AI Project 1 - REPORT

Uber-Demand-Supply-Forecasting-with-AI

Faraz Razi - Ahmad

i201866 - i201893

04.20.2023

Section - F

Context

INTRODUCTION

This project involves building an AI-based model that predicts the gap between the supply and demand of ride-hailing services in a city, based on data collected over a period of time. The aim is to help Uber or other ride-hailing companies maximize the utilization of their drivers, and provide a reliable service to their customers.

PROBLEM AND DATA DETAILS

1. Background

As less than 10% of the world's citizens own automobiles, the frequency at which citizens commute on taxis, buses, trains, and planes is very high. Uber (or to some extent Careem), the dominant ride-hailing company, processes over 11 million trips, plans over 9 billion routes and collects over 50 TB of data per day. To meet needs of riders, Uber must continually innovate to improve cloud computing and big data technologies and algorithms in order to process this massive amount of data and uphold service reliability. Supply-demand forecasting is critical to enabling Uber to maximize utilization of drivers and ensure that riders can always get a car whenever and wherever they may need a ride. Supply-demand forecasting helps to predict the volume of drivers and riders at a certain time period in a specific geographic area. For instance, demand tends to surge in residential areas in the mornings and in business regions in the evenings. Supply-demand forecasting allows Uber to predict demand surges and guide drivers to those areas. The end result is higher earnings for drivers and no surge pricing for riders!

2. Definition and Evaluation Criteria

1. Definition

A passenger calls a ride(request)by entering the place of origin and destination and clicking "Request Pickup" on the Uber app. A driver answers the request (answer) by taking the order. Uber divides a city into n non-overlapping square regions D = $d,d,d,d,\ldots\ldots\ldots,d$ and divides one day uniformly into 144 time slots $t,t,t,\ldots\ldots\ldots,d$ each 10 minutes long. In region d , and time slot t , the number of passengers' requests is denoted as r, and drivers' answers as a . In region d and time slot t the demand is denoted as demand = r and the supply as supply = a , and the demand supply gap is: gap: gap = $r - a$. Given the data of every region d and time slot t, you need to predict gap, $\forall d \in$ D.

2. Evaluation Metrics

Given i regions and j time slots, for region d in time slot t , suppose that the real supply- demand gap is gap, and predicted supply-demand gap is s , then:

$$MeanAbsoluteError = \frac{1}{n}\sum_{d_i}(\frac{1}{q}\sum_{t_i}|gap_{ij} - s_{ij}|)$$

The lowest mean absolute error will be the best. The detailed description of each field is as follows:

Data Name	Data Type	Example
Region ID	String	1,2,3,4 (the same as region mapping ID)
Time slot	String	2016-01-23-1 (The first time slot on Jan. 23rd, 2016)
Prediction value	Double	6\.0

3. Data Format

The training set contains three consecutive weeks of data for City M in 2016, and you need to forecast the supply-demand gap for a certain period in the fourth and fifth weeks of City M. The test set contains the data of half an hour before the predicted time slot. The specific time slots where you need to predict the supply-demand gap are shown in the 1 explanation document in the test set.

The Order Information Table, Weather Information Table and POI Information Table are available in the database. All sensitive data has been anonymized.

The Order Information Table shows the basic information of an order, including the passenger and the driver (if driver id =NULL, it means the order was not answered by any driver), place of origin, destination, price and time. The fields order id, driver id, passenger id, start hash, and dest_hash are made not sensitive.

1. Order Information Table

Field	Type	Meaning	Example
order_id	string	order ID	70fc7c2bd2caf386bb50f8fd5dfef0cf
driver_id	string	driver ID	56018323b921dd2c5444f98fb45509de
passenger_id	string	user ID	238de35f44bbe8a67bdea86a5b0f4719
start_region_hash	string	departure	d4ec2125aff74eded207d2d915ef682f
dest_region_hash	string	destination	929ec6c160e6f52c20a4217c7978f681
Price	double	Price	37\.5
Time	string	Timestamp of the order	2016-01-15 00:35:11

2. Region Information Table

The Region Information Table shows the information about the regions to be evaluated in the contest. You need to do the prediction given the regions from the Region Definition Table. In the submission of the results, you need to map the region hash value to the region mapped ID.

Field	Type	Meaning	Example
region hash	string	Region hash	90c5a34f06ac86aee0fd70e2adce7d8a
region id	string	Region ID	1

3. POI Information Table

The POI Information Table shows the attributes of a region, such as the number of different facilities. For example, 2#1:22 means in this region, there are 22 facilities of the facility class 2#1. 2#1 means the first level class is 2 and the second level is 1, such as entertainment#theater, shopping#home appliance, sports#others. Each class and its number is separated by

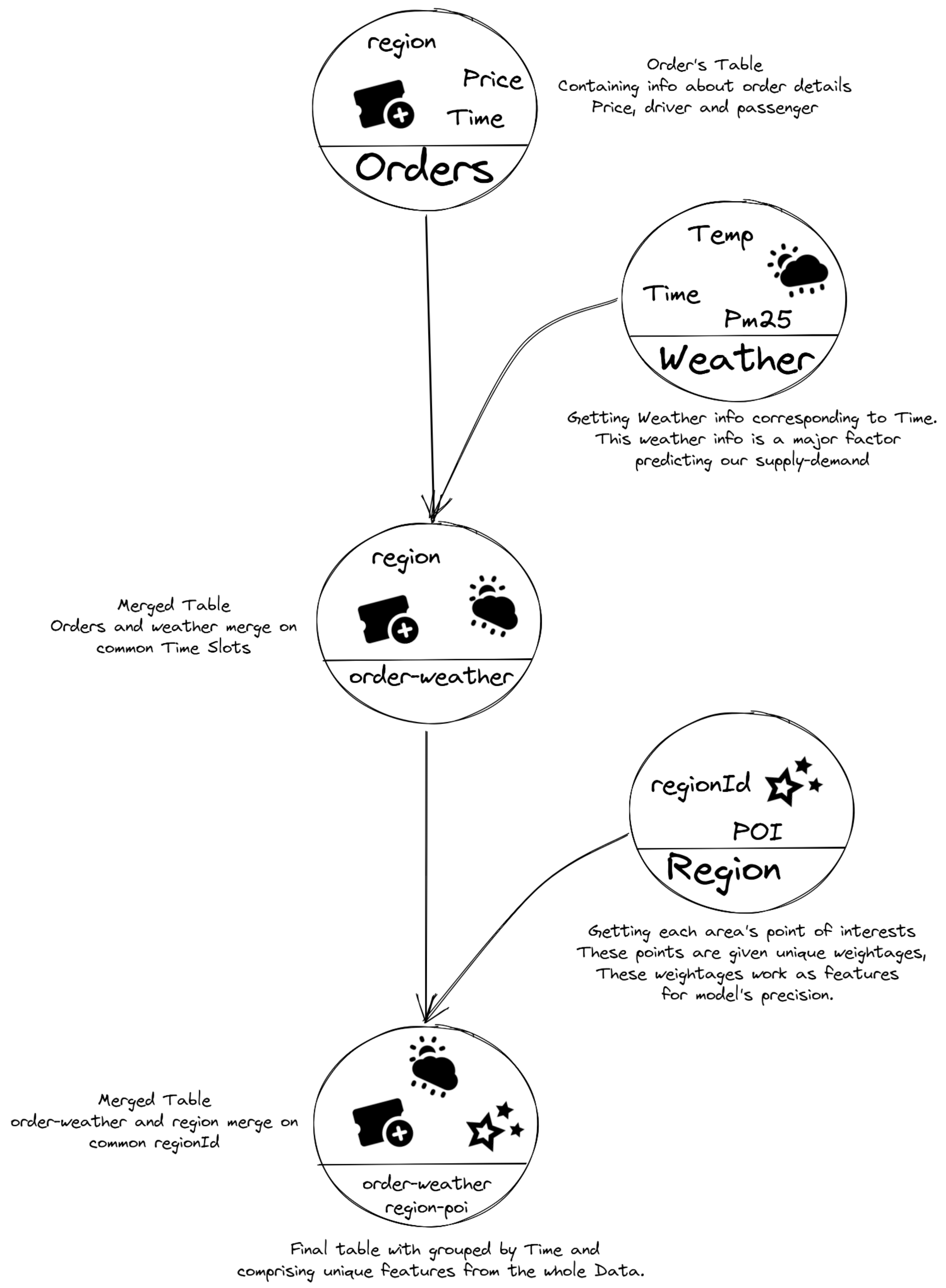
Field	Type	Meaning	Example
region hash	string	Region hash	74c1c25f4b283fa74a5514307b0d0278
poi class	string	POI class and its number	1#1:41 2#1:22 2#2:32

4. Weather Information Table

The Weather Information Table shows the weather info every 10 minutes in each city. The weather field gives the weather conditions such as sunny, rainy, and snowy etc; all sensitive information has been removed. The unit of temperature is Celsius degree, and PM2.5 is the level of air pollution.

Field	Type	Meaning	Example
Time	string	Timestamp	2016-01-15 00:35:11
Weather	int	Weather	7
temperature	double	Temperature	-9
PM2.5	double	pm25	66

SELECTING AND EXTRACTING FEATURE


region
Price
Time
Orders
Order's Table
Containing info about order details
Price, driver and passenger
Temp
Time
Pm25
Weather
Getting Weather info corresponding to Time.
This weather info is a major factor
predicting our supply-demand
region
order-weather
Merged Table
Orders and weather merge on
common Time Slots
regionId
POI
Region
Getting each area's point of interests
These points are given unique weightages,
These weightages work as features
for model's precision.
Merged Table
order-weather and region merge on
common regionId
order-weather
region-poi
Final table with grouped by Time and
comprising unique features from the whole Data.

Final Data Includes following:

All variables :

1. time_slot,
2. weekday,
3. order_gap
4. supply
5. demand
6. temperature
7. pm25
8. start_region_id
9. start_poi_ids
10. dest_region_id
11. dest_poi_ids

dependent variable:

1. order_gap
2. supply
3. demand

independent variables:

1. time_slot
2. weekday
3. temperature
4. pm25
5. start_region_id
6. start_poi_ids
7. dest_region_id
8. dest_poi_ids

Training data processing:

Explanation of Code to Label Cluster Map:

```
# label the cluster map
# labels:
# region_hash, region_id

columns = ['region_hash', 'region_id']
# read the cluster map
cluster_map =
pd.read_csv('../dataset/training_data/cluster_map/cluster_map', sep='\t',
on_bad_lines='skip', header=None, names=columns)
print('cluster_map finished')

print('cluster_map.head(): \n', cluster_map.head())


cluster_map.to_csv('../dataset/labeledData/cluster_map.csv', index=False)
```

The code aims to label the cluster map by assigning two labels - 'region_hash' and 'region_id' - to each cluster. It starts by defining the column names and then reads the cluster map from a CSV file using pandas' read_csv function. The 'skip' parameter is used to skip bad lines in the file, and the 'header' parameter is set to 'None' since the file doesn't have a header. The cluster map is then printed using the head() function to display the first few rows. Finally, the labeled cluster map is exported to a CSV file using the to_csv function. The output file will be saved in the 'labeledData' folder.

Labeling and Concatenating Orders Data into a CSV File:

```
# label the orders data
# labels:
# order_id, driver_id, passenger_id, start_district_hash,
dest_district_hash, price, time
columns = ['order_id', 'driver_id', 'passenger_id', 'start_region_hash',
'dest_region_hash', 'price', 'time']

# read the orders data
orders_data = []
for f in glob.glob('../dataset/training_data/order_data/order_data_*'):
    # file name
    print('filename: ', f)
    df = pd.read_csv(f, sep='\t', on_bad_lines='skip', header=None,
names=columns)
    orders_data.append(df)

print('orders_data finished')
orders_data = pd.concat(orders_data,  ignore_index=True)

# print('orders_data.head(): ', orders_data.head())
orders_data.to_csv('../dataset/labeledData/orders_data.csv', index=False)
```

This code reads the orders data from multiple files, labels the columns as order_id, driver_id, passenger_id, start_district_hash, dest_district_hash, price, and time. It then concatenates the data from all files into a single pandas DataFrame and saves it as a CSV file. The function uses glob to read all files that match the pattern in the specified directory. The data is separated by tabs and any bad lines are skipped. The resulting CSV file contains all the orders data.

Code to Label and Combine Weather Data:

```
# label the weather data
# labels:
# time, weather, temperature, pm25
columns = ['time', 'weather', 'temperature', 'pm25']

# print('weather_data.head(): \n', weather_data.head())


# # read the weather data
weather_data = []
for f in
glob.glob('../dataset/training_data/weather_data/weather_data_*'):
    # file name
    print('filename: ', f)
    df = pd.read_csv(f, sep='\t', on_bad_lines='skip', header=None,
names=columns)
    weather_data.append(df)

print('weather_data finished')
weather_data = pd.concat(weather_data, ignore_index=True)
```

This Python code reads weather data from multiple files, labels it with columns for time, weather, temperature, and pm25, and combines it into a single Pandas dataframe. The script first defines the column names and then uses a for loop to read in the data from each file, skipping any bad lines. It then combines all the data into a single dataframe using the pd.concat() method and saves the labeled data to a CSV file.

Data Preprocessing: Combining and Labeling POI Data:

```
# label the poi data
# labels:
# region_hash, poi_id
# 1st column: district_hash
# whole next column is: poi_id
columns = ['region_hash', 'poi_id']



# read the poi data
poi_data = pd.read_csv('../dataset/training_data/poi_data/poi_data',
sep='\t', header=None, on_bad_lines='skip')

# extract the district_hash column and the POI ID columns
district_hash = poi_data.iloc[:, 0]
poi_ids = poi_data.iloc[:, 1:]

# combine all the POI IDs for each row into a list
poi_ids_list = poi_ids.apply(lambda x: x.tolist(), axis=1)

# combine the district_hash and poi_ids_list into a new DataFrame
labeled_poi_data = pd.concat([district_hash, poi_ids_list], axis=1)
labeled_poi_data.columns = ['region_hash', 'poi_ids']

# print the result
# print(labeled_poi_data.head())

# updated list
updated_list = []

# convert the column of lists to a list of lists
list_of_lists_poi_id = labeled_poi_data['poi_ids'].tolist()

# poi format poi_id = class1#class2:numofFacilities
# seperate numofFacilities from list_of_lists_poi_id and sum them up
# for each list in list_of_lists_poi_id
# change the list of poi_id to weighted sum of numofFacilities
for poi_list in list_of_lists_poi_id:
    weighted_sum = 0
    for poi in poi_list:
        if(pd.isna(poi)==False):
            poi_id, num_of_facilities = poi.split(':')
            poi_class = poi_id.split('#')
            # combine the class1 and class2 numbers
            if(len(poi_class) == 1):
                poi_class[0] = '0' + poi_class[0]
            else:
                poi_number = poi_class[0] + '' + poi_class[1]
            weighted_sum += int(num_of_facilities) * int(poi_number)

    updated_list.append(weighted_sum)

# print(list_of_lists_poi_id)

# change labeled_poi_data['poi_ids'] to list_of_lists_poi_id
labeled_poi_data['poi_ids'] = updated_list

print('labeled_poi_data.head(): ', labeled_poi_data.head())

labeled_poi_data.to_csv('../dataset/labeledData/poi_data.csv',
index=False)
```

The code reads POI (point of interest) data and extracts the district hash and POI ID columns. It then combines all the POI IDs for each row into a list and creates a new DataFrame with the district hash and the list of POI IDs. The code then converts the list of POI IDs into a weighted sum of the number of facilities for each class of POI and saves the resulting labeled POI data to a CSV file.

Mapping Time to Time Slots with Weekday:

```
# map time to time slot
# devide day in 10 min time slots (144 time slots)



# convert time to datetime
```

```
orders_data['time'] = pd.to_datetime(orders_data['time'])
weather_data['time'] = pd.to_datetime(weather_data['time'])

# map time to time slot
orders_data['time_slot'] = orders_data['time'].dt.hour * 6 +
orders_data['time'].dt.minute // 10
weather_data['time_slot'] = weather_data['time'].dt.hour * 6 +
weather_data['time'].dt.minute // 10

# map time to time slot with weekday
orders_data['weekday'] = orders_data['time'].dt.weekday
weather_data['weekday'] = weather_data['time'].dt.weekday

# remove the time column
orders_data = orders_data.drop(['time'], axis=1)
weather_data = weather_data.drop(['time'], axis=1)

print(orders_data.head())
print(weather_data.head())
```

This code maps time to time slots, dividing a day into 10-minute time slots (144 time slots). It converts the time data to datetime format and creates a new column 'time_slot' with the corresponding time slot number for each order and weather data. It also creates a new column 'weekday' for the day of the week. Finally, it drops the original 'time' column and prints the first few rows of both orders and weather data.

Grouping and Calculating Supply-Demand Deficit in Orders Data:

```
# group the orders data by time slot
# aggregate count the number of orders where driver_id = NULL

# this is supply demand deficit - order gap
orders_data_grouped =
orders_data[orders_data['driver_id'].isnull()].groupby(['start_region_hash
','dest_region_hash','time_slot', 'weekday']).agg({'order_id':
'count'}).rename(columns={'order_id': 'order_gap'}).reset_index()

# this is the total demand
total_orders_grouped =
orders_data.groupby(['start_region_hash','dest_region_hash','time_slot',
'weekday']).agg({'order_id': 'count', 'price':
'mean'}).rename(columns={'order_id': 'demand'}).reset_index()

# merge the two dataframes on the region, time slot and weekday
orders_data_grouped = pd.merge(orders_data_grouped, total_orders_grouped,
on=['start_region_hash','dest_region_hash','time_slot', 'weekday'])

# calculate the supply variable as the difference between total_orders and
order_gap
orders_data_grouped['supply'] = orders_data_grouped['demand'] -
orders_data_grouped['order_gap']

print(orders_data_grouped.head())
```

This code performs a series of operations on an orders dataset. It first groups the data by time slot and counts the number of orders where the driver ID is null, creating a dataframe called orders_data_grouped. This represents the supply-demand deficit or order gap. It then calculates the total demand by grouping the data again by region, time slot, and weekday, creating a dataframe called total_orders_grouped. The two dataframes are merged on region, time slot, and weekday, and the supply variable is calculated as the difference between the total orders and order gap. The resulting dataframe is printed.

Grouping and Merging DataFrames in Python

```
# group the weather data by time slot
# aggregate the mean of temperature and pm25
weather_data_grouped = weather_data.groupby(['time_slot',
'weekday']).agg({'temperature': 'mean', 'pm25': 'mean'}).reset_index()

print(weather_data_grouped)

# merge the orders data and weather data
orders_weather_data = pd.merge(orders_data_grouped, weather_data_grouped,
on=['time_slot', 'weekday'], how='inner' )

print(orders_weather_data)

# merge the poi_list class characteristics with the cluster_map
# cluster_map: region_hash, region_id
# poi_data: district_hash, poi_ids
# merge on district_hash
cluster_map_poi = pd.merge(cluster_map, poi_data, left_on='region_hash',
right_on='region_hash', how='inner')

# remove the region_hash column
# cluster_map_poi = cluster_map_poi.drop(['region_id'], axis=1)

print(cluster_map_poi)
```

The code above first groups weather data by time slot and weekday, calculating the mean temperature and pm25 for each group. It then merges this grouped weather data with another DataFrame containing orders data, matching on time slot and weekday. Finally, it merges a cluster map DataFrame with POI (Point of Interest) data, using district_hash as the key. The resulting DataFrame includes information about both the characteristics of different geographic regions and the points of interest located within them.

Merging, renaming and cleaning data:

```
orders_weather_cluster_map_poi = pd.merge(orders_weather_data,
cluster_map_poi, left_on='start_region_hash', right_on='region_hash',
how='inner')
orders_weather_cluster_map_poi=orders_weather_cluster_map_poi.rename(colum
ns={'poi_ids': 'start_poi_ids'}).rename(columns={'region_id':
'start_region_id'})

orders_weather_cluster_map_poi = pd.merge(orders_weather_cluster_map_poi,
cluster_map_poi, left_on='dest_region_hash', right_on='region_hash',
how='inner')
orders_weather_cluster_map_poi=orders_weather_cluster_map_poi.rename(colum
ns={'poi_ids': 'dest_poi_ids'}).rename(columns={'region_id':
'dest_region_id'})

# remove the start_district_hash column
orders_weather_cluster_map_poi =
orders_weather_cluster_map_poi.drop(['start_region_hash',
'dest_region_hash', 'region_hash_x','region_hash_y'], axis=1)

print(orders_weather_cluster_map_poi)



# save the data
```

```
orders_weather_cluster_map_poi.to_csv('../dataset/processedData/orders_wea
ther_cluster_map_poi.csv', index=False)
```

This code merges three pandas dataframes based on common columns, renames some columns, drops some unnecessary columns and saves the final dataframe to a CSV file. The merged dataframe contains information on orders, weather and points of interest (POI) clustered by region. The code first merges the orders and weather data with the cluster map based on the start and destination region hashes. Then, it renames the POI and region ID columns for the start and destination regions. Finally, it drops some redundant columns and saves the resulting dataframe as a CSV file.

Testing data processing:

Labeling and Saving Cluster Map:

```
# label the cluster map
# labels:
# region_hash, region_id

columns = ['region_hash', 'region_id']
# read the cluster map
cluster_map = pd.read_csv('../dataset/test_set/cluster_map/cluster_map',
sep='\t', on_bad_lines='skip', header=None, names=columns)
print('cluster_map finished')

print('cluster_map.head(): \n', cluster_map.head())


cluster_map.to_csv('../dataset/labeledTestSet/cluster_map.csv',
index=False)
```

This code reads a cluster map from a file in the test_set directory using pandas library and then assigns two labels to the columns - region_hash and region_id. After reading, it prints a message indicating that the cluster map has been read successfully and then displays the first few rows of the cluster map. Finally, it saves the labeled cluster map to a CSV file named "cluster_map.csv" in the labeledTestSet directory without the index column. The purpose of this code is to label and save the cluster map for further analysis.

Code to Label and Concatenate Orders Data into a CSV File:

```
# label the orders data
# labels:
```

```
# order_id, driver_id, passenger_id, start_district_hash,
dest_district_hash, price, time

columns = ['order_id', 'driver_id', 'start_region_hash',
'dest_region_hash', 'time']

# read the orders data
orders_data = []
for f in glob.glob('../dataset/test_set/order_data/test_order_data_*'):
    # file name
    print('filename: ', f)
    df = pd.read_csv(f, sep=',', on_bad_lines='skip', header=None,
names=columns)
    orders_data.append(df)

print('orders_data finished')
orders_data = pd.concat(orders_data,  ignore_index=True)

# print('orders_data.head(): ', orders_data.head())
orders_data.to_csv('../dataset/labeledTestSet/orders_data.csv',
index=False)
```

This Python code reads in multiple CSV files of orders data and concatenates them into a single DataFrame. The columns of the DataFrame are labeled as order_id, driver_id, start_district_hash, dest_district_hash, and time. The data is then saved as a CSV file named orders_data.csv in a designated directory. The code utilizes the Pandas library and glob module to accomplish this task.

Concatenating and Labeling Weather Data from Multiple Files:

```
# label the weather data
# labels:
# time, weather, temperature, pm25
columns = ['time', 'weather', 'temperature', 'pm25']

# print('weather_data.head(): \n', weather_data.head())


# # read the weather data
weather_data = []
for f in glob.glob('../dataset/test_set/weather_data/weather_data_*'):
    # file name
    print('filename: ', f)
    df = pd.read_csv(f, sep='\t', on_bad_lines='skip', header=None,
names=columns)
    weather_data.append(df)

print('weather_data finished')
weather_data = pd.concat(weather_data, ignore_index=True)


weather_data.to_csv('../dataset/labeledTestSet/weather_data.csv',
index=False)
```

This code reads weather data from multiple files, concatenates them into a single dataframe, and then labels the columns as time, weather, temperature, and pm25. The weather data is read from files located in a specified directory using the glob library. The resulting concatenated dataframe is then saved as a CSV file in a specified directory.

Code to extract and process POI (Point of Interest) data:

```

columns = ['region_hash', 'poi_id']



# read the poi data
poi_data = pd.read_csv('../dataset/test_set/poi_data/poi_data', sep='\t',
header=None, on_bad_lines='skip')

# extract the district_hash column and the POI ID columns
district_hash = poi_data.iloc[:, 0]
poi_ids = poi_data.iloc[:, 1:]

# combine all the POI IDs for each row into a list
poi_ids_list = poi_ids.apply(lambda x: x.tolist(), axis=1)

# combine the district_hash and poi_ids_list into a new DataFrame
labeled_poi_data = pd.concat([district_hash, poi_ids_list], axis=1)
labeled_poi_data.columns = ['region_hash', 'poi_ids']

# print the result
```

```python
# print(labeled_poi_data.head())

# updated list
updated_list = []

# convert the column of lists to a list of lists
list_of_lists_poi_id = labeled_poi_data['poi_ids'].tolist()

# poi format poi_id = class:numofFacilities
# seperate numofFacilities from list_of_lists_poi_id and sum them up

# for each list in list_of_lists_poi_id
# change the list of poi_id to sum of numofFacilities
for poi_list in list_of_lists_poi_id:
    weighted_sum = 0
    for poi in poi_list:
        if(pd.isna(poi)==False):
            poi_id, num_of_facilities = poi.split(':')
            poi_class = poi_id.split('#')
            # combine the class1 and class2 numbers
            if(len(poi_class) == 1):
                poi_class[0] = '0' + poi_class[0]
            else:
                poi_number = poi_class[0] + '' + poi_class[1]
            weighted_sum += int(num_of_facilities) * int(poi_number)

    updated_list.append(weighted_sum)

# print(list_of_lists_poi_id)

# change labeled_poi_data['poi_ids'] to list_of_lists_poi_id
labeled_poi_data['poi_ids'] = updated_list

print('labeled_poi_data.head(): ', labeled_poi_data.head())

labeled_poi_data.to_csv('../dataset/labeledTestSet/poi_data.csv',
index=False)
```

This code reads in a POI dataset from a file and extracts the district_hash column and POI ID columns. It then combines all POI IDs for each row into a list and creates a new DataFrame with the district_hash and poi_ids_list. The code then converts the column of lists to a list of lists and sums up the number of facilities for each POI ID. Finally, the labeled_poi_data is saved to a CSV file.

Mapping time to time slots for orders and weather data:

```
# convert time to datetime
orders_data['time'] = pd.to_datetime(orders_data['time'])
weather_data['time'] = pd.to_datetime(weather_data['time'])


# map time to time slot
orders_data['time_slot'] = orders_data['time'].dt.hour * 6 + orders_data['time'].dt.minute // 10
weather_data['time_slot'] = weather_data['time'].dt.hour * 6 + weather_data['time'].dt.minute // 10


# map time to time slot with weekday
orders_data['weekday'] = orders_data['time'].dt.weekday
weather_data['weekday'] = weather_data['time'].dt.weekday


# remove the time column
orders_data = orders_data.drop(['time'], axis=1)
weather_data = weather_data.drop(['time'], axis=1)


print(orders_data.head())
print(weather_data.head())

# merge the poi_list class characteristics with the cluster_map
# cluster_map: region_hash, region_id
# poi_data: district_hash, poi_ids
# merge on district_hash
cluster_map_poi = pd.merge(cluster_map, poi_data, left_on='region_hash',
right_on='region_hash', how='inner')

# remove the region_hash column
# cluster_map_poi = cluster_map_poi.drop(['region_id'], axis=1)
```

```
print(cluster_map_poi.head())

# merge the orders_data with the cluster_map_poi
# orders_weather_data: start_district_hash, time_slot, weekday, order_gap, temperature, pm25
# cluster_map_poi: region_id, poi_ids
# merge on start_district_hash
# print(orders_weather_data.head())
# print(cluster_map_poi.head())
orders_weather_cluster_map_poi = pd.merge(orders_weather_data, cluster_map_poi,
left_on='start_region_hash', right_on='region_hash', how='inner')
orders_weather_cluster_map_poi=orders_weather_cluster_map_poi.rename(columns={'poi_ids'
: 'start_poi_ids'}).rename(columns={'region_id': 'start_region_id'})

orders_weather_cluster_map_poi = pd.merge(orders_weather_cluster_map_poi,
cluster_map_poi, left_on='dest_region_hash', right_on='region_hash', how='inner')
orders_weather_cluster_map_poi=orders_weather_cluster_map_poi.rename(columns={'poi_ids'
: 'dest_poi_ids'}).rename(columns={'region_id': 'dest_region_id'})

# remove the start_district_hash column
orders_weather_cluster_map_poi = orders_weather_cluster_map_poi.drop(['start_region_hash',
'dest_region_hash', 'region_hash_x', 'region_hash_y'], axis=1)

print(orders_weather_cluster_map_poi)


# save the data
orders_weather_cluster_map_poi.to_csv('../dataset/processedData/orders_weather_cluster_ma
p_poi_test.csv', index=False)
```

This code performs several data preprocessing and merging operations on three datasets: orders_data, weather_data, and cluster_map_poi. The first section of the code maps time to time slot and adds a weekday column to orders_data and weather_data. The next section merges cluster_map and poi_data on district_hash, then merges orders_weather_data with cluster_map_poi on start_district_hash and dest_district_hash. Finally, the resulting dataset is saved as a CSV file.

Merging and renaming dataframes in Python using pandas library:

```
# merge the orders_data with the cluster_map_poi
# orders_weather_data: start_district_hash, time_slot, weekday, order_gap,
temperature, pm25
# cluster_map_poi: region_id, poi_ids
# merge on start_district_hash
# print(orders_weather_data.head())
# print(cluster_map_poi.head())

orders_weather_cluster_map_poi = pd.merge(orders_weather_data,
cluster_map_poi, left_on='start_region_hash', right_on='region_hash',
how='inner')
orders_weather_cluster_map_poi=orders_weather_cluster_map_poi.rename(colum
ns={'poi_ids': 'start_poi_ids'}).rename(columns={'region_id':
'start_region_id'})

orders_weather_cluster_map_poi = pd.merge(orders_weather_cluster_map_poi,
cluster_map_poi, left_on='dest_region_hash', right_on='region_hash',
how='inner')
orders_weather_cluster_map_poi=orders_weather_cluster_map_poi.rename(colum
ns={'poi_ids': 'dest_poi_ids'}).rename(columns={'region_id':
'dest_region_id'})

# remove the start_district_hash column
orders_weather_cluster_map_poi =
orders_weather_cluster_map_poi.drop(['start_region_hash',
'dest_region_hash', 'region_hash_x', 'region_hash_y'], axis=1)

print(orders_weather_cluster_map_poi)


# save the data
orders_weather_cluster_map_poi.to_csv('../dataset/processedData/orders_wea
ther_cluster_map_poi_test.csv', index=False)
```

This code merges two dataframes, 'orders_weather_data' and 'cluster_map_poi', based on a common column 'start_region_hash'. The resulting merged dataframe is then modified to rename some columns and remove unnecessary columns. Finally, the resulting dataframe is saved as a csv file.

Training ML models & Finding MAE:

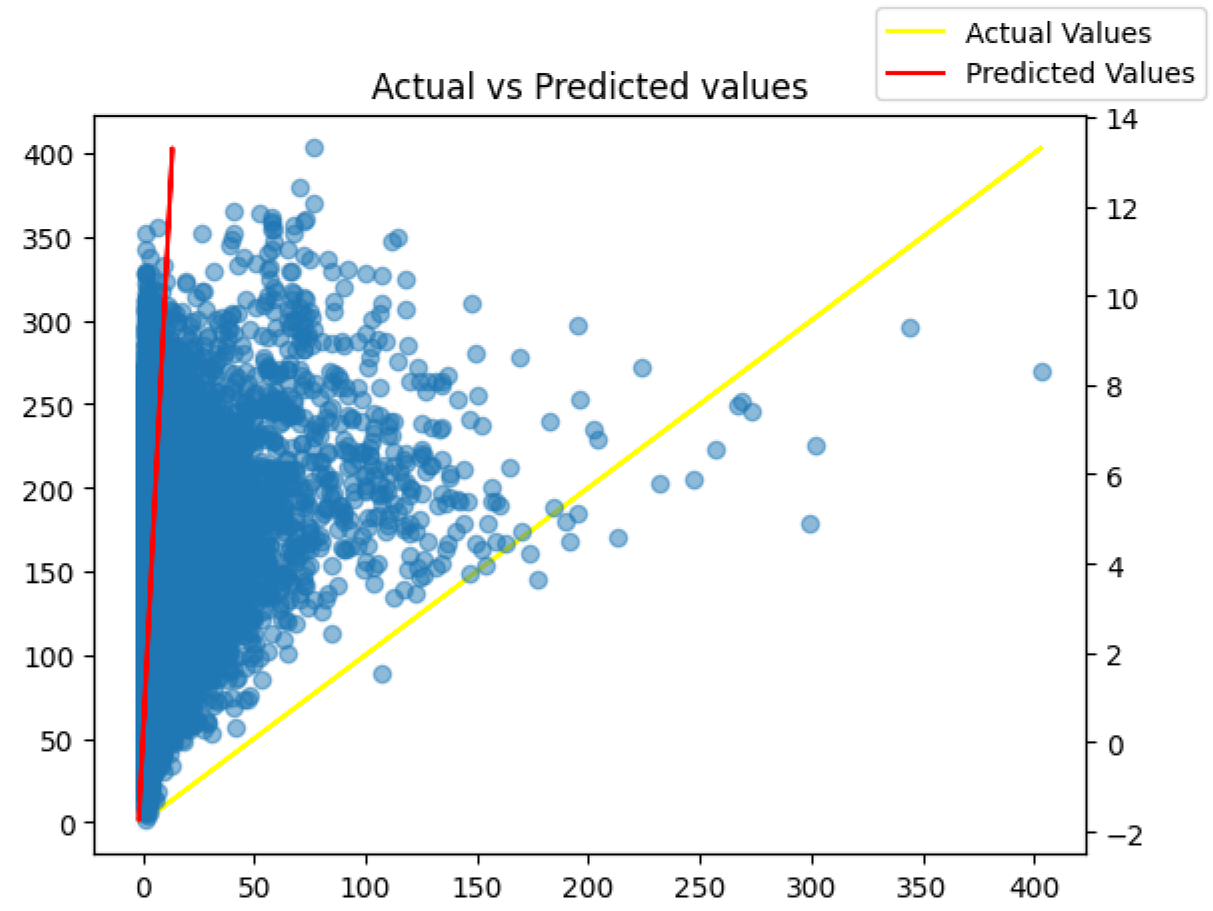
Code to Train and Predict Model on Weather and POI Data:

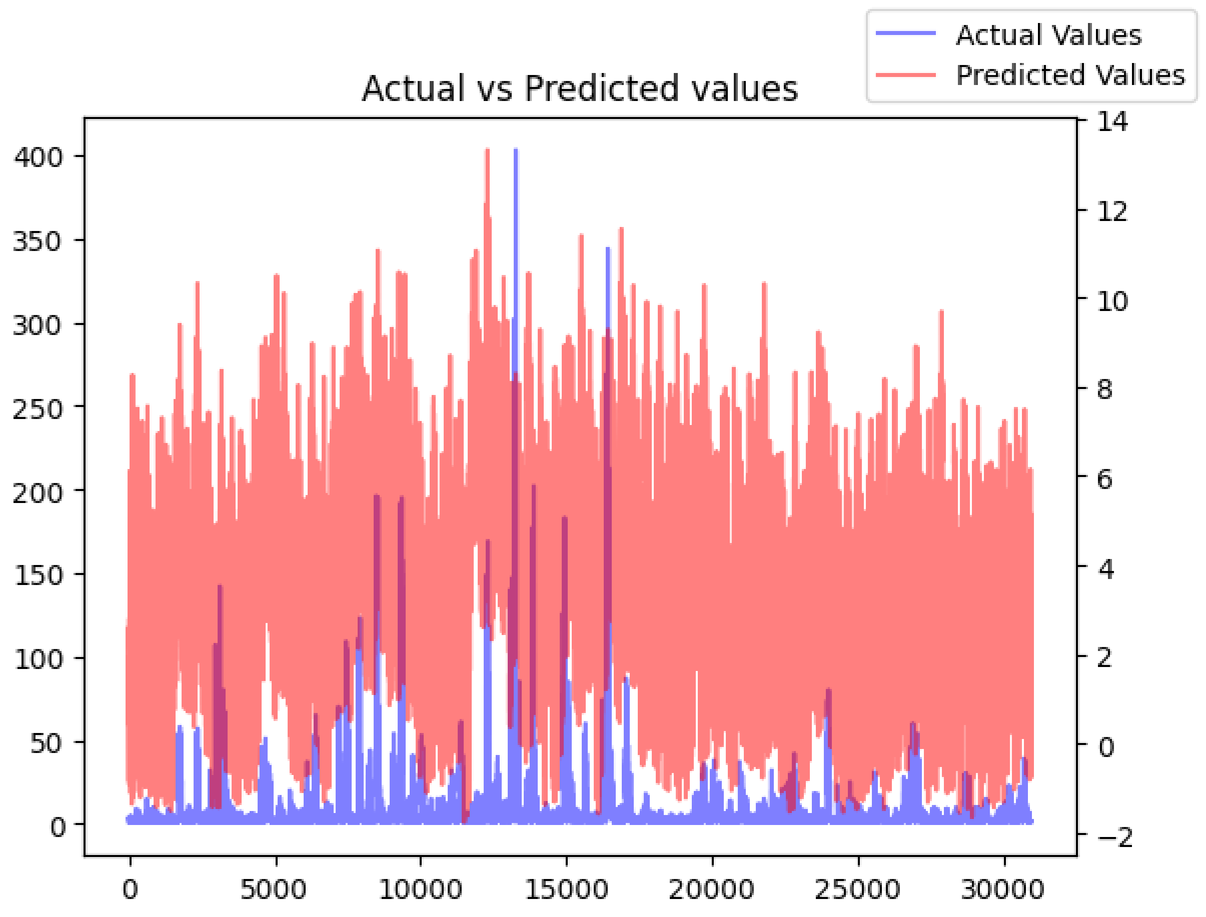
```
# Read the CSV file
data =
pd.read_csv('../dataset/processedData/orders_weather_cluster_map_poi.csv')
# laod test data
test_data =
pd.read_csv('../dataset/processedData/orders_weather_cluster_map_poi_test.
csv.csv')

def train_predict_model(model):

    # variables :
time_slot,weekday,order_gap,supply,demand,temperature,pm25,start_region_id
,start_poi_ids,dest_region_id,dest_poi_ids
    # dependent variable: order_gap, supply, demand
    # independent variables:
time_slot,weekday,temperature,pm25,start_region_id,start_poi_ids,dest_regi
on_id,dest_poi_ids

    # Split the data into independent and dependent variables
    X =
data[['time_slot','weekday','temperature','pm25','start_poi_ids','dest_poi
_ids']]
    y = data['order_gap']

    # Create a model and fit the data
    ml = model()
    ml.fit(X, y)

    # Predict the values for the test set
    y_pred =
ml.predict(test_data[['time_slot','weekday','temperature','pm25','start_po
i_ids','dest_poi_ids']])

    # Compute the R-squared score for the Linear Regression model
    accuracy = r2_score(test_data['supply'], y_pred)
    # Print the accuracy
    print("Accuracy of model:", accuracy)

    # Calculate the mean absolute error
    mae = np.mean(np.abs(test_data['supply'] - y_pred))

    # Print the result
    print('Mean Absolute Error:', mae)

    # Plot the predicted values against the actual values in scatter plot
    plot_scatter(test_data['supply'], y_pred)

    # Plot the predicted values against the actual values in line plot
    plot_line(test_data['supply'], y_pred)
```

This code loads two CSV files, one for training data and one for test data, containing weather and point of interest (POI) data. The function train_predict_model takes a machine learning model as input, splits the training data into independent and dependent variables, fits the model to the data, and predicts the dependent variable for the test data. The code then computes the accuracy of the model using the R-squared score and prints it, calculates the mean absolute error, and plots the predicted values against the actual values in scatter and line plots. This code can be used to train and test various machine learning models on weather and POI data.

Linear Regression Model:

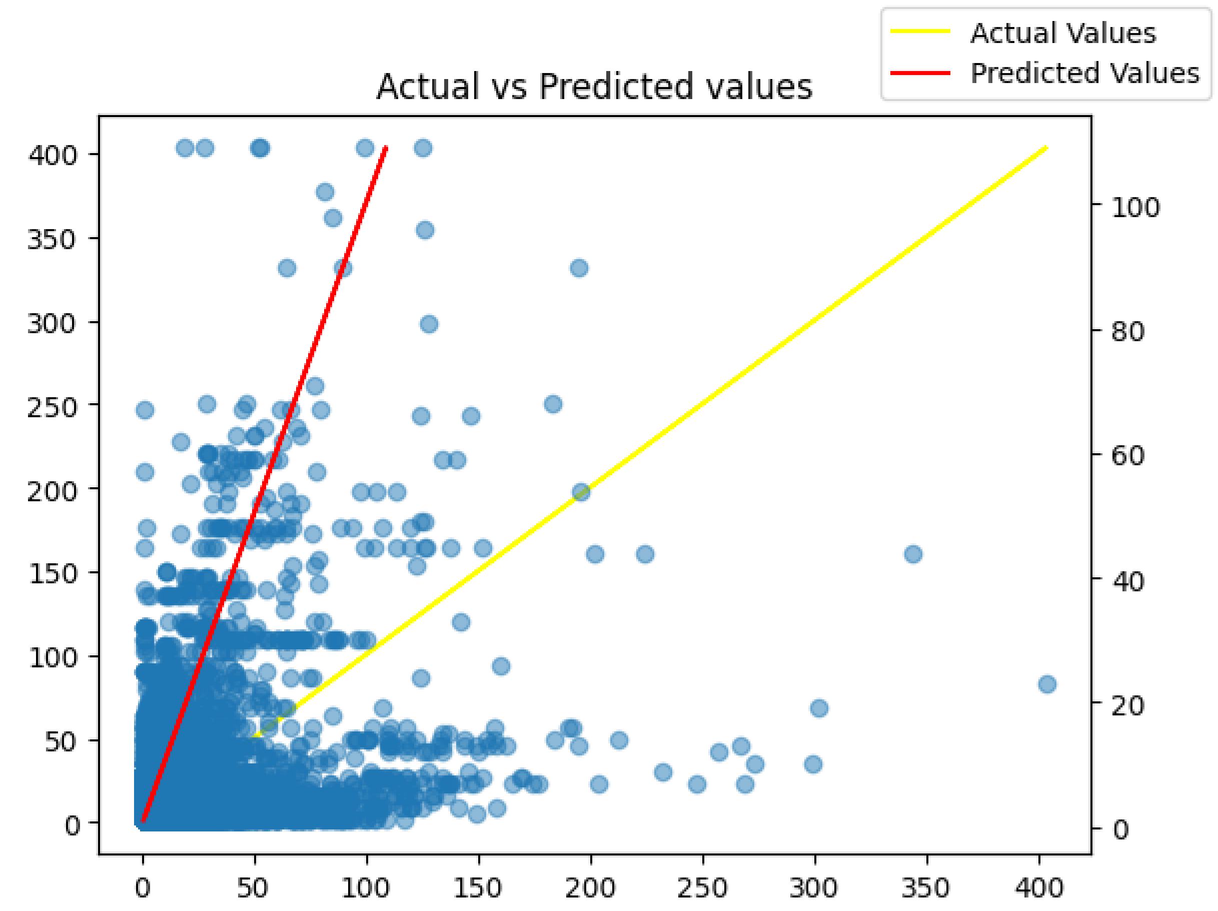
```
from sklearn.linear_model import LinearRegression

# Train and Predict using model
train_predict_model(LinearRegression)

Accuracy of model: 0.010337428834833795
Mean Absolute Error: 5.7491976298786005
```

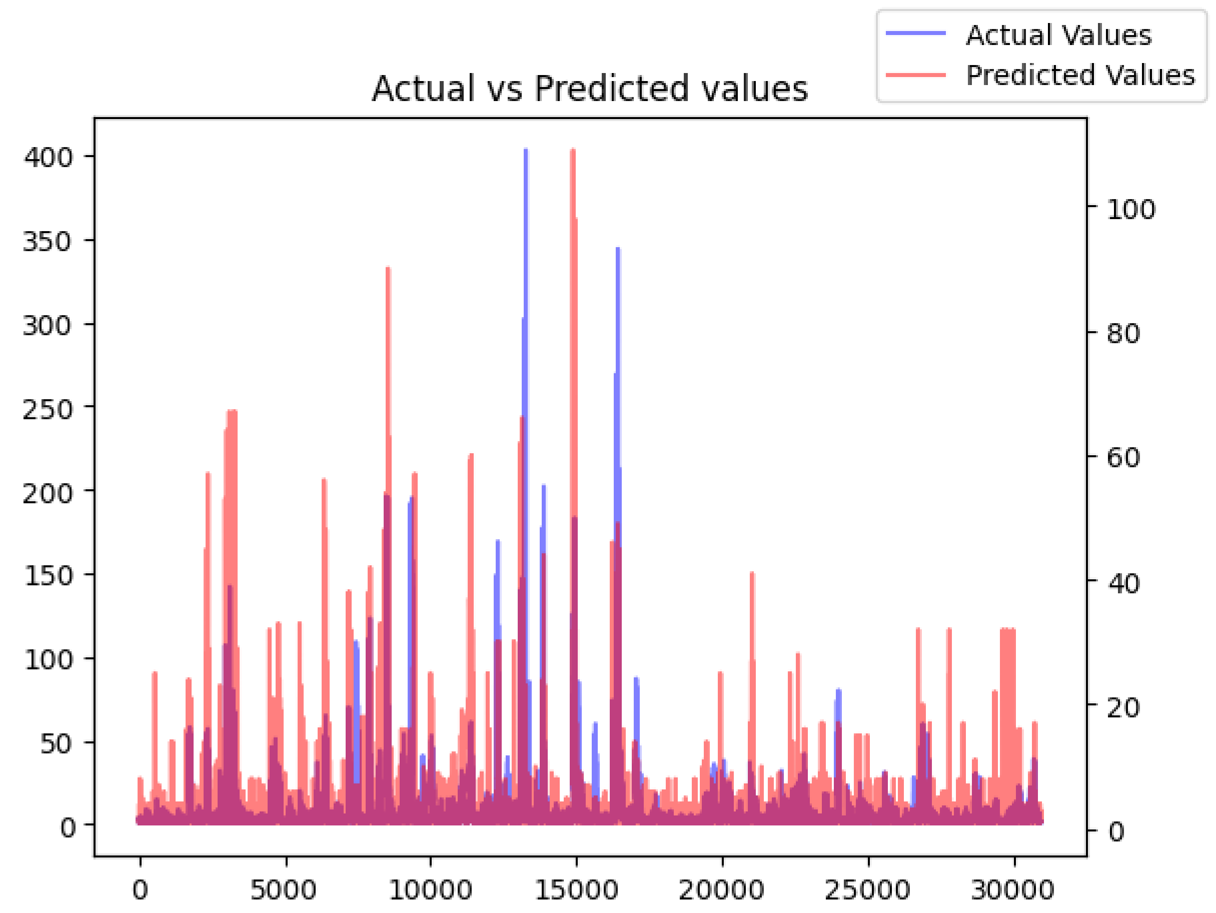

Actual vs Predicted values
Actual Values
Predicted Values
0
50
100
150
200
250
300
350
400
−2
0
2
4
6
8
10
12
14



Decision Tree Model:

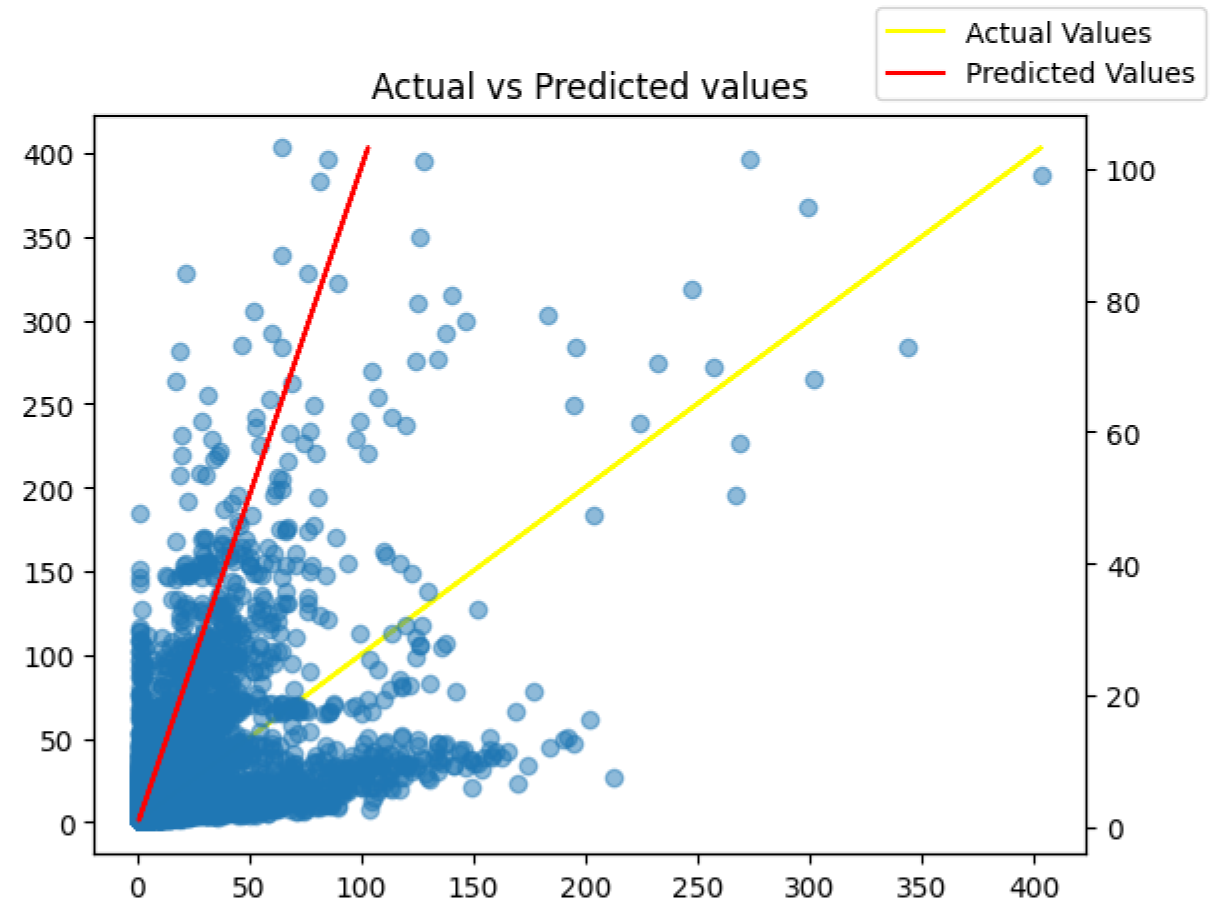
```
from sklearn.tree import DecisionTreeRegressor

# Create a Decision Tree Regression model and fit the data
train_predict_model(DecisionTreeRegressor)

Accuracy of model: 0.10016400077405407
Mean Absolute Error: 5.113798790544255
```

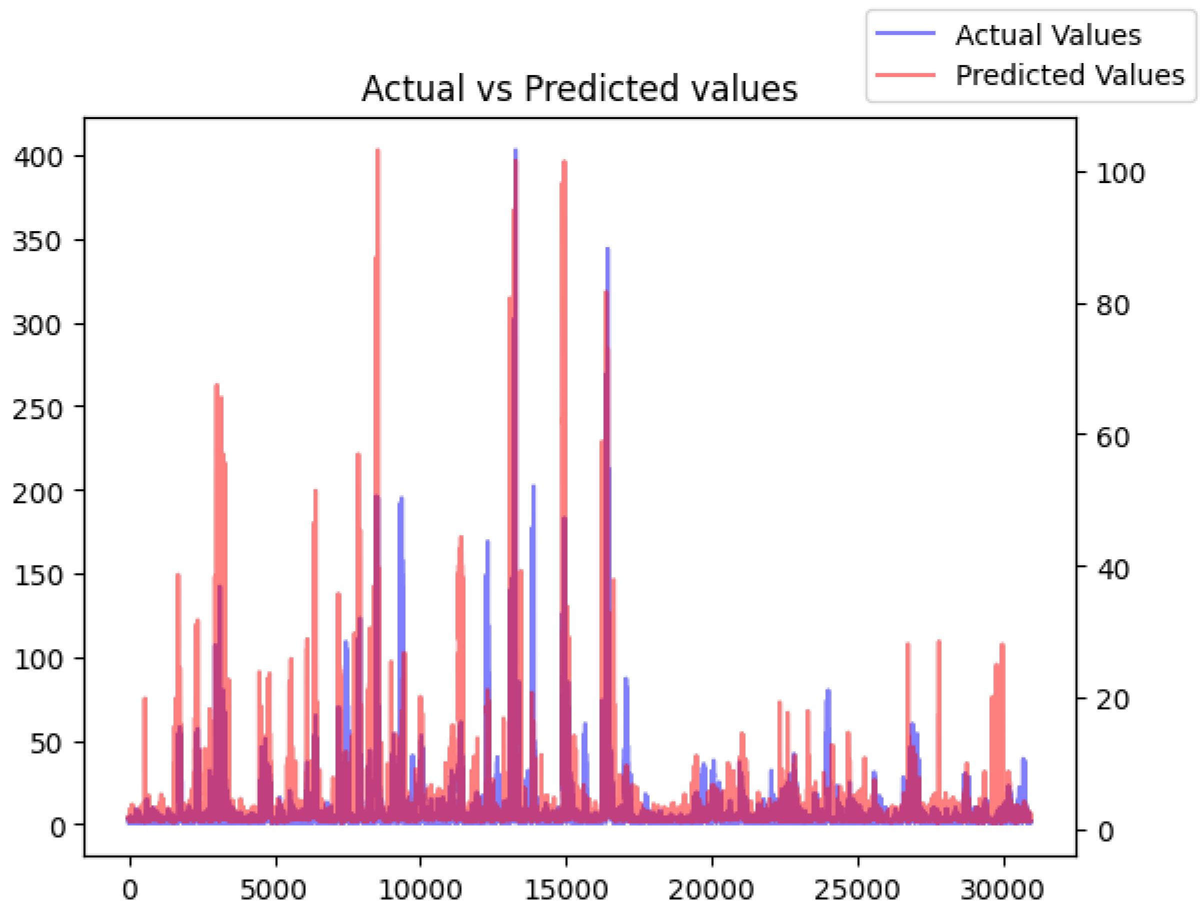

Actual vs Predicted values
Actual Values
Predicted Values
400
350
300
250
200
150
100
50
0
100
80
60
40
20
0
0
50
100
150
200
250
300
350
400



Random Forest Regression Model:

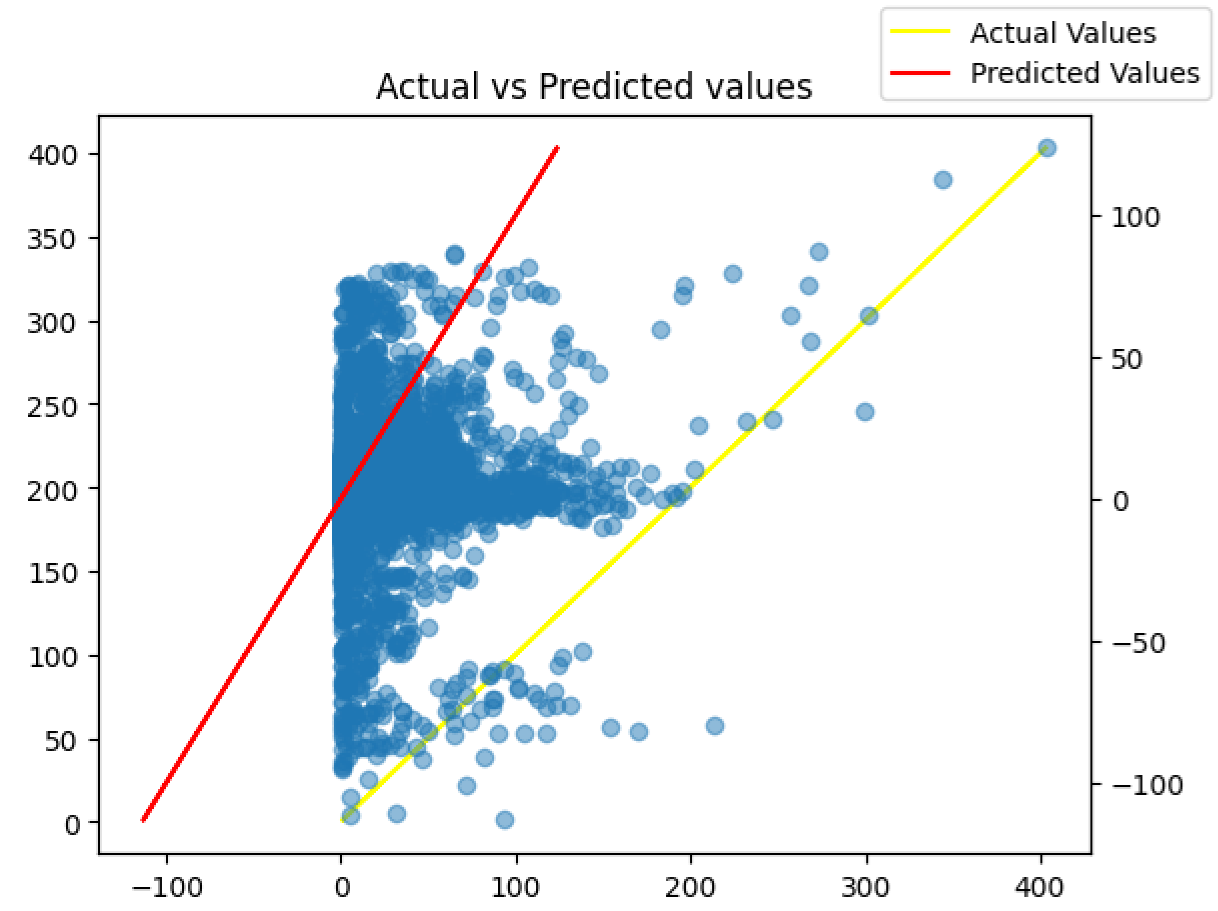
```
from sklearn.ensemble import RandomForestRegressor

# Create a Random Forest model and fit the data
train_predict_model(RandomForestRegressor)

Accuracy of model: 0.209783550383171 87
Mean Absolute Error: 4.750892216149792
```

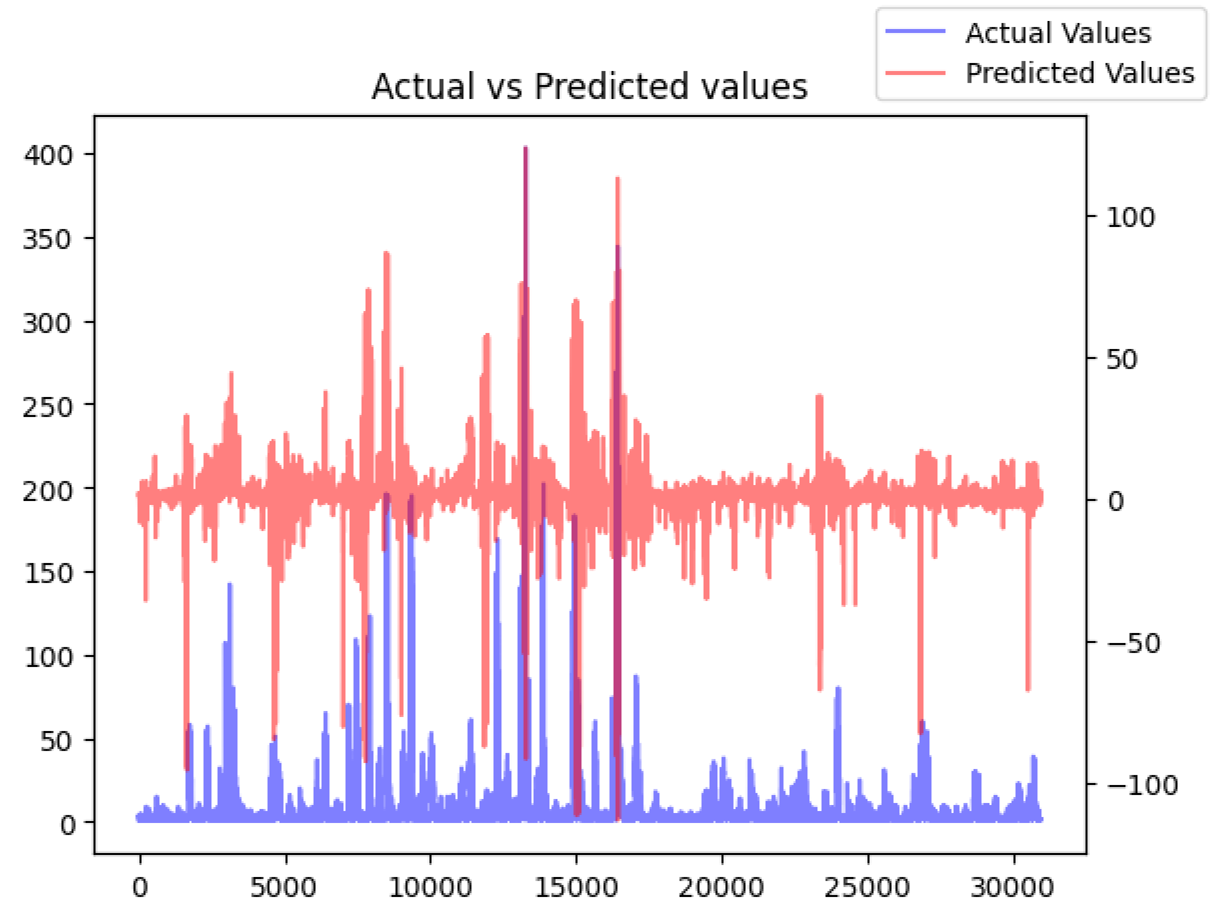

Actual vs Predicted values
Actual Values
Predicted Values
0
50
100
150
200
250
300
350
400
0
20
40
60
80
100



XGBoost Regression Model:

```
from xgboost import XGBRegressor

# Create a XGB model and fit the data
train_predict_model(XGBRegressor)

Accuracy of model: -0.32487380591908677
Mean Absolute Error: 6.847874969751133
```


Actual vs Predicted values
Actual Values
Predicted Values
400
350
300
250
200
150
100
50
0
100
50
0
−50
−100
−100
0
100
200
300
400



Plotting Rolling Window Line Graph of Actual vs Mean Predicted Values:

```
def plot_rolling_window_line(test, predictions, window_size):
    # Apply a rolling average to the predicted values
    rolling_predictions =
pd.Series(predictions).rolling(window_size).mean().values

    # Plot the mean predicted values against the actual values
    fig, ax1 = plt.subplots()
    ax1.plot(test_data['supply'], color='blue', label='Actual Values',
alpha=0.5)
    ax1.set_ylabel('Actual Values')
    ax2 = ax1.twinx()
    ax2.plot(rolling_predictions, color='red', label='Mean Predicted
Values', alpha=0.5)
    ax2.set_ylabel('Mean Predicted Values')
    plt.title("Actual vs Mean Predicted values")
    fig.legend()
    plt.show()
```

The code defines a function that takes in three parameters: test, predictions, and window_size. It then applies a rolling average to the predicted values using the rolling() function from the Pandas library. Next, the function creates a line graph with two y-axes, where the left axis represents the actual values, and the right axis represents the rolling mean predicted values. Finally, the function sets the title of the graph and displays it using plt.show().

Code for training and predicting a rolling window with a machine learning model:

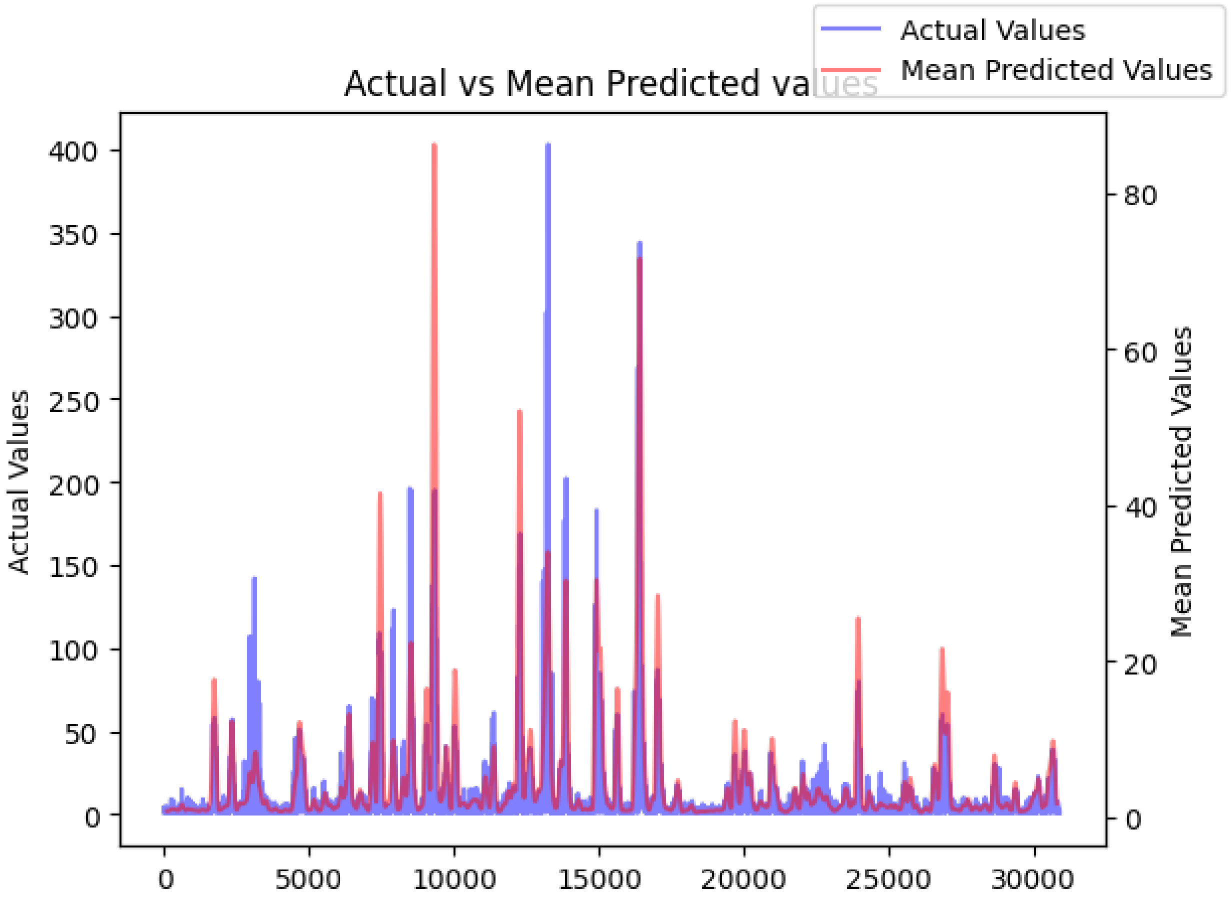
```
def train_predict_rolling_window(model, window_size):
    print('\t-------------\tWindow size:', window_size)

    # Split the data into independent and dependent variables
    # and select the first 100 windows of data
    X =
data[['time_slot','weekday','temperature','pm25','start_poi_ids','dest_poi
_ids']]
    y = data['order_gap']


    # Create a Linear Regression model
    ml = model()

    # Fit the model to the current window of data
    ml.fit(X, y)

    # Create an empty list to store the predicted values for each window
    predictions = []

    # Iterate through the data using a rolling window approach
    for i in range(window_size, len(test_data) + 1):
        # if i % window_size == 0:
            # print("Window number:", i//window_size)

        window = test_data.iloc[i - window_size : i, :]
        X_window =
window[['time_slot','weekday','temperature','pm25','start_poi_ids','dest_p
oi_ids']]
        y_window = window['supply']
        # Predict the values for the window
        y_pred = ml.predict(X_window)

        mae = np.mean(np.abs(y_window - y_pred))
        # Add the predicted values to the list
        predictions.append(mae)

    # Calculate the mean of the predicted values for all windows
    mean_prediction = np.mean(predictions, axis=0)

    # minimum prediction
    min_prediction = np.min(predictions, axis=0)

    # max prediction
    max_prediction = np.max(predictions, axis=0)

    # Print the result
    print('Minimum prediction:', min_prediction)
    print('Maximum prediction:', max_prediction)
    print('Mean Absolute Error for the mean prediction:', mean_prediction)

    plot_rolling_window_line(test_data['supply'], predictions,
window_size)
```

The function 'train_predict_rolling_window' takes in a model and a window size as input parameters. It first splits the data into independent and dependent variables and creates a linear regression model. Then, it uses a rolling window approach to iterate through the data and predict the values for each window. The mean, minimum, and maximum predictions are calculated and printed. Finally, a line plot of the rolling window predictions is plotted using the 'plot_rolling_window_line' function.

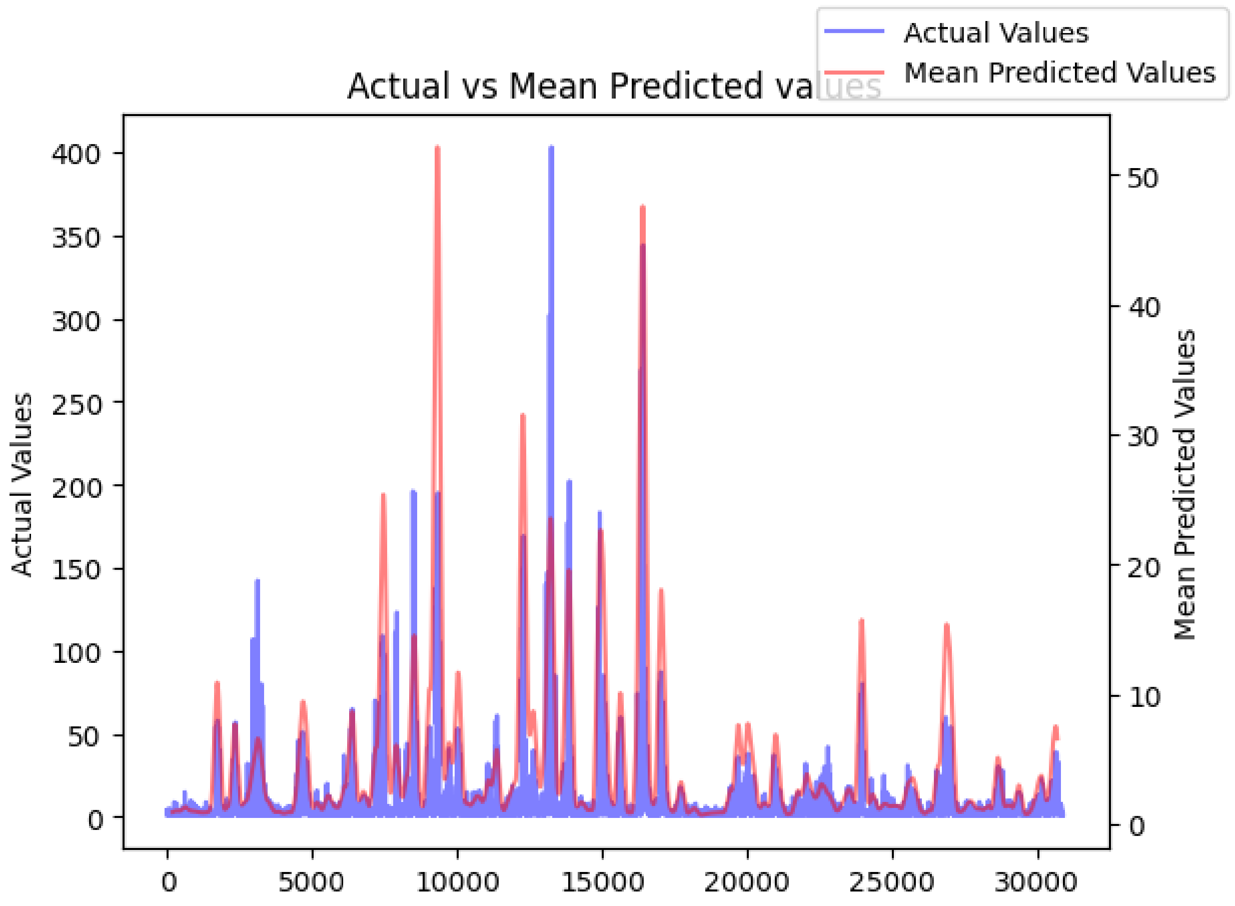
RFG Results:

```
from sklearn.ensemble import RandomForestRegressor

# Train and Predict using model with rolling window
train_predict_rolling_window(RandomForestRegressor, 100)
```

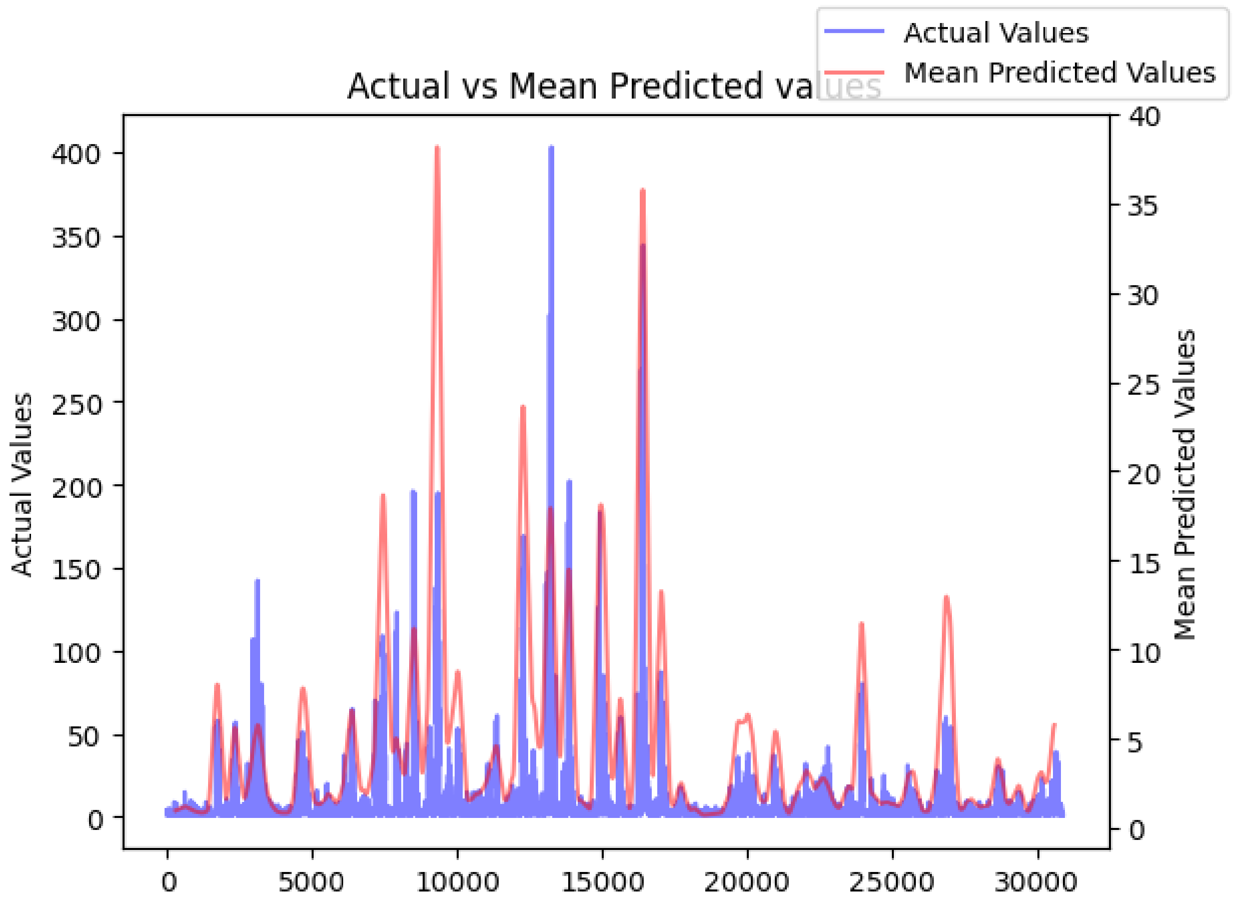
```
    -------------   Window size: 100
Minimum prediction: 0.5132
Maximum prediction: 104.66670000000002
Mean Absolute Error for the mean prediction: 4.7806549669089025
```



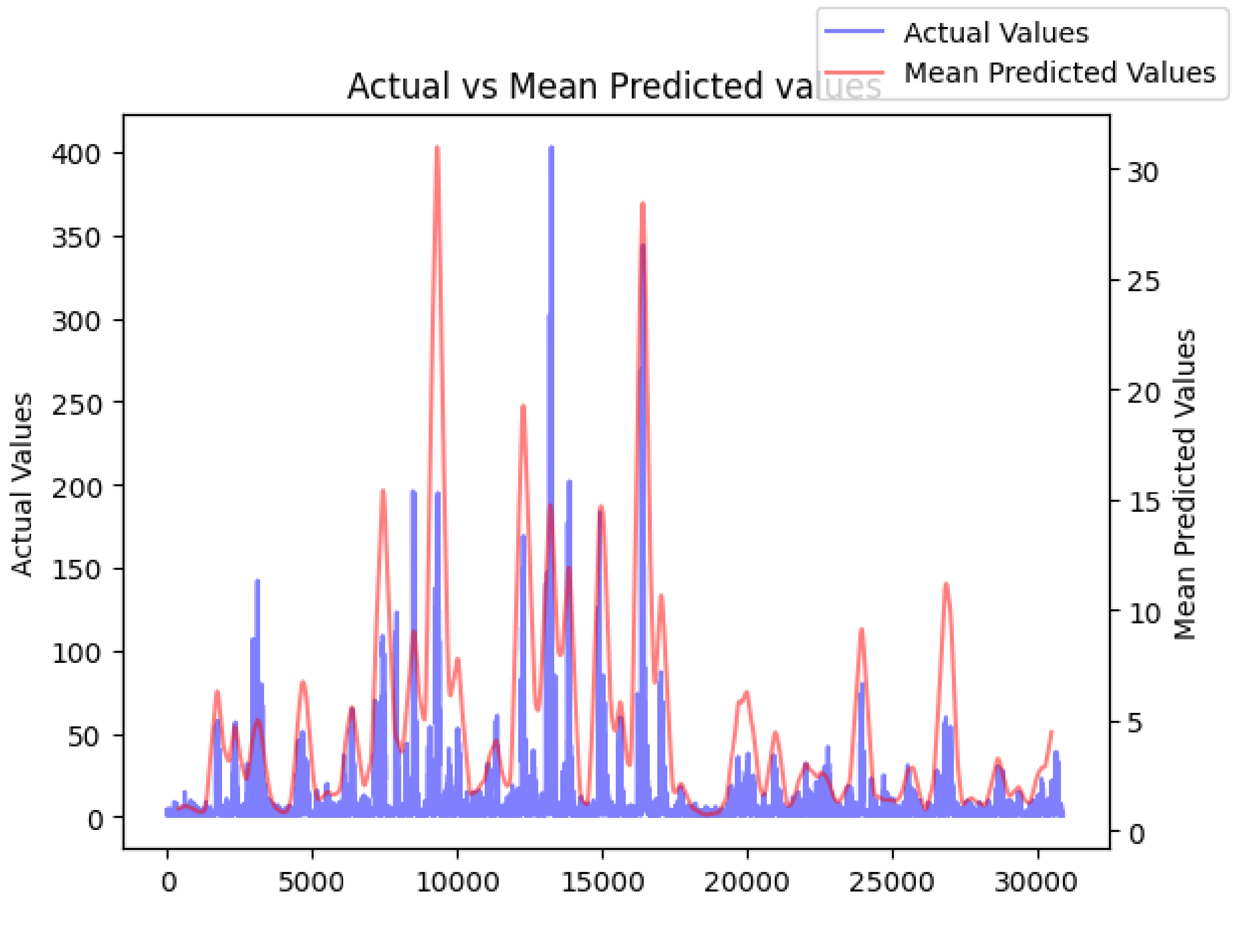
```
# Train and Predict using model with rolling window
train_predict_rolling_window(RandomForestRegressor, 200)



    -------------   Window size: 200
Minimum prediction: 0.5801000000000001
Maximum prediction: 60.449600000000004
Mean Absolute Error for the mean prediction: 4.783841020049473
```



```
# Train and Predict using model with rolling window
train_predict_rolling_window(RandomForestRegressor, 300)


    ------------   Window size: 300
Minimum prediction: 0.6860666666666666
Maximum prediction: 41.76106666666668
Mean Absolute Error for the mean prediction: 4.786338561259143
```



```
# Train and Predict using model with rolling window
train_predict_rolling_window(RandomForestRegressor, 400)



    -------------   Window size: 400
Minimum prediction: 0.6473
Maximum prediction: 35.92125
Mean Absolute Error for the mean prediction: 4.793592848250557
```


Actual Values
Mean Predicted Values
Actual vs Mean Predicted values
Actual Values
Mean Predicted Values
400
350
300
250
200
150
100
50
0
30
25
20
15
10
5
0
0
5000
10000
15000
20000
25000
30000