Configuration Manual

MSc Research Project
Data Analytics

Tejas Sanjay Shinde
Student ID: 18180159

School of Computing
National College of Ireland

Supervisor: Mr. Hicham Rifai

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Tejas Sanjay Shinde
Student ID:	18180159
Programme:	Data Analytics
Year:	2020
Module:	MSc Research Project
Supervisor:	Mr. Hicham Rifai
Submission Due Date:	28/09/2020
Project Title:	Configuration Manual
Word Count:	1382
Page Count:	16

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	27th September 2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Tejas Sanjay Shinde
18180159

1 Introduction

The presented configuration manual depicts the specifications of the utilized hardware and software along with the detailed implementation followed in the presented study titled "**Parking availability prediction in the Seattle city using spatio-temporal features**"

2 System Configuration

2.1 Hardware Configuration

In the presented study OpenStack cloud instance (NAME : NCI_0159) provided by the National College of Ireland is used as Infrastructure as a Service(IaaS). The configuration of the machine created on OpenStack is given in the Figure 1. The instance is secured using SSH-key.

Hardware Configuration	
Instance	Ubuntu-Bionic 18.04.3
RAM	16 GB
Virtual CPU's	8
Hard Disk Storage	160 GB
Availability Zone	nova

Figure 1: Hardware

2.2 Software Configuration

All the softwares used for this study along with their versions are presented below:

Software Configuration	
Sublime	3.1.1
Apache Spark	3.0.0
Python	3.7.6
Scala	2.12.10
Anaconda	4.8.2
Java	1.8.0_265

Figure 2: Software

Sublime Text Editor:
Sublime is used as a tool to write the Scala code which makes it easy to code.

Apache Spark and Scala:
Apache Spark[1] is a framework that provides an environment which assists in the distributed processing of the big data. Apache provides support via different API's out of which Scala is used in this study. Due to its parallel processing it is used for the complex preprocesing involved in the computation of the distances from the nearest public transport stations and financial centre of the city, which is explained in the Section 3.1.2.

Anaconda and Jupyter Notebook:
Jupyter Notebook development environment for Python provided by the Anaconda distribution[2] is used in this study. All the python related processing and model implementation code is run in this software.

Python Libraries:
Python[3] is used for the processing and machine learning involved in the parking availability prediction. It provides support for several libraries which are utilized in the presented study. The version and description of those libraries is presented in the Figure 3:

Package	Version	Package	Version
pandas	1.0.1	plotly	4.8.2
matplotlib	3.1.3	scikit-learn	0.22.1
numpy	1.18.1	xgboost	1.1.1
scipy	1.4.1	eli5	0.30.1

Figure 3: Hardware

Java:
Java is installed as it is a pre-requisite for the above mentioned packages

3 Project Development

3.1 Data Preparation

Majority of the codes executed in the data preparation are executed using Python. However, the initial computation of the nearest transport stations in this study is carried out using Scala, which is explained in the Section 3.1.2

3.1.1 Loading the parking data

At first, the parking data in the Seattle city in acquire via an API[4] from July 2019 to Dec 2020 in four different files for 100 parking segments because of the API restrictions. This is done using Python's JSON library and the data is then stored into pandas dataframe. All of these dataframes are combined into a single file and stored in a CSV format as "CombinedParking.csv" in the "bigdataparking" folder shown in the Figure 4.

[1] https://spark.apache.org/downloads.html
[2] https://www.anaconda.com/products/individual
[3] https://www.python.org/downloads/
[4] https://data.seattle.gov/Transportation/2019-Paid-Parking-Occupancy-Year-to-date-/qktt-2bsy

```python
#Loading minute level data of 100 road segments from July 2019 to Dec 2019
url_Jul = "https://data.seattle.gov/resource/qktt-2bsy.json?$limit=28500000&$where=(date_extract_m(occupancydatetime)=7)\
%20AND%20(sourceelementkey<10000)"
url_Aug = "https://data.seattle.gov/resource/qktt-2bsy.json?$limit=28500000&$where=(date_extract_m(occupancydatetime)=8)\
%20AND%20(sourceelementkey<10000)"
url_Sept = "https://data.seattle.gov/resource/qktt-2bsy.json?$limit=28500000&$where=(date_extract_m(occupancydatetime)=9)\
%20AND%20(sourceelementkey<10000)"
url_Oct = "https://data.seattle.gov/resource/qktt-2bsy.json?$limit=28500000&$where=(date_extract_m(occupancydatetime)=10)\
%20AND%20(sourceelementkey<10000)"
url_Nov = "https://data.seattle.gov/resource/qktt-2bsy.json?$limit=28500000&$where=(date_extract_m(occupancydatetime)=11)\
%20AND%20(sourceelementkey<10000)"
url_Dec = "https://data.seattle.gov/resource/qktt-2bsy.json?$limit=28500000&$where=(date_extract_m(occupancydatetime)=12)\
%20AND%20(sourceelementkey<10000)"
```

```python
#Loading minute level data of 100 road segments from July 2019 to Dec 2019 with Json
response_Jul = urllib.request.urlopen(url_Jul)
response_Aug = urllib.request.urlopen(url_Aug)
response_Sept = urllib.request.urlopen(url_Sept)
response_Oct = urllib.request.urlopen(url_Oct)
response_Nov = urllib.request.urlopen(url_Nov)
response_Dec = urllib.request.urlopen(url_Dec)
```

```python
# Reading from Json
data_Jul = json.loads(response_Jul.read())
data_Aug = json.loads(response_Aug.read())
data_Sept = json.loads(response_Sept.read())
data_Oct = json.loads(response_Oct.read())
data_Nov = json.loads(response_Nov.read())
data_Dec = json.loads(response_Dec.read())
```

```python
# Converting into Dataframes
Aug = pd.DataFrame(data_Aug)
Sept = pd.DataFrame(data_Sept)
Oct = pd.DataFrame(data_Oct)
Nov = pd.DataFrame(data_Nov)
Dec = pd.DataFrame(data_Dec)
```

```python
#Combining all the dataframes
combined = pd.concat([Aug, Sept], ignore_index=True)
combined = pd.concat([combined, Oct], ignore_index=True)
combined = pd.concat([combined, Nov], ignore_index=True)
combined = pd.concat([combined, Dec], ignore_index=True)
combined.to_csv("/home/ubuntu/bigdataparking/CombinedParking.csv")
```

Figure 4: Seattle Parking Data

3.1.2 Initial processing and Computation of distance from the nearby public transport station and city financial centre

The "CombinedParking.csv" file from Section 3.1.1 is then loaded into a Scala code as shown in Figure 5. From this the hour, min, latitude, longitude of the parking segment is extracted and stored in new columns as shown in figure below. After this the longitude and latitude columns are converted in float and a new dataframe is created which shows the coordinates and id of each parking segment. This new dataframe is then used to calculate the distances as shown in Figures the 7 and 8

```scala
// Loading the combined parking data from July 2019 to Dec 2020
var df = spark.read.format("csv").option("header","true").load("/home/ubuntu/bigdataparking/CombinedParking.csv")

// Selecting the required features features
var df2 = df.select("occupancydatetime", "paidoccupancy", "sideofstreet", "sourceelementkey", "parkingtimelimitcategory", "parkingspacecount", "location")

// Extracting the Longitude and Latitude
var df3 = df2.withColumn("loc", regexp_extract(col("location"), "(?<=\\[)(.*)(?=\\])", 0))
df3 = df3.withColumn("p_lon", split($"loc", ", ").getItem(0))
df3 = df3.withColumn("p_lat", split($"loc", ", ").getItem(1))

// Extracting the Hour and Min
df3 = df3.withColumn("Date", split($"occupancydatetime", "T").getItem(0))
df3 = df3.withColumn("Timestamp", split($"occupancydatetime", "T").getItem(1))
df3 = df3.withColumn("Hour", split($"Timestamp", ":").getItem(0))
df3 = df3.withColumn("Min", split($"Timestamp", ":").getItem(1))

// Droping the original columns which are not required
df3 =  df3.drop("loc")
df3 =  df3.drop("location")
df3 =  df3.drop("Timestamp")
df3 =  df3.drop("occupancydatetime")


// Changing the datatype of Longitude and Latitude to float
df3 = df3.withColumn("p_lon",col("p_lon").cast("float")).withColumn("p_lat",col("p_lat").cast("float"))

// Datframe which will state the Longitude and Latitude of each parking segment
var df4 = df3.select("sourceelementkey","p_lon", "p_lat")
df4 = df4.distinct()
```

Figure 5: Cleaning the Seattle Parking Data

The stops, trips, routes and stop_items textual files obtained from the King county repository[5] are loaded into Scala as shown in Figure 6. All of these files are combined using unique stop, trip and route identifiers. This gives the important columns such as the "stop_id", coordinates and their respective "route_id". This helps us to identify the public transport station and its mode of transport such as Bus, Rail or Ferry.

[5]https://kingcounty.gov/depts/transportation/metro/travel-options/bus/app-center/developer-resources.aspx

```
// Loading the stops, stop_times, trips and routes data files which specify the public transport stations in the Seattle
var stop = spark.read.format("csv").option("header","true").load("/home/ubuntu/stops.txt")
var stop_times = spark.read.format("csv").option("header","true").load("/home/ubuntu/stop_times.txt")
var trips = spark.read.format("csv").option("header","true").load("/home/ubuntu/trips.txt")
var route = spark.read.format("csv").option("header","true").load("/home/ubuntu/routes.txt")

// Selecting only the limited feature such as longitude and Latitude
stop = stop.select("stop_id","stop_lat","stop_lon")
stop_times = stop_times.select("trip_id","stop_id")
trips = trips.select("route_id","trip_id")
route = route.select("route_id","route_type")

// Joining all of these tables to get the stop_id, stop_lat, stop_lon and the route_type
val trans  = stop.join(stop_times, Seq("stop_id"), "left")
var trans2  = trans.join(trips, Seq("trip_id"), "left")
var trans3  = trans2.join(route, Seq("route_id"), "left")
var trans4 = trans3.select("stop_lat","stop_lon","route_type")
trans4 = trans4.distinct() //removing the duplicates
// Converting the datatype to float
trans4 = trans4.withColumn("stop_lat", col("stop_lat").cast("float")).withColumn("stop_lon", col("stop_lon").cast("float"))
```

Figure 6: Bus, Rail, and Ferry Coordinates

The code shown in Figure 7 then combines the datasets containing parking coordinates 5 and transport station coordinates 6, which are used to identify the distance between each parking segment and transport station such as Rail, Bus, and Ferry. Post this the closest distance from rail, bus and ferry station is identified using min function in SQL select statement as shown below. Similarly, the distance from the airport and financial centre are calculated as shown in Figure 8. The coordinates of the airport and financial centre are available on google maps. Post this all of these dataframes are combined. For both these operations Haversine's formula (Winarno et al.; 2017) is used as shown in the figure below. Due to the huge size of the data only the 49 parking locations within 1 km from the city center are selected. This results in 43,63,631 rows. The final dataframe is then stored in "bigdataparking/FinalData". Initially a long name was assigned automatically by Scala command which was then renamed to"ProcessedParkingData.csv" for ease of use. **Note that all of these scala commands are executed into the spark-shell.**

```
// Finding the nearby ferry station from all the parking segments
var stoptype4 = trans4.filter(trans4("route_type") === 4) // filtering by route_type 4 which is ferry
// Cross joining the both these tables to get the coordinates of each parking segment from each ferry station
var dist_route4 = df4.crossJoin(stoptype4)
//calculting the distance between each parking segment from each ferry station
dist_route4 = dist_route4.withColumn("a", pow(sin(radians($"stop_lat" - $"p_lat") / 2), 2)  +
    cos(radians($"p_lat")) * cos(radians($"stop_lat")) *
    pow(sin(radians($"stop_lon" - $"p_lon") / 2), 2)).withColumn("distance", atan2(sqrt($"a"), sqrt(-$"a" + 1)) * 2 * 6371)
// Selecting the smallesy distances from the above table for each segment which shows the closest ferry station
dist_route4.createOrReplaceTempView("table")
var q = "SELECT sourceelementkey,distance FROM (SELECT *, MIN(distance) OVER (PARTITION BY sourceelementkey)
AS MinB FROM table) M WHERE distance = MinB"
var dist_route4final = spark.sql(q)
dist_route4final = dist_route4final.withColumnRenamed("distance","NearByFerry") //final table showing distance from the nearest ferry station


// Finding the nearby bus station from all the parking segments
var stoptype3 = trans4.filter(trans4("route_type") === 3)// filtering by route_type 3 which is bus
// Cross joining the both these tables to get the coordinates of each parking segment from each bus station
var dist_route3 = df4.crossJoin(stoptype3)
//calculting the distance between each parking segment from each bus station
dist_route3 = dist_route3.withColumn("a", pow(sin(radians($"stop_lat" - $"p_lat") / 2), 2)  +
 cos(radians($"p_lat")) * cos(radians($"stop_lat")) *
 pow(sin(radians($"stop_lon" - $"p_lon") / 2), 2)).withColumn("distance", atan2(sqrt($"a"), sqrt(-$"a" + 1)) * 2 * 6371)
// Selecting the smallesy distances from the above table for each segment which shows the closest ferry station
dist_route3.createOrReplaceTempView("table")
var q = "SELECT sourceelementkey,distance FROM (SELECT *, MIN(distance) OVER (PARTITION BY sourceelementkey)
AS MinB FROM table) M WHERE distance = MinB"
var dist_route3final = spark.sql(q)
dist_route3final = dist_route3final.withColumnRenamed("distance","NearByBus")//final table showing distance from the nearest bus station


// Finding the nearby rail station from all the parking segments
var stoptype0 = trans4.filter(trans4("route_type") === 0)// filtering by route_type 0 which is rail
// Cross joining the both these tables to get the coordinates of each parking segment from each rail station
var dist_route0 = df4.crossJoin(stoptype0)
//calculting the distance between each parking segment from each rail station
dist_route0 = dist_route0.withColumn("a", pow(sin(radians($"stop_lat" - $"p_lat") / 2), 2)  +
 cos(radians($"p_lat")) * cos(radians($"stop_lat")) *
 pow(sin(radians($"stop_lon" - $"p_lon") / 2), 2)).withColumn("distance", atan2(sqrt($"a"), sqrt(-$"a" + 1)) * 2 * 6371)
// Selecting the smallesy distances from the above table for each segment which shows the closest rail station
dist_route0.createOrReplaceTempView("table")
var q = "SELECT sourceelementkey,distance FROM (SELECT *, MIN(distance) OVER (PARTITION BY sourceelementkey)
AS MinB FROM table) M WHERE distance = MinB"
var dist_route0final = spark.sql(q)
dist_route0final = dist_route0final.withColumnRenamed("distance","NearByRail")//final table showing distance from the nearest rail station


// Combining all of these tables to get the final table which shows distances of each parking segment from all the public transport stations
var dist_route = dist_route3final.join(dist_route0final,Seq("sourceelementkey"), "left") // parking rail and bus
dist_route = dist_route.join(dist_route4final,Seq("sourceelementkey"), "left") // parking rail , bus and ferry
```

Figure 7: Distance from Nearest Bus, Rail, and Ferry Stations

```
// Calculating distance from city centre(Downtown Seattle)
// City coordinates
df4 = df4.withColumn("city_lon", lit(-122.3321))
df4 = df4.withColumn("city_lat", lit(47.6062))
df4 = df4.withColumn("city_lat", col("city_lat").cast("float")).withColumn("city_lon", col("city_lon").cast("float"))
var df4city = df4.withColumn("a", pow(sin(radians($"city_lat" - $"p_lat") / 2), 2)  +
    cos(radians($"p_lat")) * cos(radians($"city_lat")) *
    pow(sin(radians($"city_lon" - $"p_lon") / 2), 2)).withColumn("DistCity", atan2(sqrt($"a"), sqrt(-$"a" + 1)) * 2 * 6371)
df4city  = df4city.select("sourceelementkey","DistCity") // combining with the above dataset
dist_route = dist_route.join(df4city,Seq("sourceelementkey"), "left")// parking rail , bus, ferry, and city


// Distance from the international airport
// Coordinates
df4 = df4.withColumn("air_lon", lit(-122.3018))
df4 = df4.withColumn("air_lat", lit(47.5282))
df4 = df4.withColumn("air_lat", col("air_lat").cast("float")).withColumn("air_lon", col("air_lon").cast("float"))
var df4air = df4.withColumn("a", pow(sin(radians($"air_lat" - $"p_lat") / 2), 2)  +
 cos(radians($"p_lat")) * cos(radians($"air_lat")) *
 pow(sin(radians($"air_lon" - $"p_lon") / 2), 2)).withColumn("DistAir", atan2(sqrt($"a"), sqrt(-$"a" + 1)) * 2 * 6371)
df4air  = df4air.select("sourceelementkey","DistAir")
dist_route = dist_route.join(df4air,Seq("sourceelementkey"), "left")// parking rail , bus, ferry, city, and airport



// Combing all the datasets using sourceelementkey
var df5 = df3.join(dist_route,Seq("sourceelementkey"), "left")
var df6 = df5.filter(df5("DistFinCentre") <  1).show(false) // selecting only parking locations within 1 km

// Storing the combined final data file on the system in CSV format
df6.repartition(1).write.format("csv").option("header","true").save("/home/ubuntu/bigdataparking/FinalData")
```

Figure 8: Distance from city centre and airport

3.1.3 Loading weather data

The weather data is extracted via an API provdied by NOAA[6]. A loop is executed to automatically fetch the daily weather from July 2019 to Dec 2019 using JSON and combine them into a single dataframe as show in Figure 9. The weather dataframe is then stored into a CSV file as "weather.csv" in the same "bigdataparking" folder where the processed parking data is stored.

```
#The access token you got from NOAA
Token = 'gaGCFEBrHyeiSMXfTQrMQufkDLMRZypd'

#Loop to exctract data from July 2019 to Dec 2019
for month in ['07','08','09','10','11','12']:
    print('Working on month '+ month)
    if month in(['07','08','10','12']):
        #If odd months
        r = requests.get('https://www.ncdc.noaa.gov/cdo-web/api/v2/data?datasetid=GHCND&\
        limit=1000&stationid=GHCND:USW00024234&startdate=2019-'
                        +month+'-01&enddate=2019-'+month+'-31', headers={'token':Token})
    else:
        #If even months
        r = requests.get('https://www.ncdc.noaa.gov/cdo-web/api/v2/data?datasetid=GHCND&\
        limit=1000&stationid=GHCND:USW00024234&startdate=2019-'
                        +month+'-01&enddate=2019-'+month+'-30', headers={'token':Token})

    d = json.loads(r.text)

    #Creating list
    avg_temps_max = [item for item in d['results'] if item['datatype']=='TMAX']
    avg_temps_min = [item for item in d['results'] if item['datatype']=='TMIN']
    avg_temps_awnd = [item for item in d['results'] if item['datatype']=='AWND']
    avg_temps_prcp = [item for item in d['results'] if item['datatype']=='PRCP']
    avg_temps_pgtm = [item for item in d['results'] if item['datatype']=='PGTM']

    date,tmax,tmin,tawnd,tprcp,tpgtm = []

    date += [item['date'] for item in avg_temps_max]
    tmax += [item['value'] for item in avg_temps_max]
    tmin += [item['value'] for item in avg_temps_min]
    tawnd += [item['value'] for item in avg_temps_awnd]
    tprcp += [item['value'] for item in avg_temps_prcp]
    tpgtm += [item['value'] for item in avg_temps_pgtm]

    #Creating dataframe from list
    weather_new = pd.DataFrame(list(zip(date,tmax,tmin,tawnd,tprcp)), columns =['Date', 'TMAX', 'TMIN', 'AWND', 'PRCP'])

    #combining all the dataframes
    if month == '07':
        weather = weather_new
    else:
        weather = pd.concat([weather, weather_new], ignore_index=True)

#Saving to CSV to storage
weather.to_csv("/home/ubuntu/bigdataparking/weather.csv")
```

Figure 9: Weather Data

[6]https://www.ncdc.noaa.gov/cdo-web/webservices/v2

3.1.4 Identifying the missing values

Here, the processed parking data file named "ProcessedParkingData.csv" form the "bigdataparking" folder is loaded and missing values are identified using the code specified in the Figure 10. However, only the holidays and Sundays are found to be missing. Also, the "weather.csv" file is loaded from the "bigdataparking" folder and the missing values are identified as depicted in the code in the Figure 11. Three missing values can observed which are imputed with the help of interpolate function.

```
parking = pd.read_csv('/home/ubuntu/bigdataparking/FinalData/ProcessedParkingData.csv')

#Generating a validation dataframe with all days between 1st July 2019 and 31st Dec 2019, which will be
dates = pd.DataFrame(pd.date_range(start='2019-07-01', end='2019-12-31',freq='1D'),columns = ['Date'])

#Converting date into string
dates['Date'] = dates['Date'].astype('str')

#Merging Seatte parking dataframe with the validation dataframe to identify the missing dates
missingdates = pd.merge(dates,parking, on="Date", how="left")

#Checking missing dates
missingdates = missingdates[missingdates['paidoccupancy'].isna()]

#Exctracting only missing dates
missingdates = missingdates[['Date']]

#Presenting the missing days
len(missingdates)

31
```

Figure 10: Missing value in Processed Parking data

```
#Merging dates dataset created above with weather to identify missing days
weather = pd.merge(dates,weather, on="Date", how="left")

#Weather data is missing for three days
weather[weather['TMAX'].isna()]
```

	Date	TMAX	TMIN	AWND	PRCP
91	2019-09-30	NaN	NaN	NaN	NaN
127	2019-11-05	NaN	NaN	NaN	NaN
140	2019-11-18	NaN	NaN	NaN	NaN

Imputing Weather Data

```
weather2 = weather.set_index('Date')
#Imputing the missing weather data with interpolate
weather = weather.assign(TMAX=weather.TMAX.interpolate(method='linear'))
weather = weather.assign(TMIN=weather.TMIN.interpolate(method='linear'))
weather = weather.assign(AWND=weather.AWND.interpolate(method='linear'))
weather = weather.assign(PRCP=weather.PRCP.interpolate(method='linear'))
```

Figure 11: Missing value in Weather

3.2 Features Engineering

Below are some of the feature engineering steps performed in the project

3.2.1 Computation of distance from city centre and the closest public transport station

This is computed beforehand using Scala as explained in the Section 3.1.2

3.2.2 Computation of Day of the Week and Availability

The Day of the week is obtained from the feature called as Date, whereas the proportional availability is computed as shown in the Figure 12

Exctration of the Day of the week

```
#calculating days
merged.loc[:,'Date']= pd.to_datetime(merged['Date'])
#merged2.assign(Day = list(merged2['Date'].dt.day_name()))
merged.loc[:,'DoW']= merged['Date'].dt.day_name()
```

Computation of parking availability

```
merged.loc[:,'avail'] = merged['parkingspacecount'] - merged['paidoccupancy']
#Where it is double parking the availability is kept as 0
merged.loc[merged['avail'] < 0, 'avail'] = 0
```

```
merged['avail%'] = merged['avail']*100/merged['parkingspacecount']
```

Figure 12: Feature Engineering

3.3 Transformation

The processed parking data obtained in the above step is then filtered in 15 min interval. Then it is grouped together as shown below based on the factors such as parking, day of the week, hour and minute. The “sideofstreet” column which was missed by the grouping is then added to the dataframe with the help of merging. The “parkinglimit” variable categories are renamed to hourly limits and the unnecessary variables are droped as shown in Figure 13.

```
merged2 = merged[merged['Min'].isin([0,15,30,45])].copy()
```

```
#Grouping data by each parking lot, day of week, hour and minute
merged3 = merged2.groupby(['sourceelementkey','DoW',"Hour","Min"]).agg({'paidoccupancy':['median'],
                                                                        'parkingtimelimitcategory':'median',
                                                                        'parkingspacecount':'median',
                                                                        'NearByBus':'mean',
                                                                        'NearByRail':'mean',
                                                                        'NearByFerry':'mean',
                                                                        'DistCity':'mean',
                                                                        'DistAir':'mean',
                                                                        'TMAX':'mean',
                                                                        'TMIN':'mean',
                                                                        'AWND':'mean',
                                                                        'PRCP':'median',
                                                                                'avail%':'mean'}).reset_index()
```

```
#Data to store only parking segment specific features
park = merged[['sourceelementkey','p_lon', 'p_lat','sideofstreet','parkingtimelimitcategory']].drop_duplicates()
```

```
#Adding the side of street
merged3 = pd.merge(merged3,park[['sourceelementkey','sideofstreet']], on = 'sourceelementkey', how = 'left')
```

```
#Converting parking limit category as str
merged3['parkingtimelimit'] = merged3['parkingtimelimit'].astype(str)
```

```
#Converting Min into a categorical variable
merged3['Min'] = merged3['Min'].astype(str)
```

```
#As per the descriptions from the data scource 120 means 2 hours, 240 means 4 hours and 30 means 10 hours
merged3['parkingtimelimit'] = merged3['parkingtimelimit'].replace(['120'], '2 Hours')
merged3['parkingtimelimit'] = merged3['parkingtimelimit'].replace(['240'], '4 Hours')
merged3['parkingtimelimit'] = merged3['parkingtimelimit'].replace(['30'], '10 Hours')
```

```
#Dropping sourceelemenkey and paidoccupacy as wont be used anymore
merged3 = merged3.drop(columns = ['sourceelementkey','paidoccupancy'])
```

Figure 13: Data Grouping and Transformation

Post this the outliers are filtered from the dataset as shown in Figure 14 below using Z score.

```
#Identifying the outliers using  zscore based on the numeric columns
from scipy import stats
out = merged3.drop(columns = ['DoW','parkingtimelimit','Min','sideofstreet'])
# There are 325 outliers
out[(np.abs(stats.zscore(out)) > 3)].drop_duplicates()
```

	Hour	parkingspacecount	NearByBus	NearByRail	NearByFerry	DistCity	DistAir	TMAX	TMIN	AWND	PRCP	avail%
139	15	4	0.104476	0.236214	0.358709	0.418267	8.662166	219.000000	143.200000	19.600000	0.0	70.000000
354	15	7	0.052970	0.272338	0.389457	0.361852	8.757577	207.000000	124.250000	21.750000	0.0	82.142857
547	15	5	0.134310	0.256748	0.455177	0.365917	8.933273	215.833333	126.833333	27.333333	0.0	76.666667
682	15	5	0.134310	0.256748	0.455177	0.365917	8.933273	134.750000	77.500000	20.250000	0.0	80.000000
1221	15	11	0.123166	0.172370	0.594974	0.427573	9.117515	150.428571	73.214286	22.071429	0.0	82.323232
...	...	...	...	...	...	...	...	...	...	...	...	...
7042	16	10	0.097552	0.505618	1.051080	0.348352	8.956050	142.000000	75.000000	21.000000	29.0	95.000000
7043	16	10	0.097552	0.505618	1.051080	0.348352	8.956050	142.000000	75.000000	21.000000	29.0	100.000000
7046	17	10	0.097552	0.505618	1.051080	0.348352	8.956050	142.000000	75.000000	21.000000	29.0	100.000000
7050	18	10	0.097552	0.505618	1.051080	0.348352	8.956050	142.000000	75.000000	21.000000	29.0	100.000000
7054	19	10	0.097552	0.505618	1.051080	0.348352	8.956050	142.000000	75.000000	21.000000	29.0	100.000000

321 rows × 12 columns

```
#Getting indexes of outliers
index_list =out[(np.abs(stats.zscore(out)) > 3)].drop_duplicates().index.values.tolist()
```

```
#Removing the outliers
merged3 = merged3[~merged3.index.isin(index_list)]
```

Figure 14: Outlier Detection

Final dataset consists of 12209 rows and features explained in the Figure 15

Hour	Hour of the day	**AWND**	Average Wind Speed
Min	0, 15, 30, and 45 minutes	**PRCP**	Precipitation
Day of The Week	Monday to Saturday	**NearByFerry**	Distance from ferry station
Parkingspacecount	Capacity of the parking	**NearByRail**	Distance from rail station
parkinglimit	Time limit of parking 2, 4, and 10 hour	**NearByBus**	Distance from bus station
sideofstreet	Side of the street such as SE, SW,W,NW,NE, and E	**DistCity**	Distance form city centre
TMIN	Minimum Temperature	**DistAir**	Distance from airport
TMAX	Maxium Temperature	**Avail%**	Percentage availability

Figure 15: Final Dataset

3.4 One-Hot Encoding

The Categorical features involved in the processed parking dataset are converted into dummy features as presented in the Figure 16

```
#Creating dummy for categorical data
merged4 =pd.get_dummies(merged3)
```

Figure 16: One-Hot Encoding

3.5 Splitting Data into Train and Test

The independent and dependent variables in the above processed and transformed data are separated into X and Y as presented in the Figure 16. Post this they are divided into Train data of size 75% and Test data of the size 25%.

```
#Separating Independent and Dependent features
X = merged4.drop(columns = 'avail%')
Y = merged4['avail%']
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25, random_state=42)
```

Figure 17: Train and Test Split

3.6 Normalization of the Data

All the features in the combine dataset are in different ranges. Therefore, all of those are transformed in a single range of 0 and 1 using the normalize library as depicted in the Figure 18. The MinMaxScaler[7] is used for the same. The same is implemented on both test and train splits. In addition to, this the target column is transformed into a scale of 0 to 1 as described in the Figure 18

"avail%" dependent variable is normalized in a scale of 0 to 1. i.e 85.25% will become 0.8525

```
#Availability(%) Normalized in a scale of 0 to 1
Y = Y/100
```

```
#Normalizing Train and Test data
from sklearn.preprocessing import MinMaxScaler

# create scaler
scaler = MinMaxScaler()

# fit scaler on data
scaler.fit(X_train)

# apply transform
x_train_normalized = scaler.transform(X_train)
x_test_normalized = scaler.transform(X_test)
```

Figure 18: Normalization

[7] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

4 Model Application

The processed data from the Section 3.6 is then used for the prediction of availability. Sections below show the models used for the same. The predictability of the models is optimized using GridSearchCV[8].

4.1 Random Forest(RF)

Here, the RandomForestRegressor[9] library is utilized. The Figure 19 shows the RF application with the base settings.

```
from sklearn.ensemble import RandomForestRegressor
rfDefault = RandomForestRegressor()
rfDefault.fit(x_train_normalized, y_train)
```

```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=100, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False)
```

- R square on the Test dataset

```
y_pred_test_rf = rfDefault.predict(x_test_normalized)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_test_rf)
```

```
0.9728217606357882
```

- RMSE on the Test dataset

```
from sklearn.metrics import mean_squared_error
from math import sqrt

rmse = sqrt(mean_squared_error(y_test, y_pred_test_rf))
print(rmse)
```

```
0.041577446190034506
```

- MAE on the Test dataset

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, y_pred_test_rf)
```

```
0.028842845929421664
```

Figure 19: RF Base Configuration

[8]https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html?highlight=gridsearch#sklearn.model_selection.GridSearchCV

[9]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

The Figure 20 shows the RF application with the optimal settings.

```
from sklearn.model_selection import GridSearchCV

param_list = {'max_features' : ['auto'],'n_estimators': [500,1000], 'min_samples_leaf' : np.arange(1,6,2),
              'min_samples_split': np.arange(2,10,2)}
rf10fold = GridSearchCV(estimator=RandomForestRegressor(), param_grid=param_list,n_jobs = -1,cv = 10)



rf10fold.fit(x_train_normalized, y_train)
```

- R Square on the Test dataset

```
y_pred_test_rf10fold = rf10fold.predict(x_test_normalized)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_test_rf10fold)
```

```
0.9730544545756287
```

- RMSE on the Test dataset

```
rmse = sqrt(mean_squared_error(y_test, y_pred_test_rf10fold))
print(rmse)
```

```
0.04139907523000112
```

- MAE on the Test dataset

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, y_pred_test_rf10fold)
```

```
0.028747803984561456
```

Figure 20: Optimized RF

The Figure 21 shows the Feature importance of the best RF.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


plt.rcdefaults()
fig, ax = plt.subplots()

# Example data
imp = pd.DataFrame(list(zip(x_train_normalized.columns,rf10fold.best_estimator_.feature_importances_ )),
                   columns = ['Feature', 'Importance'])
imp = imp.sort_values(by=['Importance'],ascending=False)
people = imp['Feature']
y_pos = np.arange(len(people))
performance = imp['Importance']

ax.barh(y_pos, performance, align='center')
ax.set_yticks(y_pos)
ax.set_yticklabels(people)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('Performance')
ax.set_title('Feature Importance')

plt.show()
```

Figure 21: RF Feature Importance

4.2 XGBoost

Here, The XGBoost[10] library is utilized. The Figure 22 the XGBoost application with the base settings.

```
import xgboost as xgb
xg = xgb.XGBRegressor()

xg.fit(x_train_normalized,y_train)
```

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.300000012, max_delta_step=0, max_depth=6,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=100, n_jobs=0, num_parallel_tree=1,
             objective='reg:squarederror', random_state=0, reg_alpha=0,
             reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method='exact',
             validate_parameters=1, verbosity=None)
```

- R Square on the Test dataset

```
y_pred_test_xg = xg.predict(x_test_normalized)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_test_xg)
```

```
0.970425028846323
```

- RMSE on the Test dataset

```
rmse = sqrt(mean_squared_error(y_test, y_pred_test_xg))
print(rmse)
```

```
0.0433719866468381
```

- R Square on the Test dataset

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, y_pred_test_xg)
```

```
0.031041751170757158
```

Figure 22: XGBoost Base Configuration

[10] https://xgboost.readthedocs.io/en/latest/parameter.html

The Figure 23 the XGBoost application with the optimal settings.

```
from sklearn.model_selection import GridSearchCV

param_list = {'max_depth' : np.arange(8,16,2),'n_estimators': [500,1000], 'learning_rate_init' : [0.01,0.02,0.03,0.04,0.05],
              'colsample_bytree' : [0.3,0.4,0.5,0.6]}
xg10fold = GridSearchCV(estimator=xgb.XGBRegressor(), param_grid=param_list,n_jobs = -1,cv = 10)



xg10fold.fit(x_train_normalized, y_train)
```

- R Square on the Test dataset

```
y_pred_test_xg10fold = xg10fold.predict(x_test_normalized)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_test_xg10fold)
```

```
0.9765964473297708
```

- RMSE on the Test dataset

```
rmse = sqrt(mean_squared_error(y_test, y_pred_test_xg10fold))
print(rmse)
```

```
0.03858229388772691
```

- MAE on the Test dataset

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, y_pred_test_xg10fold)
```

```
0.027220761554566597
```

Figure 23: Optimized XGBoost

The Figure 24 shows the feature importance of best XGBoost

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


plt.rcdefaults()
fig, ax = plt.subplots()

# Example data
imp = pd.DataFrame(list(zip(x_train_normalized.columns,xg10fold.best_estimator_.feature_importances_ )),
                   columns = ['Feature', 'Importance'])
imp = imp.sort_values(by=['Importance'],ascending=False)
people = imp['Feature']
y_pos = np.arange(len(people))
performance = imp['Importance']

ax.barh(y_pos, performance, align='center')
ax.set_yticks(y_pos)
ax.set_yticklabels(people)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('Performance')
ax.set_title('Feature Importance')

plt.show()
```

Figure 24: XGBoost Feature Importance

4.3 Back Propagation Neural Network(BPNN)

Here, The MLPRegressor[11] library is utilized which provides a Back Propagation Neural Network . The Figure below 25 the BPNN application with the base settings.

```
m = MLPRegressor()
m.fit(x_train_normalized, y_train)
```

```
MLPRegressor(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
             beta_2=0.999, early_stopping=False, epsilon=1e-08,
             hidden_layer_sizes=(100,), learning_rate='constant',
             learning_rate_init=0.001, max_fun=15000, max_iter=200,
             momentum=0.9, n_iter_no_change=10, nesterovs_momentum=True,
             power_t=0.5, random_state=None, shuffle=True, solver='adam',
             tol=0.0001, validation_fraction=0.1, verbose=False,
             warm_start=False)
```

- R Square on the test dataset

```
y_pred_test_bpnn = m.predict(x_test_normalized)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_test_bpnn)
```

```
0.8640668034370425
```

- RMSE on the test dataset

```
from sklearn.metrics import mean_squared_error
from math import sqrt

rmse = sqrt(mean_squared_error(y_test, y_pred_test_bpnn))
print(rmse)
```

```
0.0929843619687437
```

- MAE on the test dataset

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, y_pred_test_bpnn)
```

```
0.0707848295518675
```

Figure 25: BPNN Base Configuration

[11] https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

The Figure 26 the BPNN application with the optimal settings.

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.model_selection import GridSearchCV
from matplotlib import pyplot
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error
import pandas as pd


#mlpr = MLPRegressor()

param_list = {'random_state' : [0,1,2,3,4,5,6,7,8,9,10],'activation' : ["relu"],'solver' :['adam'],
              'max_iter': [1000],'hidden_layer_sizes':[(100,90),(100,100),(100,110),(100,120)],
              'learning_rate':['adaptive'],'learning_rate_init' : [0.01,0.02,0.03,0.04,0.05]}
m10fold = GridSearchCV(estimator=MLPRegressor(), param_grid=param_list,n_jobs=-1, cv = 10)



m10fold.fit(x_train_normalized, y_train)
```

- R Sqaure on the Test dataset

```
y_pred_test_bpnn10fold = m10fold.predict(x_test_normalized)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_test_bpnn10fold)
```

```
0.9450081583948783
```

- RMSE on the Test dataset

```
from sklearn.metrics import mean_squared_error
from math import sqrt

rmse = sqrt(mean_squared_error(y_test, y_pred_test_bpnn10fold))
print(rmse)
```

```
0.05914203752330328
```

- MAE on the Test dataset

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, y_pred_test_bpnn10fold)
```

```
0.043353234814817315
```

Figure 26: Optimized BPNN

The Figure 27 shows the feature importance of best BPNN

```python
import eli5
from eli5.sklearn import PermutationImportance
from IPython.display import display
perm = PermutationImportance(m10fold).fit(x_test_normalized, y_pred_test_bpnn10fold)
importance = eli5.formatters.as_dataframe.explain_weights_df(perm,feature_names = x_test_normalized.columns.tolist())
```

```python
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


plt.rcdefaults()
fig, ax = plt.subplots()

# Example data
people = importance['feature']
y_pos = np.arange(len(people))
performance = importance['weight']

ax.barh(y_pos, performance, align='center')
ax.set_yticks(y_pos)
ax.set_yticklabels(people)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('Weights')
ax.set_title('Feature Importance')

plt.show()
```

Figure 27: BPNN Feature Importance

References

Winarno, E., Hadikurniawati, W. and Rosso, R. N. (2017). Location based service for presence system using haversine method, pp. 1–4.