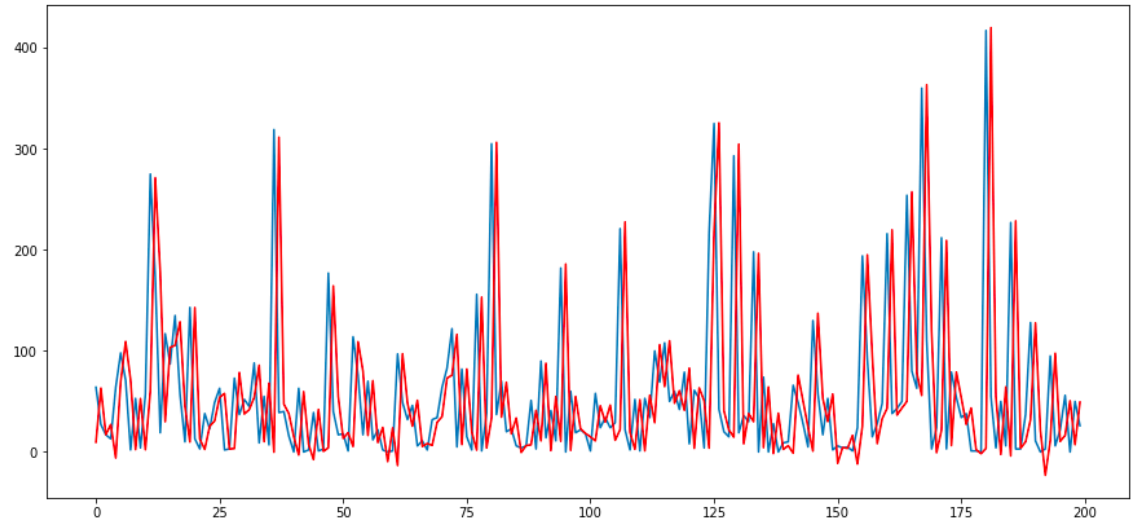
Build and tune a SARIMA model.
According to former observation of the data, we introduce a seasonal parameter for improvement. Perform a grid search on the model, using the following parameters:

```
def sarima_configs(seasonal=[12]):
    models = list()
    # define config lists
    p_params = [0, 1]
    d_params = [0, 1]
    q_params = [0, 1]
    t_params = ['n']
    P_params = [0, 1, 2]
    D_params = [0, 1]
    Q_params = [0, 1, 2]
    m_params = seasonal
    # create config instances
    for p in p_params:
        for d in d_params:
            for q in q_params:
                for t in t_params:
                    for P in P_params:
                        for D in D_params:
                            for Q in Q_params:
                                for m in m_params:
                                    cfg = [(p,d,q), (P,D,Q,m), t]
                                    models.append(cfg)
```

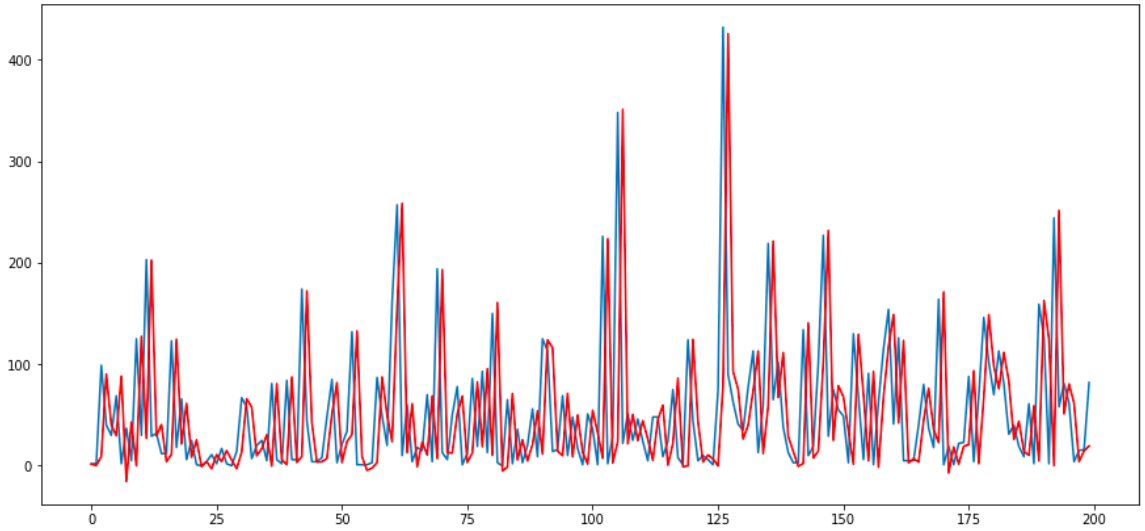
By its results, we select a the best model as : SARIMAX(histories, trend='n', order=(0,1,0), seasonal_order=(2,0,0,12))

e). Forecast according to tuned SARIMA model.

The performance against east side:



…And west side.



Now that the trend is good enough but yet the prediction is not accurate. The prediction is depending on its previous data point. So there appears to be a phase difference between the predictions and the observations.

Pipeline the model:

The approach of the team to solve such a problem is to break down date time element into pieces of features, and try to find tree regression algorithms for better performance.

With the help of tpot.

```
from tpot import TPOTRegressor

pipeline_optimizer = TPOTRegressor(generations=20, population_size=20, cv=3,
                                   random_state=42, verbosity=2)
pipeline_optimizer.fit(X_train, y_train)
print(pipeline_optimizer.score(X_test, y_test))
pipeline_optimizer.export('tpot_exported_pipeline.py')
```

This gives a pipeline of:

exported_pipeline = make_pipeline(
    StackingEstimator(estimator=LassoLarsCV(normalize=False)),
    StackingEstimator(estimator=RandomForestRegressor(bootstrap=True,
max_features=0.25, min_samples_leaf=1, min_samples_split=9, n_estimators=100)),
    ElasticNetCV(l1_ratio=0.7000000000000001, tol=0.001)
)
With the performance of:

```
RMSE: 23.95
MAPE: 60.42%
R2: 0.92
MAE: 13.28
```

Which is way more better.

	model_id	mean_residual_deviance	rmse	mse	mae
0	StackedEnsemble_AllModels_AutoML_20181214_044937	610.798416	24.714336	610.798416	13.572719
1	StackedEnsemble_BestOfFamily_AutoML_20181214_0...	622.599897	24.951952	622.599897	13.617491
2	XGBoost_2_AutoML_20181214_044937	624.291777	24.985832	624.291777	13.739135
3	XGBoost_1_AutoML_20181214_044937	629.165614	25.083174	629.165614	14.229641
4	XRT_1_AutoML_20181214_044937	669.727878	25.879101	669.727878	14.255596
5	DRF_1_AutoML_20181214_044937	670.066152	25.885636	670.066152	14.211585
6	GLM_grid_1_AutoML_20181214_044937_model_1	2411.222196	49.104197	2411.222196	30.884585

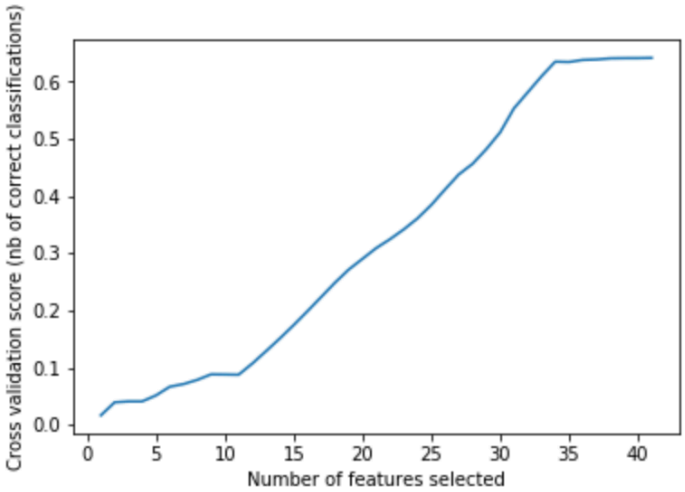
The execution takes long and the performance improves little. When trying greater complexities it gives similar cv scores between generations.

b). With the help of h2o:
It gives the algorithms above with their performances.

TPOT and H2O seems like agree with each other.

Feature Selection:

Although the Tpot has already provide us with ideal model, we also want to know the how the number of features will affect our result and we will use RFECV to do that:



['temperature', 'humidity', 'pressure', 'wind_direction', 'wind_speed', 'work_status', 'hour_0.0', 'hour_1.0', 'hour_10.0', 'hour_11.0', 'hour_12.0', 'hour_13.0', 'hour_14.0', 'hour_15.0', 'hour_16.0', 'hour_17.0', 'hour_18.0', 'hour_19.0', 'hour_2.0', 'hour_20.0', 'hour_21.0', 'hour_22.0', 'hour_23.0', 'hour_3.0', 'hour_4.0', 'hour_5.0', 'hour_6.0', 'hour_7.0', 'hour_8.0', 'hour_9.0', 'day_of_week_Fri', 'day_of_week_Mon', 'day_of_week_Sat', 'day_of_week_Sun', 'day_of_week_Thu', 'day_of_week_Tue', 'day_of_week_Wed', 'season_Fall', 'season_Spring', 'season_Summer', 'season_Winter']
Optimal number of features : 41

As we can see that as the number of features increase the prediction score increases. And the Optimal number means that all the features should be selected.

Model Interpretability:

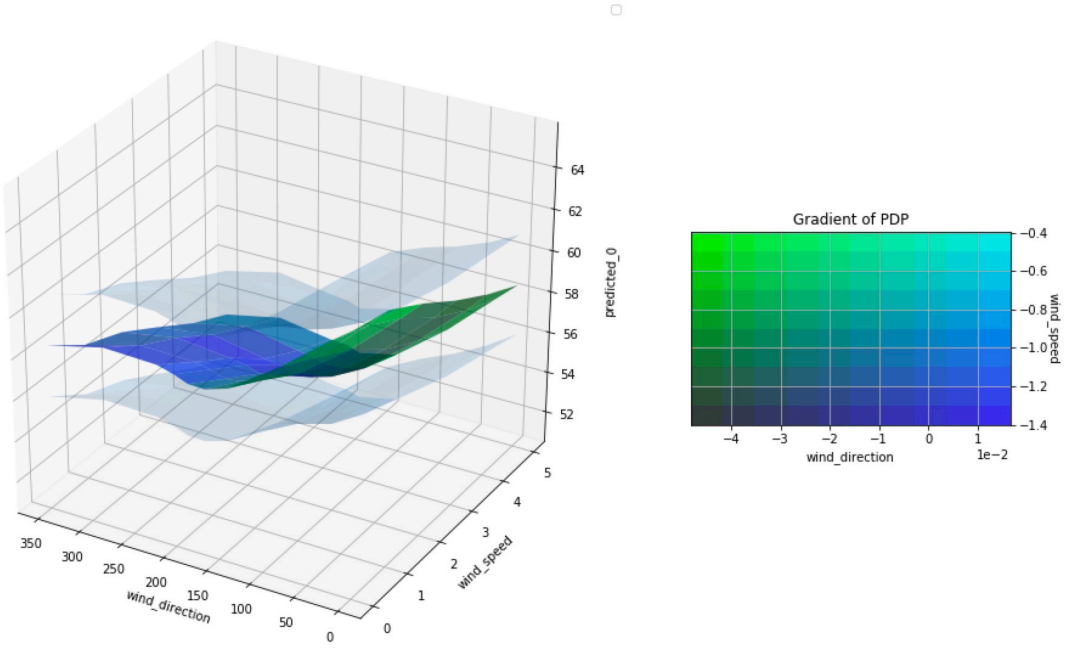
We will use the Skater package to interpret our model.
What we want to see in this regression model is the inner interaction between the features:

```
def understanding_interaction():
    # ['worst area', 'mean perimeter'] --> list(feature_selection.value)
    interpreter.partial_dependence.plot_partial_dependence(list(feature_selection.value),
                                                           pyint_model,
                                                           grid_resolution=grid_resolution.value,
                                                           with_variance=True)
```

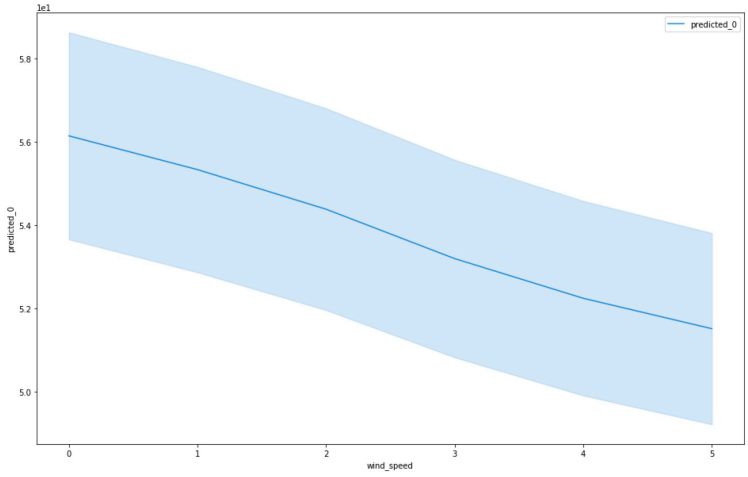
```
button = widgets.Button(description="Generate Interactions")
display(button)

def on_button_clicked(button_func_ref):
    clear_output()
    understanding_interaction()

button.on_click(on_button_clicked)
```

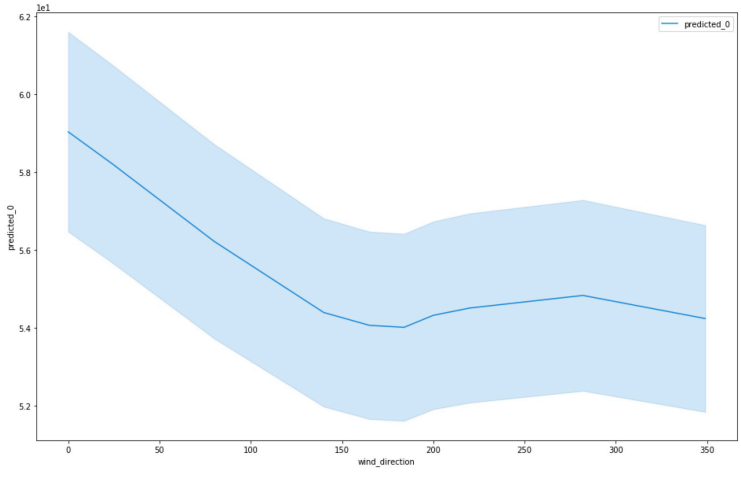
we will look into the features to see how the feature affect the prediction

This a three dimension graph. We can see the relation between the wind_direction and wind_speed and how these feature influence the prediction



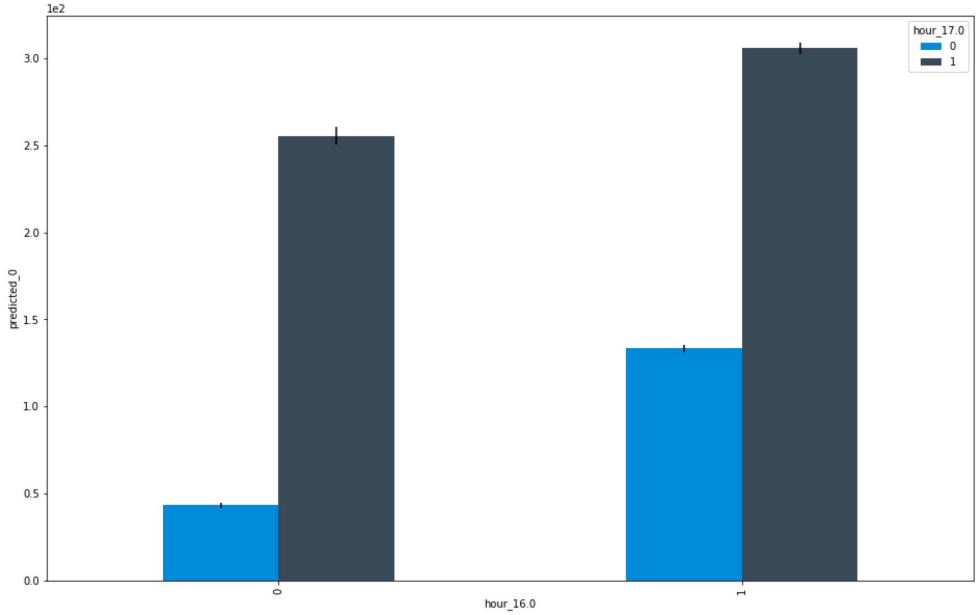
To have a better view of the feature we can also plot two dimension:





We can see both wind speed and wind direction will have the negative affect on the prediction. Which makes sense because the bike flow will decrease when the outside windspeed is high.

For the categorical feature the Skater can not do the continuous analysis. So the plot is basically a bar plot:



We can see that both hour 16 and hour 17 will influence the prediction and because the hour 17 is the people off duty time, so the impact will be higher compared to hour 16.

Web Application Implementation:

Some functions:

Function: **get_input(days):**
We will use this function to get the weather forecast of Seattle and pre process it to a data frame.
For the weather forecast we will use The API called forecastio. This api will give us the hourly weather forecast of the location we want.
We first need to find the geographical location of the Seattle:

```
#Seattle location
lat = 47.606209
lng = -122.332069
```

Then after that we can use the api to generate the forecast

```
forecast = forecastio.load_forecast(api,lat,lng,time = current+datetime.timedelta(offset),units='si')
h = forecast.hourly()
d = h.data
for p in d:
    times.append(p.time)
    for attr in attributes:
        data[attr].append(p.d[attr])
```

We just need to use the input days to get the date we want to predict and the api will give the result based on location and date.
Then we need to add some time series features which matches the model input feature.

```
result = pd.DataFrame()
result['hour'] = df['Hour']
result['temperature'] = df['temperature']
result['humidity'] = df['humidity'] * 100
result['pressure'] = df['pressure']
result['wind_direction'] = df['windBearing']
result['wind_speed'] = df['windSpeed']
result['work_status'] = df['Work_Status']
result['day_of_week'] = df['Day_of_Week']
result['season'] = df['season']
```

Function: **feature_engineering(df_in)**

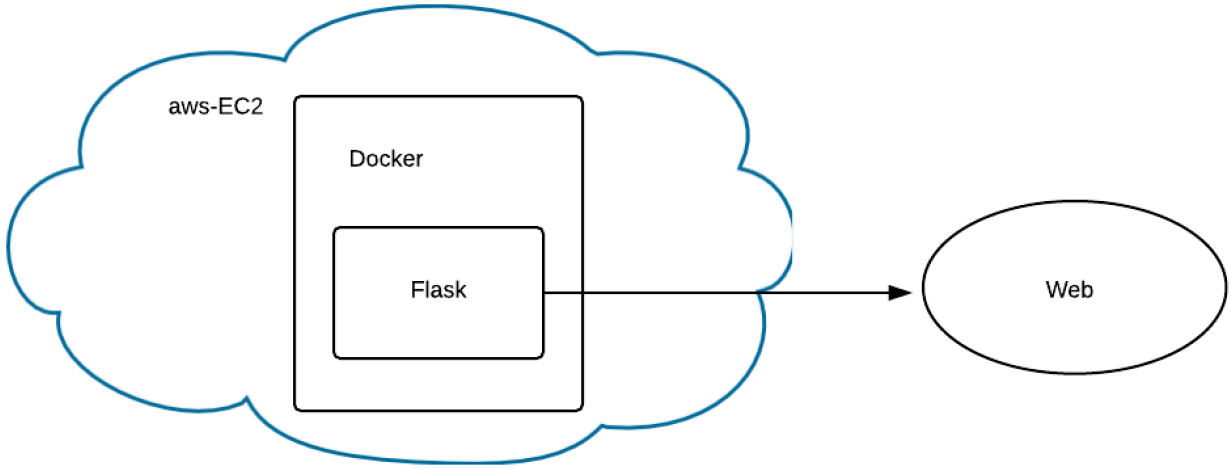
This function will create the input which can be used in our saved prediction model.
The challenge we met was the input data only have the information of one day, which is fine for the numeric data. But for the categorical data like 'season' ,'day_of_week' , we use one hot encoding to transform it into readable data. So for the input data we can't extract all the information of the categorical data. What we do to solve this problem is to use a **sample data frame** to combine with our input. The **sample.csv** will contain 1 year X-train data. By combining them together and then do the encoding, all the categorical columns will be created. After that we extract the rows we want..

```
def feature_engineering(df_in):
    sample = pd.read_csv('sample.csv',low_memory = False)
    df = df_in.copy()
    df_com = sample.append(df)
    df_com = pd.get_dummies(data=df_com, columns=['day_of_week', 'hour','season'])
    return df_com.tail(24)
```

Function **east_prediction(df_X) & west_prediction(df_X)**
These two functions will read the saved model and make prediction form the input data frame and then plot the **24 hours results** and save the photo for future display.

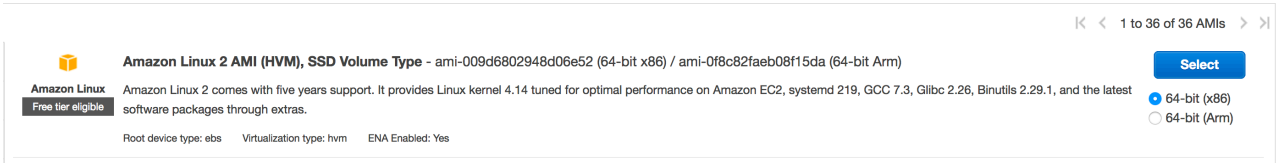
```
def east_prediction(df_x):
    filename = 'gaoshunan_east_model.sav'
    east_model = pickle.load(open(filename, 'rb'))
    result = east_model.predict(df_x)
    hour = np.arange(0, 24, 1)
    plt.plot(hour,result)
    plt.xlabel('Time (h)')
    plt.ylabel('East side bike flow')
    plt.title('East Prediction')
    plt.grid(True)
    plt.savefig("p-east.png")
```

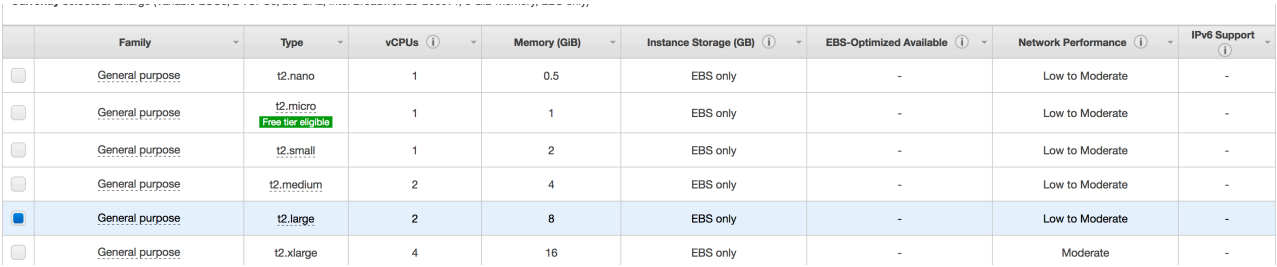
How to deploy our application.



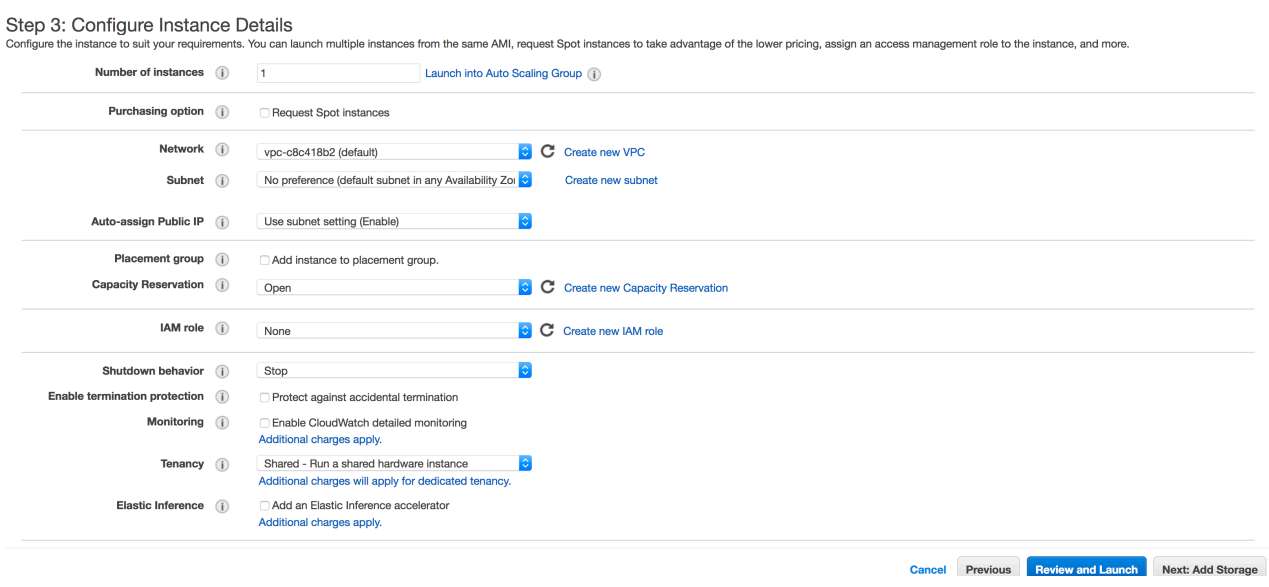
Docker Hub: https://cloud.docker.com/u/yinking422/repository/docker/yinking422/v3

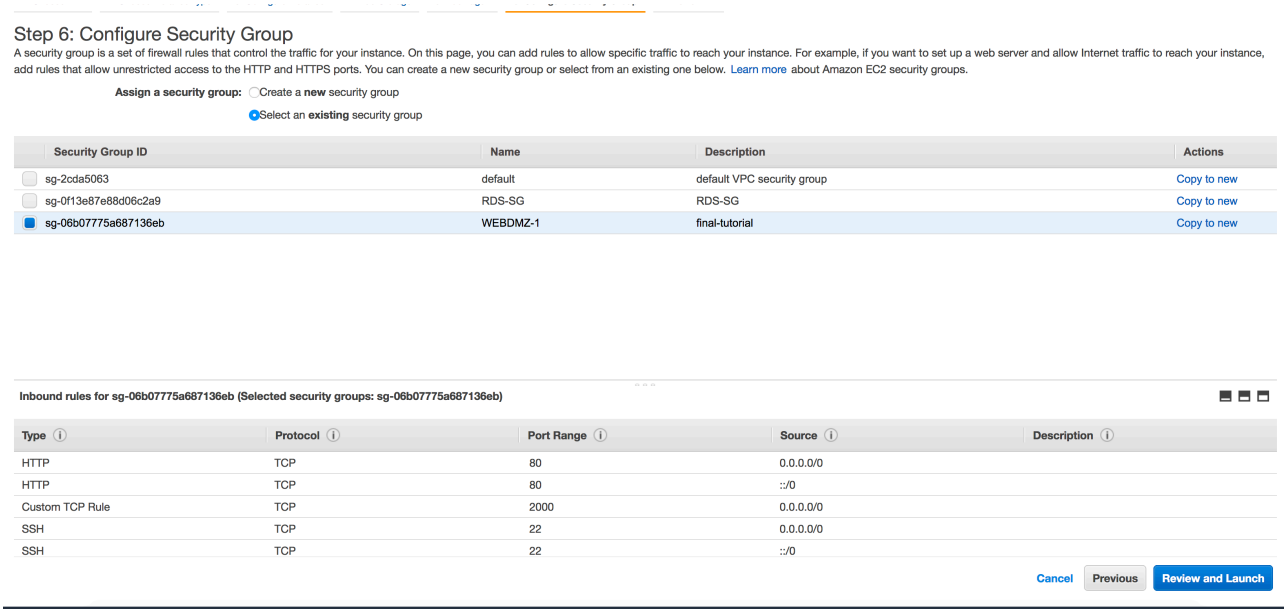
Step 1: Set up Aws EC2





	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance	IPv6 Support
☐	General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate	-
☐	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate	-
☐	General purpose	t2.small	1	2	EBS only	-	Low to Moderate	-
☐	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate	-
☑	General purpose	t2.large	2	8	EBS only	-	Low to Moderate	-
☐	General purpose	t2.xlarge	4	16	EBS only	-	Moderate	-





Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. Learn more about Amazon EC2 security groups.

Assign a security group: Create a new security group

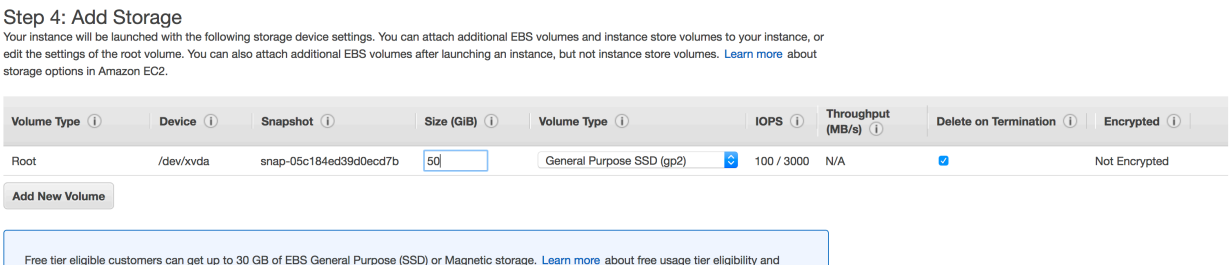
Select an existing security group

Security Group ID	Name	Description	Actions
sg-2cda5063	default	default VPC security group	Copy to new
sg-0f13e87e88d06c2a9	RDS-SG	RDS-SG	Copy to new
sg-06b07775a687136eb	WEBDMZ-1	final-tutorial	Copy to new

Inbound rules for sg-06b07775a687136eb (Selected security groups: sg-06b07775a687136eb)

Type	Protocol	Port Range	Source	Description
HTTP	TCP	80	0.0.0.0/0	
HTTP	TCP	80	::/0	
Custom TCP Rule	TCP	2000	0.0.0.0/0	
SSH	TCP	22	0.0.0.0/0	
SSH	TCP	22	::/0	

Cancel Previous Review and Launch



Step 4: Add Storage

Your instance will be launched with the following storage device settings. You can attach additional EBS volumes and instance store volumes to your instance, or edit the settings of the root volume. You can also attach additional EBS volumes after launching an instance, but not instance store volumes. Learn more about storage options in Amazon EC2.

Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Throughput (MB/s)	Delete on Termination	Encrypted
Root	/dev/xvda	snap-05c184ed39d0ecd7b	50	General Purpose SSD (gp2)	100 / 3000	N/A	☑	Not Encrypted

Add New Volume

Free tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or Magnetic storage. Learn more about free usage tier eligibility and

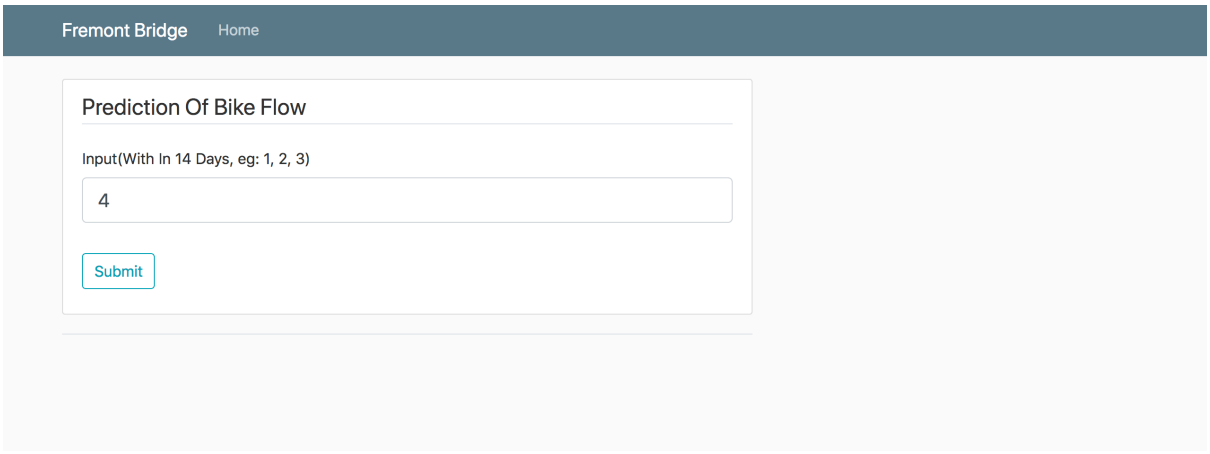
Since the running model need enough cpu.
The cpu and storage configuration is crucial.

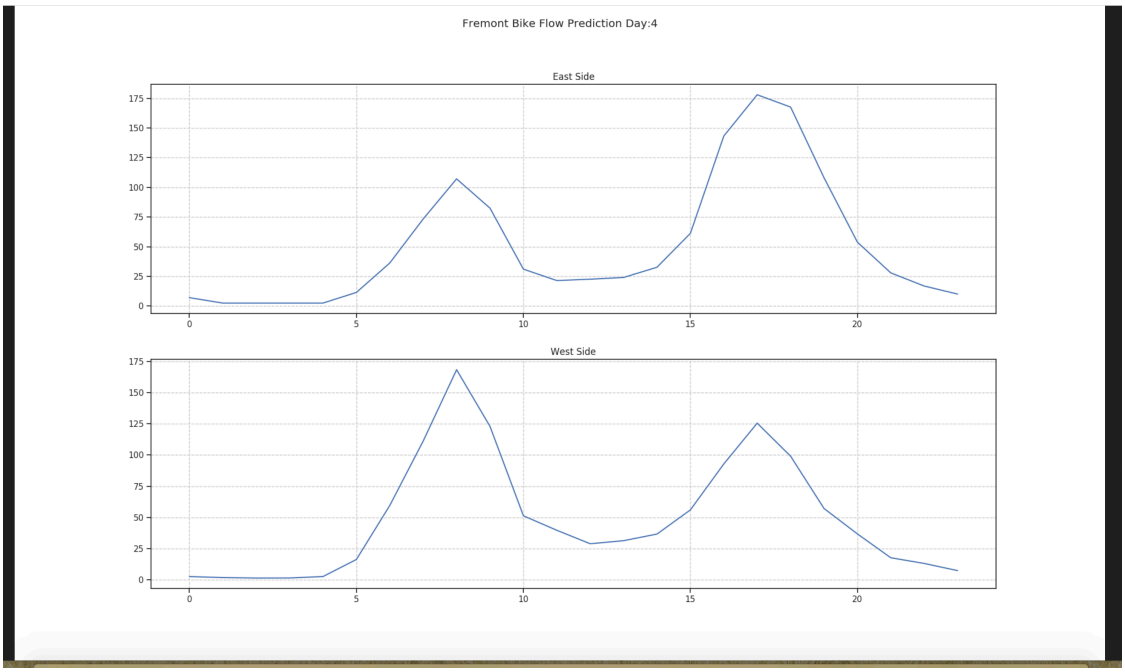
Step 2:
configure docker environment on ec2 instance

sudo yum update -y
sudo amazon-linux-extras install docker
sudo service docker start
sudo usermod -a -G docker ec2-user
docker info

pull docker image from our docker hub:
docker pull yinking422/v3
docker run -p 2000:2000 --name v3 -t yinking422/v3
publicIp: 2000 to visit our app

Screenshot of our application:





Conclusion:

1. Our application can perfectly run both on docker and EC2.
2. The prediction model we select is basically predict the ideal result
3. The result will be pretty useful for the bike drivers in Seattle