# FIT5196 Task 1 in Assessment 3

**Student Name: Vishal Pattabiraman**

**Student ID: 31131441**

Date: 18/11/2020

Version: 1.1

Environment: Python 3.8.5 and Jupyter notebook

---

## Library Import

In [ ]:
```
#Basic scientific python libs
import pandas as pd # data frame etc.
import numpy as np # lin solve
import re # extract info from shopping cart using regex
import math # for Math calculations.
import datetime # extracting hours and minutes from time.
import geopandas as gpd # reading shape file and creting geopandas datafr
from shapely.geometry import Point, Polygon # to use spatial join and rea
pd.options.mode.chained_assignment = None # to remove "chained" assignmen
```

## 1. Load Dataset

In this section we will need to load data from different type of dataset provided to us. This includes reading files from :

- Excel
- XML
- HTML
- CSV
- PDF

The following sections shows how data from these different file types are read into python and loaded using pandas and other library to store them in a pandas data frame.

### 1.1 Import Hospital excel file

Hospital Excel file contains the information related to hospitals with their corresponding unique Hospital ID with corresponding Lattitude and Longitude of these hospitals. Using Pandas we will read the Excel file using **read_excel** function and store them onto a pandas dataframe

**hospital_data**.

In [ ]:
```
# Read Excel file.
```

Using **Head()** function we can check the top 6 rows of the dataframe that is created using read_excel function.

In [ ]:
```
# Display columns and values
```

In order to perform any manipulation to this file, we need to understand the datatype of all the columns so that if any changes or conversion is required or needs to be performed. **info()** function helps to display datatypes of all the dataframe columns with number of entries which in this case is 199 rows with 5 rows.

In [ ]:
```
# Check for datatypes
```

Describe() function comes handy to handle number type type data. This has builtin operation that shows the min, max, mean , Standard deviation, 25% , 50% and 75% quartiles of distribution of these columns. We will use this to see if there are any outliers. Clearly, the data here doesnt show any extereme values and standard deviation is between 0.63 and 0.84 for both lat and lng columns respectively.

In [ ]:
```
# Check the excel_data count and stat information.
```

**Preprocess Hospital data.**

Although describe() function does provide information of the spread of all the columns, we are still not sure if these rows have

1. Unnamed column is not required and needs to be pruned.
2. Duplicated rows
3. Duplicated Lat and Lng
4. Check if Lattitude and Longitude provide for hospitals are within Victoria boundary.

In [ ]:
```
# 1. we will drop the duplicated index. Unnamed: 0 column
```

Empty search result shows that there are no duplicated rows.

In [ ]:
```
# 2. Check for duplicates in excel_data dataset.
```

Exmpty search result shows that both Lat and Lng columns are not duplicated.

In [ ]:
```
# 3. Check for lat and long duplicates.
```

Using the information from Wiki (https://en.wikipedia.org/wiki/Module:Location_map/data/Australia_Victoria) we know that Victoria state has the following boundaries **-33.8 top , -39.3**

**bottom ,140.6 left and 150.3 right**. In order to make sure that Hospital_data dataframe does not have hospital information that do not belong to **Victoria state** we will check this information with the lat and lng columns of Hospital_data. Empty search result shows that all the hospital information provided belong to Victoria state.

In [ ]:
```
# 4. Check for invalid Hospital data Using Victoria Lat, Long
# According to Wiki (-33.8 top, -39.3 bottom,140.6 left and 150.3 right)

index= (hospital_data.lat<-39.3) | (hospital_data.lat>-33.8) | (hospital_
```

## 1.2 Import real_state xml file.

In this section we need to import information from the XML file real_state. This file contains data related to Real state property in Victoria state. Since this file is in XML format we will read the file using inbuilt file open and readline() function real_state data is read initially. Once we have this information, XML is stored in a particular format which will be exploited using Regex.

In [ ]:
```
# Read XML data
with open ("real_state.xml",encoding="UTF-8") as fd:
    xml_file=fd.readline() # read xml file
```

We can see that xml_file contain data in tags within **<>**. The columns, **'property_id','lat','lng','addr_street','price','property_type','year','bedrooms','bathrooms','park** are included into for instance, which denotes the start of this column and **< //property_id>**denotes the end of this column.

In [ ]:
```
# Display the file.
```

In this section we will set up the overall frame what the real_state dataframe should look like, All the columns that are available in XML is used to create this dataframe.

In [ ]:
```
# Dataframe to store xml data.
real_state_xml=pd.DataFrame(columns=['property_id','lat','lng','addr_stre
```

It is important to note that based on understand of data we found that each **property** in the XML is unique and a single row has 'property_id','lat','lng','addr_street','price','property_type','year','bedrooms','bathrooms','parking_spac with this unique property ID value. Knowing this information we will extract all the Property Id separately using the regex shown below. This regex helps to anything that is within start tag to <//property_id> end tag. Once we have this information, we could see that for instance **<**n55964 type="int"**>**56499**</**n55964**>** has the property id number embedded inside the end tag </n**55964**> highlighted. Again using Regular expression </n(\d+)> we extract all digits that are 1 or more into id_raw list.

In [ ]:
```
# obtain row Id from XML id.
id_dump=re.findall(r'<property_id type="dict">(.*?)</property_id>',xml_fi
# Convert to string.
id_dump=str(id_dump)
```

```
# store row ids.
```

id_raw shows all the extracted proeprty id that will be used to fetch information of all the other columns

In [ ]:
```
# Ids of Rows
```

Image (https://regexper.com/#%3Cn%7Bitem%7D%20type%3D%22%28%3F%3Aint%7Cstr%7Cfloat%29%22%3E%28.*%3F%29%3C%5C%2Fn%7Bitem%7D%3E) shows how this regualar expression works in a detailed manner. The essence is that we check for all items with property id in the format inside tag with type= int or str or float. We extract the value using (.*?) and store this information into the pandas dataframe. Since we know for a single property there are values for 'property_id','lat','lng','addr_street','price','property_type','year','bedrooms','bathrooms','parking_spac using the loop we extract information for each column one by one and store into corresponing pandas columns.

In [ ]:
```
# Increment the index and append rows to dataframe.
i=1 # index value
for item in id_raw:
    # search through xml_file and get corresponding row data.
    trans_value=re.findall(rf'<n{item} type="(?:int|str|float)">(.*?)</n{
    # append data to dataframe.
    real_state_xml.loc[i]=trans_value
```

once column information is extracted from the XML we display the pandas data frame which contains information of real_state.

In [ ]:
```
# display xml_data
```

**Preprocess real_state_xml data**

Although we have extracted the XML data into pandas dataframe there are chances that we might have erros and issues with the rows and columns of this dataframe. We will check for:

1. Duplicated Property ID
2. Duplicated Lattitude and Longitude.
3. Check if Lattitude and Longitude provide for Properties are within Victoria boundary.

Empty search result shows that there no duplicated Property ID in xml_data

In [ ]:
```
# 1. Check for duplicate value in xml_data
# ~ is for False.
```

Empty search result shows that there no duplicated lattitude or longitude in xml_data

In [ ]:
```
# 2. Duplicated lat and long
real_state_xml.loc[real_state_xml['lat'].duplicated() & real_state_xml['l
```

As explained for the Hospital_data, we use the latitude information provide in Wikipedia for Victoria state and check if there are any properties that are outside Victoria state. Empty search result shows that there are not properties that are outside Victoria state.

In [ ]:
```
# 3. Check for invalid real_state_xml data Using Victoria Lat, Long
# According to Wiki (-33.8 top, -39.3 bottom,140.6 left and 150.3 right)
index= (real_state_xml.lat.astype(float)<-39.3) | (real_state_xml.lat.ast
real_state_xml.loc[index]
```

## 1.3 Import real_state Json file

In this section we will import the information from JSON file. Here we will use pandas read_json() function, which can read json file and provide a pandas dataframe. Here we store the pandas dataframe as real_state_json and display the column values as shown below. Also it is important to note that we need one amalgamated dataframe that contains information about all real_state_properties from XML and JSON file.

In [ ]:
```
# read json file and store as dataframe.
real_state_json=pd.read_json('real_state.json')
# display json dataframe.
```

### Preprocess real_state_json data

Although we have extracted the JSON data into pandas dataframe there are chances that we might have erros and issues with the rows and columns of this dataframe. We will check for:

1. Duplicated property_ID
2. Duplicated Latitude and Longitude
3. Check if Lattitude and Longitude provide for Properties are within Victoria boundary.

We can see that there are 7 duplicated rows with same property ID which is has the same information repeated twice. This means that it doesnt matter if we remove the first row or the second row we do not lose any information. In this case duplicates are dropped while keeping the first entry of that property id. Using duplicated() function we can check for rows which have the same property id.

In [ ]:
```
# 1. check for duplicated propertyID.
```

The below code chunk shows a sample property id 69903 which is duplicated with same value accross all rows.

In [ ]:
```
# 1. Display one duplicated Property.
```

In this section we will drop the second occurence of the same property id entry using **drop_duplicates()** funtions. In this function we will specify the parameters keep='first' to denote which record needs to be dropped.

In [ ]:
```
# 1. remove duplicated records and keep first one.
```

Empty search result shows that there are no duplicated lattitude and longitude, in other words no two properties have the same lattitude and longitude.

In [ ]:
```
# 2. duplicated lat and long
```

Empty search result shows that there are no properties that are outside **Victoria state**. We need to make sure about this information as data manipulations that will be done below corresponds to properties that are inside Victoria state.

In [ ]:
```
# 3. Check for invalid real_state_json data Using Victoria Lat, Long
# According to Wiki (-33.8 top, -39.3 bottom,140.6 left and 150.3 right)
index= (real_state_json.lat.astype(float)<-39.3) | (real_state_json.lat.a
real_state_json.loc[index]
```

## 1.4 Merge realstate data to one dataframe

We need to have one amalgamated dataframe that contains data from both the XML and JSON files. So in this section we will merge these two files into one pandas dataframe **real_state_data**. In order accomplish this we will use concat() function that is contatinates information from real_state_xml and real_state_json into one single pandas container.

In [ ]:
```
# Store real_state data from json and xml to one dataframe
```

In [ ]:
```
# convert lat and long to float as to make comparision.
real_state_data['lat']=real_state_data['lat'].astype(float)
```

In [ ]:
```
# display realstate data.
```

**Preprocess real_state_data information.**

Although we have performed preprocessing to both real_state_xml and real_state_json dataframes separately, we are still not sure if there are issues and anamolies that may have occured in the contatinated dataframe. We will check for :

1. Duplicated PropertyID
2. Duplicated Latitude, Longitude and Price.

Here we dont look for duplicated lattitude and longitude it is possible that there might be different units within the same property that are being sold. At this point we need to check only for duplicated property ids.

Empty values shows that there are no duplicated properties in the real_state_data information.

In [ ]:
```
# 1. check if duplicates in json and xml combined.
```

In [ ]:
```
# 2. check if a property with same lat and long is sold for same price.
# This is so that we dont sell the same property with different property
```

## 1.5 Import supermarkets HTML file.

In this section we will read the information from HTML file using the pandas in built function read_html that extracts information from supermarkets.html file and returns a list of column information. Since we need a pandas dataframe we will use the index 0 to store it into supermarket_data data frame.

In [ ]:
```
# Read Html data.
html_file=pd.read_html('supermarkets.html')
# html_file is a list so taking first element which contains columns infr
supermarket_data=html_file[0]
# display supermarket_data
```

**Preprocess supermarket_data dataframe.**

The information extract from HTML file may contains errors and other issues which needs to be identified and fixed before this dataframe can be used for manipulations. We will check for the following anamolies:

1. Remove Unnamed column.
2. Check for duplicated id.
3. Duplicated Lattitude and Longitude.
4. check if supermarkets are inside Victoria state.

The below code will drop the Unnamed column which is the default index that is not required.

In [ ]:
```
# 1. we will drop the duplicated index. Unnamed: 0 column
```

Empty search result shows that there are no duplicated Supermarket with same ID

In [ ]:
```
# 2. check for duplicates
```

We can see that there are two super markets with ids S_226 and S_227 which have the same lattitude and longitude that needs to be removed as this is a duplicated Coles market with same Lattitude and Longitude but different Id.

In [ ]:
```
# 3. check for lat and long duplicated record.
```

In [ ]:
```
# 3. Drop lat and lng duplicated.
```

The below code shows SuperMarkets that **do not belong Victoria state**. We need to clean these columns and remove them before performing any manupulations further on.

In [ ]:
```
# 4. Check for invalid supermarket_data data Using Victoria Lat, Long
# According to Wiki (-33.8 top, -39.3 bottom,140.6 left and 150.3 right)
index= (supermarket_data.lat.astype(float)<-39.3) | (supermarket_data.lat
# Display Supermarkets that are not inside Victories State.
supermarket_data.loc[index]
```

We can see there are 11 rows with supermarkets that are not inside Victoria state. These will be removed using the '~' which denote the negation. The code below identifies rows that do not satify the index condition shown above and the results are stored back onto **supermarket_data** dataframe.

In [ ]:
```
# 4. Remove Non Victorian supermarkets.
supermarket_data=supermarket_data.loc[~index]
# Display supermarkets that are inside Victoria state.
```

## 1.6 Import Shoppingcenter PDF file

In this section we will read the PDF file using tabula-py library. Since this is not readily available we need to use **pip install tabula-py** command install this package before importing the libraries. from tabula import read_pdf command we can import the tabula library. Using the read_pdf() function we read shopingcenters.pdf file and specify parameter pages to denote from which pdf pages the data needs to be imported. This function returns a list of dataframe with columns which will then be concatinated into one pandas dataframe using concat() function shown below. We specify the parameter ignore_index=True so that there wont be any issues with the index that are already populated in dataframe. It is important to note that each dataframe has index from 0 to the length of items in that page stored inside the list including the ignore index helps to take the data as a whole.

In [ ]:
```
# Use the below command to install tabula-py library that will be used to
# pip install tabula-py
from tabula import read_pdf
```

In [ ]:
```
shopping_data= pd.concat([df[0],df[1],df[2]],ignore_index=True)
```

### Preprocess shopping_data

We are still not sure if there are any anamolies in the PDF shopping centers information, in order to remove them we will check for the following:

1. Remove Unnamed Column
2. Check for Duplicated Shopping id
3. Check for Duplicated Lattitude and Longitude
4. Check if a a Shopping center is inside Victoria State.

Empty search result shows that there are no duplicated Shopping center data with same SC_id.

In [ ]:
```
# 1. We will drop the duplicated index. Unnamed: 0 column
```

```
shopping_data.drop(['Unnamed: 0'], axis=1, inplace=True)
# 1. rename the column sc_id to id to use in distance calculation.
shopping_data.rename(columns = {'sc_id':'id'}, inplace = True)

# 2. check for duplicates
```

Empty search result shows that there are no duplicated Lattitude and Longitude for Shopping center.

In [ ]:
```
# 3. duplicated lat and long
```

In this section we will check for shopping center that are **inside the Victoria state**. It is important we use data of Victoria state and not outside as the properties belong to Victoria state. We will use the information provided in Wikipedia for the boundary of Victoria state. The serach result below shows shopping centers that are **outside Victoria state**.

In [ ]:
```
# 4. Check for invalid shopping_data data Using Victoria Lat, Long
# According to Wiki (-33.8 top, -39.3 bottom,140.6 left and 150.3 right)
index= (shopping_data.lat.astype(float)<-39.3) | (shopping_data.lat.astyp
# Display shopping_data that are not inside Victories State.
shopping_data.loc[index]
```

In the code below we will use the '~' symbol that helps to identify negation of non victorian shopping centers. We search for these shopping centers and assign them to the required Shopping_data dataframe.

In [ ]:
```
# Remove non Victoria Shopping centers.
```

## 1.7 Import stops text file

In this section we will read the Text file which is in the form of csv, or comma separated as delimiter. Using the pandas in build function read_csv() stops.txt file is read and stored into a pandas dataframe train_data. THe column names are renamed to id, lat and lng to match the data that is available in other files. This is done to make the manipulations easier. This file contains all the stops that are available in the GTFS Victoria route. Every location has a unique stop_id and their corresponding long and short name providing information about the stop.

In [ ]:
```
# Read stops text file.
```

In [ ]:
```
# rename columns to make the manipulations easier.
```

### Preprocess train_data

In this section we will check for any anamolies or issues that might have been there while reading the data from txt format. We will check for the following issues:

1. Duplicated Lattitude and Longitude.
2. Duplicated Stop_id.

Empty search result shows that there are no duplicated lattitude or longitude in the train_data.

In [ ]:
```
# 1. duplicated lat and long
```

Empty search result shows that there are no duplicated stops in the train_data

In [ ]:
```
#2.  check for duplicated stop id.
```

# 2. Calculating distance from Real state.

In this section we will be calculating the distance between property and Super Market, Shopping center, closest station and hospital. This is done to identify the closest Super Market, Shopping center, closest station and hospital that are avilable to a property. This is acoomplied using two functions dis_sphere and calculate_distance that calculates the distances.

## 2.1 Functions used to calculate distance from two points.

dis_sphere() function takes two sets of lattitude and longitude and calculates the distance between those two spatial points. We use the earth distance as 6378 which is provided in the specification document. We convert a lattitude, longitude point into degrees initially and then using the Harvesine distance calculation formula to identify the distance between these two points.

In [ ]:
```
# Function that will be used for calc distance from two points.
def dis_sphere(lat1, long1, lat2, long2):
    # Converts lat & long to spherical coordinates in radians.
    deg_rad = math.pi/180.0

    # p = 90 - latitude
    p1 = (90.0 - lat1)*deg_rad
    p2 = (90.0 - lat2)*deg_rad

    # theta = longitude
    t1 = long1*deg_rad
    t2 = long2*deg_rad

    # Compute the spherical distance from spherical coordinates.
    cos = (math.sin(p1)*math.sin(p2)*math.cos(t1 - t2) +
       math.cos(p1)*math.cos(p2))
    arc = math.acos(cos)*6378 #radius of the earth in km
```

calculate_distance() function is the interface between dis_sphere() function that helps sent the spatial points(lattitude,longitude) point. We use the codes to help the function identify the type of distance it has to calcualte. the corresponding codes and their actions are shown below:

val = 1 hospital_data
val = 2 shopping_center
val = 4 train station

In [ ]:
```
# function to calculate the minimum distance and get id for dataframe.
def calculate_distance(lat_real,long_real,val):

    ### Function operations ####
    ##########################
    # val = 1 hospital_data   #
    # val = 2 shopping_center #
    # val = 3 supermarket     #
    # val = 4 train station   #
    ##########################

    # set corresponding code dataframes.
    if val is 1:
        df=hospital_data
    elif val is 2:
        df= shopping_data
    elif val is 3:
        df= supermarket_data
    elif val is 4:
        df= train_data

    for i, irow in df.iterrows():
        df.loc[i,'distance']= dis_sphere(irow['lat'],irow['lng'],lat_real

    # Calculate the minumum distance and id
    min_df=df.loc[(df.distance==min(df.distance)),['id','distance']]
```

## 2.2 Calculate distance to closest Hospital from property.

As denoted in the previous section we will use the val value as 1 to inform calculate_distance() function to return the closest hospital to the property lattitude and longitude. we set this information in a temporary column in the real_state_data pandas dataframe and then extract the required distance and hospital id as shown in the below sections.

In [ ]:
```
# Extract Hospital ID and distance.
real_state_data['hospital_temp']=real_state_data.apply(lambda x: calculat
```

In [ ]:
```
# Store to corresponding columns.
real_state_data['Hospital_id']= real_state_data['hospital_temp'].apply(la
```

## 2.3 Calculate distance to closest shopping center from property.

As denoted in the previous section we will use the val value as 2 to inform calculate_distance() function to return the closest shopping center to the property lattitude and longitude. we set this information in a temporary column in the real_state_data pandas dataframe and then extract the

required distance and shopping center id as shown in the below sections.

In [ ]:
```
# Extract Shopping center ID and distance.
real_state_data['shopcenter_temp']=real_state_data.apply(lambda x: calcul
```

In [ ]:
```
# Store to corresponding columns.
real_state_data['Shopping_center_id']= real_state_data['shopcenter_temp']
```

### 2.4 Calculate distance to closest Super Market from property.

As denoted in the previous section we will use the val value as 3 to inform calculate_distance() function to return the closest Super Market to the property lattitude and longitude. we set this information in a temporary column in the real_state_data pandas dataframe and then extract the required distance and Super Market id as shown in the below sections.

In [ ]:
```
# Extract Shopping center ID and distance.
real_state_data['supermarket_temp']=real_state_data.apply(lambda x: calcu
```

In [ ]:
```
# Store to corresponding columns.
real_state_data['Supermarket_id']= real_state_data['supermarket_temp'].ap
```

In [ ]:
```

```

### 2.5 Calculate distance to closest Train stop from property.

As denoted in the previous section we will use the val value as 4 to inform calculate_distance() function to return the closest Train stop to the property lattitude and longitude. we set this information in a temporary column in the real_state_data pandas dataframe and then extract the required distance and Train stop id as shown in the below sections.

In [ ]:
```
# Extract Shopping center ID and distance.
real_state_data['train_temp']=real_state_data.apply(lambda x: calculate_d
```

In [ ]:
```
# Store to corresponding columns.
real_state_data['Train_station_id']= real_state_data['train_temp'].apply(
```

In [ ]:
```
# remove temporary columns which are no longer needed.
```

In [ ]:
```
# reset index since this will be a problem.
real_state_data=real_state_data.reset_index(drop=True)
```

## 3. Identify if there is direct train to Flinders. (Transfer_flag)

In order to calculate the Transfer flag we will first filter out all data from trip.txt to match service id equal to **T0**. **T0** is the trip that runs through all the weekdays .i.e. from Monday to Friday on all days. So we Filter trips that run on all the weekdats. Then we also match the head sign

(trip_headsign) to **'City (Flinders Street)'** as we need all trips that head to Flinders street on all weekdays.

## 3.1 Load Trip data and validate Weekday trips.

As denoted in the previous section we will read the trips.txt file and extract data that match service_id to T0 and trip_headsign to City (Flinders Street). This information is stored into a pandas dataframe **weekday_trips**.

In [ ]:
```
# read trips text file.
trip_data = pd.read_csv('trips.txt')
```

In [ ]:
```
# get all weeks trip data. trips that head to Flinders Street.
# & (trip_data['direction_id'] == 0)
weekday_trips=trip_data.loc[(trip_data['service_id'] == "T0") & (trip_dat
```

## 3.2 Load Stop_times and merge all weektrips.

In this section we will merge the information of Flinders trips that run on all days with the stop_times.txt data which contains actual trip information with arrival time and departure times. We match the trip_id previously extracted with the stop_times data, this is done as trip_id are unique and correspond to a trip from one origin to a destination. We can use the **stop_sequence** columns to check the sequence in which that particular trip starts and how many stops are there in that trip and the stop_id that are there in that trip.

#### Load Stop times file.

In [ ]:
```
# read the stop times file.
```

In [ ]:
```
# display columns.
```

#### Merge weekday data with stoptimes based on trip id

In [ ]:
```
# merge trips and stop times
trip_with_stops=df = pd.merge(weekday_trips[['trip_id']]
                 ,stop_times[['trip_id','stop_id','arrival_time','departu
```

In [ ]:
```
# display merge data.
```

#### Extract rows where data is between 7.00AM to 9.00AM.

In this section we extract the trips that are from 7.00 AM to 9.00 AM along with the leading condition, trips going to Flinders on all weekdays. It is important to note that if a trip started at 9.00AM at a particular stop that there are chances that we might not know the end of that trip if we sorted the data using departure time. So we will use a **lead time** of 3 hours to capture all the complete trips that might be missed if we extracted trips within 7.00AM to 9.00AM on departure time. We will extract the exact 7.00AM to 9.00AM trip on a particular stop in the following sections.

Also we will use regular expression to extract information from departure information.

Image (https://regexper.com/#07%3A00%3A00%7C08%3A%5Cd%5Cd%3A%5Cd%5Cd%7C09%3A%5Cd%5Cd%3A%5Cd%5Cd%7C10%3A%5Cd%5Cd%3A%5Cd%5Cd%7C11%3A%5Cd%5Cd%3A%5Cd%5Cd%7C12%3A00%3A00) shows how the regular expression works, typically it searches for all trips that follow the time format of 07:00:00 to 12:00:00 and any time within this frame given by the 08:\d\d:\d\d|09:\d\d:\d\d|10:\d\d:\d\d|11:\d\d:\d\d expression.

In [ ]:
```
# extract index of trips between 7:00AM to 12:00PM
```

In [ ]:
```
# subset the required trips.
trip_stop_time=trip_with_stops.loc[index]
# Final Subset dataframe with trips, stops and time.
```

**Extract stations that lead to Filders stop**

In this section we extract the trips that lead to Flinders stop or contain the Flinders stop ID 19854 this information can be extracted from trips.txt file or train_data pandas dataframe. Using this we merge the information from trip_stop_time calculated in the previous section with Flinder_trips and extract trips that lead to Flinders station.

In [ ]:
```
# Extract Filders station Stop_ID
Filder_stop_id= int(train_data.loc[(train_data.stop_name=='Flinders Stree
```

In [ ]:
```
# Filders station trips with stop_id 19854.
Flinder_trips= trip_stop_time.loc[trip_stop_time.stop_id==Filder_stop_id]
```

As discussed above this section extracts trips that lead to flinders station and stores the time at which a trip lands on Flinders station. This information will be handy when calculating average travel time which will be done in the following section.

In [ ]:
```
# obtain data with Flinders information.
flinders_final_data=pd.merge(trip_stop_time[['trip_id','stop_id','arrival
         Flinder_trips[['trip_id','stop_id','departure_time']]
        ,on=['trip_id'])
```

**Prune out trips that start before 9.00 for a given station.**

Now that we have all the trips that lead to Flinders station on all weekdays we need to identify the correct trips that start before 9.00AM based on stops. This we are able to exactly extract the correct trips that lead to Flinders on all weekdays and are between 7-9AM. Using date.time function we identify trips that are before 9.00 am and set a flag value Valid true or false to it, which then can be used to prune the data further. Here it is important to note that there are trips that go beyond Flinders which are still in this dataframe. We need to remove them as well this is handled in the else section where those invalid portion of the trip will have the valid flag set to False.

In [ ]:
```
# obtain valid trips below 9AM
for i,irow in flinders_final_data.iterrows():
    temp_hour= int(datetime.datetime.strptime(irow['departure_time_x'],'%
```

```
    temp_min= int(datetime.datetime.strptime(irow['departure_time_x'],'%H
    if temp_hour<9:
        flinders_final_data.loc[i,'valid']=True
    elif temp_hour==9 and temp_min==0:
        flinders_final_data.loc[i,'valid']=True
    else:
```

In [ ]:
```
# update final data with correct subset
flinders_final_data=flinders_final_data.loc[flinders_final_data.valid==Tr
```

**Set the Transfler Flag to Real_state data**

Now **flinders_final_data** has the correct information of all direct trips to Flinders so we can use the unique() function to extract all the stop id that are in these trips and check them against the real_state_data to set the Transfer_flag to 0 if they match transfer flag station 'tf_stations'. If the stations do not match then 1 is set to those properties.

In [ ]:
```
# transfer flag stations.
tf_stations=flinders_final_data.stop_id_x.unique()
# Create TransferFlag column and calculate the value.
```

# 4. Calculate Average travel time to CBD

From the previous section we have flinders_final_data pandas dataframe that contains the correct trips that lead to Flinders. As all rows in this datafram contains the departure_time_y column that contains the time at which that trip goes to Flinders we will use the **groupby()** function based on stop_id_x and get the grouped object. On this grouped object we can apply mean() function that will fetch the average time it takes from a stop to Flinders station. These steps are done in the following sections.

In [ ]:
```
# Sample output for a station id 19847
```

In this section we extract the hour and minute information from departure_time_x column using strptime() function that is available in datetime library and calculate the time taken from a stop to Flinders station.

In [ ]:
```
# calculate the minutes of columns.
for j,jrow in flinders_final_data.iterrows():

    h1=int(datetime.datetime.strptime(jrow['departure_time_x'],'%H:%M:%S'
    m1=int(datetime.datetime.strptime(jrow['departure_time_x'],'%H:%M:%S'
    h2=int(datetime.datetime.strptime(jrow['departure_time_y'],'%H:%M:%S'
    m2=int(datetime.datetime.strptime(jrow['departure_time_y'],'%H:%M:%S'
    t1 = timedelta(hours=h1, minutes=m1)
    t2 = timedelta(hours=h2, minutes=m2)
    # calculate the differences
    diff=timedelta(hours=0, minutes=0)
    # Check if hours from stopx is less than Flinders stopy time.
    if h1<h2:
        diff=t2 - t1 # calculate time difference.
```

```
    # If hours match and minutes of stopx less than Flinders stopy time.
    elif h1==h2 and m1<m2:
        diff=t2 - t1 # Calculate the time difference.
    else:
        diff=timedelta(hours=0, minutes=0)# if Hours and minutes are grea

    # assign the minutes to trips.
    flinders_final_data.loc[j,'traveltime']= diff.seconds
```

In this section we check the minutes that is available to each station and check for outliers in the minutes distribution.

In [ ]:
```
# Check for outliers.
```

Using the groupby() function previously discussed calcualte the average time for a station.

In [ ]:
```
# calculate average time using groupby
average_time_data=flinders_final_data.groupby(by='stop_id_x').mean('trave
```

Extract the calculated average time to a new pandas dataframe average_time_data that contains stop id and the average time it takes to go to Flinders station.

In [ ]:
```
# Average Travel time from stop to CBD
average_time_data=average_time_data.reset_index(drop=True)
average_time_data.drop(['stop_sequence','stop_id_y'],axis=1,inplace=True)
```

In this section we will check through real_state_data data and identify the average time to the matching stop_id. This information is stored to a new column **travel_min_to_CBD**

In [ ]:
```
# Update Travel Time.
for k,krow in real_state_data.iterrows():
        temp_travel= list(average_time_data.loc[(average_time_data['stop_
        # If a stop doesnt match with average_time dataframe update time
        if not temp_travel:
            temp_travel=None

        real_state_data.loc[k,'travel_min_to_CBD']= temp_travel
```

We will handle the None values that was set to stations that **do not have a direct trip to flinders with 0.**

In [ ]:
```
# update nan values to 0
index=pd.isna(real_state_data['travel_min_to_CBD'])
real_state_data.loc[index,'travel_min_to_CBD']=0
# round the minutes.
real_state_data['travel_min_to_CBD']=round(real_state_data['travel_min_to
# Display final dataframe.
```

## Validating Travel flag and travel_timeCBD

In this section we will validate the work that has been done in the previous section, using the

knowledge that if there are no direct trip from a station to Flinders then the Transfer_Flag is set to 1 and the Average travel time to these rows should be 0.

In [ ]:
```
# Check if there are rows where there is no direct transfer with travel m
```

Here we check for scenario where the transfer flag is set to 0 or there is a direct connection to Flinders and the average time is 0. Upon checking we could see that the stopID closes to this property is Flinders which the reason why the average travel time is 0

In [ ]:
```
# Check for rows where Transfer flag 0 and travel_time is 0
```

# 5. Identify Suburb for property.

In this section we will identify the suburb that belong to a particular propertyID. We accomplish this by using the lattitude and longitude information that is available to use. The process in which this is done is shown below.

1. Read the shape file from VIC_LOCALITY_POLYGON_shp
2. convert property id into geopandas dataframe.
3. Fetch corresponding Suburbs.
4. Update real_state_date with Suburb.
5. Cleanup suburb column.

**Read the shape file from VIC_LOCALITY_POLYGON_shp**

We are given a shape file that contain the polygon that denotes the points that correspond to a particular suburb. We will use geopandas library to read the shape file. Geopandas helps to read this shape file and store it as a pandas dataframe. It is important to note that the column **geometry** column contain the **POLYGON** with all the coordinate points. **VIC_LOCA_2** column contains the suburb information that we will need to extract for a particular property lattitude and longitude.

In [ ]:
```
# 1. Read the Locality file.
Locality = gpd.read_file('VIC_LOCALITY_POLYGON_shp.shp')
```

**convert property id into geopandas dataframe.**

Geopandas data frame provides the **CRS or Coordinate Reference System** that help identify what type of metric is used to denote a point in the spatial aspect. We will first understand the CRS type of Locality geopandas data frame that was extracted from VIC_LOCALITY_POLYGON_shp shape file.

In [ ]:
```
# 2. Get the CRS type of Locality geopandas frame.
```

Here we create new columns geometry that contains the POINT information of the property .i.e the lattitude and Longitude tuple information. We will use the Shapely library that offers both Point

and Polygon function to acheive this tranformation.

In [ ]:
```
# 2. Create Geometry file.
```

Once we have the geometry information we can create a new geopandas dataframe with the same CRS value of Locality dataframe. Here we can see that Locality has the CRS as {'proj': 'longlat', 'ellps': 'GRS80', 'no_defs': True} which is set to gdf geo pandas dataframe. We create this temporary dataframe so that we can exploit the Spatial join function which is available in Geopandas to find if a point is in a polygon.

In [ ]:
```
# 2. Convert Realstate data into Geopandas dataframe. use CRS from Locali
gdf = gpd.GeoDataFrame(
    real_state_data[['property_id','geometry']], geometry=real_state_data
```

In [ ]:
```
# 2. Crs of Locality need to match to use Join.
```

In [ ]:
```
# 2. Display geopandas dataframe. created with property lat and long.
```

**Fetch corresponding Suburbs.**

Geopandas offers the spatial join **sjoin()** function that helps us to directly do a search for a spatial POINT inside a spatial POLYGON. Here we specify the parameters how as inner to denote the type of join we are doing, and op is the operation which is to be done by the spatial join, here we use within as to check if a POINT is in POLYGON we need to use **within** and if it was the opposite case we would have used **contains**. Here we specify the columns to which the join needs to be done as 'VIC_LOCA_2','geometry' from Locality dataframe as we need the suburb information that is stored in 'VIC_LOCA_2' column. This join result with suburb information is stored on to suburbs dataframe.

In [ ]:
```
# 3. Get the suburbs
```

**Update real_state_date with Suburb.**

Now we will use the pandas merge() function to fetch the corresponding propertyID suburb from suburd dataframe and update the real_state_date dataframe with correct information.

In [ ]:
```
# 4. merge suburb to real_state_data
```

**Cleanup suburb column.**

Based on the sample output we could see that SUburb information has first letter capitalized for instance Underbool but information from 'VIC_LOCA_2' is UNDERBOOL. So using str.capitalize() function we will convert the suburb column information to the correct form.

In [ ]:
```
# 5. Rename to Suburbs
real_state_data.rename(columns={'VIC_LOCA_2':'suburb'}, inplace = True)
```

In [ ]:
```
# 5. Capitalize first letter of suburb.
```

In [ ]:
```
# 5. Remove unwanted columns
```

### Write Output file

Now that all the required information as per the specification has been correct identified and calculated we need to set the correct format as per the specification. Using reindex() function we will change the order of columns to match the spec document. Using pandas to_csv() function we will create the required csv file with real state information.

In [ ]:
```

```

In [ ]:
```
real_state_data=real_state_data.reindex(columns=['property_id', 'lat', 'l
       'year', 'bedrooms', 'bathrooms', 'parking_space','Shopping_center_
       'Distance_to_train_station', 'travel_min_to_CBD','Transfer_flag','
       'Distance_to_hospital',
       'Supermarket_id', 'Distance_to_supermaket'
       ])
```

In [ ]:
```
# Write files to CSV
```

## 6. Data reshaping

In this section we will analyse the impact of data reshaping methods like Z-score standardisation, Min-Max standardisation and it effect on the input columns 'price','Distance_to_sc','travel_min_to_CBD','Distance_to_hospital'. This is done as we need to develop a linear model that will predict 'price' with predictors 'Distance_to_sc','travel_min_to_CBD','Distance_to_hospital'. It is important to note that for a linear model two important assumptions needs to be met which are Normality and Linearity. In the section we will explore the effect of each standardisation on the each of these columns.

In [ ]:
```
# store data to dataframe to perform transformation and build model.
price_model_data = real_state_data[['price','Distance_to_sc','travel_min_
# check for Datatype of all the 4 fields.
```

In [ ]:
```
# Convert price to Float to compare all four columns using describe
```

In [ ]:
```
# check the distribution of predictors for linear model.
```

We can see that **price is in Dollars (AUD)** , which completely on a different scale compared to **Distance_to_sc(Meters)** and **travel_min_to_CBD (Minutes)**. This makes the interpretation of coefficients in the linear regression model difficult. Similary **travel_min_to_CBD (Minutes)** predictor is of different scale compared to **price (AUD) , Distance_to_sc (Meters), Distance_to_hospital (Meters).**

It is important to identify the correct normalisation and transformation that needs to be applied to

these predictors that the all the columns are on the level field to make comparision.

## 6.1 Z-Score Normalisation (standardisation):

This transformation converts the data by taking the mean of a column and divides it by standard deviation to convert a column to standardise data (mean=0, SD=1).

In [ ]:

In [ ]:
```
# Standardised transform Using Zscore
std_scale = preprocessing.StandardScaler().fit(price_model_data[['price',
transfored_df = std_scale.transform(price_model_data[['price', 'Distance_
transfored_df
```

In [ ]:

In [ ]:
```
# Store into  dataframe and view the difference.
price_model_data['Z_Price'] = transfored_df[:,0] # Scaled price
price_model_data['Z_Distance_to_sc'] = transfored_df[:,1] # scaled distan
price_model_data['Z_travel_min_to_CBD'] = transfored_df[:,2] # scaled tra
```

In [ ]:
```
# explore the newly created columns.
```

In [ ]:
```
# Understand the distribution and spread of standardised columns.
```

### 6.1.1 Mean and SD before and After standardisation

In this section we can see that the columns 'price','Distance_to_sc','travel_min_to_CBD','Distance_to_hospital' are of different scales and metrics before doing the standardisation and the mean and standard deviation are extremely volatile before the standardisation is done. We can also see the Mean and SD is set 0 and 1 after the Z-Scale Standardisation.

In [ ]:
```
# Mean Before and after standardisation.
print('Mean before standardisation :\n Price: {:.2f}, Distance_to_sc: {:.
                                                ,price_model_data.Distance
                                                ,price_model_data.travel_m
                                                ,price_model_data.Distance
print('Mean after standardisation :\n Z_Price: {:.2f}, Z_Distance_to_sc:
                                                ,price_model_data.Z_Distan
                                                ,price_model_data.Z_travel
                                                ,price_model_data.Z_Distan
```

In [ ]:
```
# Standard deviation before and after zscale standardisation:

print('SD before standardisation :\n Price: {:.2f}, Distance_to_sc: {:.2f
                                                ,price_model_data.Distance
                                                ,price_model_data.travel_m
                                                ,price_model_data.Distance
print('SD after standardisation :\n Z_Price: {:.2f}, Z_Distance_to_sc: {:
```

```
                                                    ,price_model_data.Z_Distan
                                                    ,price_model_data.Z_travel
                                                    ,price_model_data.Z_Distan
```

### 6.1.2 Plot standardised columns.

In this section we will check both the Linearity and Normal aspect of the columns before and after standardisation and infer important information on the effect these standardisation has on the the data in these columns.

In [ ]:
```
# to plot the graphs
import matplotlib.pyplot as plt
```

In [ ]:
```
# Line graph shows the deviation of all four columns between mean = 0 and
price_model_data.Z_Price.plot(figsize=(20,5)
                              ,title='Distribution of price, Distance_to_
                              ,label='Z_Price'
                             ,legend='upper right'),
price_model_data.Z_Distance_to_sc.plot(label='Z_Distance_to_sc'
                                       ,legend='upper right'),
price_model_data.Z_travel_min_to_CBD.plot(label='Z_travel_min_to_CBD'
                                          ,legend='upper right'),
price_model_data.Z_Distance_to_hospital.plot(label='Z_Distance_to_hospita
```

### 6.1.3 Check Normality and Linearity : distribution of all columns

In [ ]:
```
# Figure to check the distribution of price, Distance_to_sc, travel_min_t
fig, axs = plt.subplots(2, 2,figsize=(10,10))
axs[0, 0].hist(price_model_data.price)
axs[0, 0].set_title('Input scale Price')
axs[0, 0].grid()
axs[0, 1].hist(price_model_data.Distance_to_sc)
axs[0, 1].set_title('Input scale Distance_to_sc')
axs[0, 1].grid()
axs[1, 0].hist(price_model_data.travel_min_to_CBD)
axs[1, 0].set_title('Input scale travel_min_to_CBD')
axs[1, 0].grid()
axs[1, 1].hist(price_model_data.Distance_to_hospital)
axs[1, 1].set_title('Input scale Distance_to_hospital')
```

In [ ]:
```
# Figure to check the distribution of price, Distance_to_sc, travel_min_t
fig, axs = plt.subplots(2, 2,figsize=(10,10))
axs[0, 0].hist(price_model_data.Z_Price)
axs[0, 0].set_title('Z-scale Price')
axs[0, 0].grid()
axs[0, 1].hist(price_model_data.Z_Distance_to_sc)
axs[0, 1].set_title('Z-scale Distance_to_sc')
axs[0, 1].grid()
axs[1, 0].hist(price_model_data.Z_travel_min_to_CBD)
axs[1, 0].set_title('Z-scale travel_min_to_CBD')
axs[1, 0].grid()
axs[1, 1].hist(price_model_data.Z_Distance_to_hospital)
```

```
axs[1, 1].set_title('Z-scale Distance_to_hospital')
```

**Inference from distribution**

1. From the histogram above, we can see that before and after standardisation we could see that the historgram has not changed.
2. **Price** and **Distance_to_hospital** is **right-skewed** this means that towards the end of the distribution there are fewer values.
3. **travel_min_to_CBD and Distance_to_sc** are closer to Gaussian distribution/Normal distribution.
4. We need to apply tranformation to Distance_to_hospital and Price to make it into Normal distrinution.

## 6.2 MinMax Normalisation:

This rescaling is similar to Zscale standardisation, here we will subtract minimum of a column with subtracted value of Maximum and minimum. Using this we will rescale the value of Price, Distance_to_hospital,travel_min_to_CBD and Distance_to_sc

In [ ]:
```
# Standardised transform Using minmax
minmax_scale = preprocessing.MinMaxScaler().fit(price_model_data[['price'
minmax_df = minmax_scale.transform(price_model_data[['price', 'Distance_t
minmax_df[0:5]
```

In [ ]:
```
# Store into  dataframe and view the difference.
price_model_data['MM_Price'] = minmax_df[:,0] # Scaled price
price_model_data['MM_Distance_to_sc'] = minmax_df[:,1] # scaled distance
price_model_data['MM_travel_min_to_CBD'] = minmax_df[:,2] # scaled travel
```

### 6.2.1 Mean and SD before and After standardisation

In this section we can see that the columns 'price','Distance_to_sc','travel_min_to_CBD','Distance_to_hospital' are of different scales and metrics before doing the standardisation and the mean and standard deviation are extremely volatile before the standardisation is done. We can also see the Mean and SD is 0 and less than 1 respectively after the Min-Max Standardisation.

In [ ]:
```
# Mean Before and after standardisation.
print('Mean before standardisation :\n Price: {:.2f}, Distance_to_sc: {:.
                                        ,price_model_data.Distance_
                                        ,price_model_data.travel_m
                                        ,price_model_data.Distance_
print('Mean after standardisation :\n Z_Price: {:.2f}, Z_Distance_to_sc:
                                        ,price_model_data.MM_Dista
                                        ,price_model_data.MM_trave
                                        ,price_model_data.MM_Dista
```

In [ ]:
```
# Standard deviation before and after zscale standardisation:
```

```
print('SD before standardisation :\n Price: {:.2f}, Distance_to_sc: {:.2f
                                                    ,price_model_data.Distance_
                                                    ,price_model_data.travel_m
                                                    ,price_model_data.Distance_
print('SD after standardisation :\n Z_Price: {:.2f}, Z_Distance_to_sc: {:
                                                    ,price_model_data.MM_Dista
                                                    ,price_model_data.MM_trave
                                                    ,price_model_data.MM_Dista
```

### 6.2.3 Check Normality : Min-Max distribution of all columns

In this section we will check both the Linearity and Normal aspect of the columns before and after standardisation and infer important information on the effect these standardisation has on the the data in these columns.

In [ ]:
```
# Figure to check the distribution of price, Distance_to_sc, travel_min_t
fig, axs = plt.subplots(2, 2,figsize=(10,10))
axs[0, 0].hist(price_model_data.MM_Price)
axs[0, 0].set_title('Min-Max scale Price')
axs[0, 0].grid()
axs[0, 1].hist(price_model_data.MM_Distance_to_sc)
axs[0, 1].set_title('Min-Max scale Distance_to_sc')
axs[0, 1].grid()
axs[1, 0].hist(price_model_data.MM_travel_min_to_CBD)
axs[1, 0].set_title('Min-Max scale travel_min_to_CBD')
axs[1, 0].grid()
axs[1, 1].hist(price_model_data.MM_Distance_to_hospital)
axs[1, 1].set_title('Min-Max scale Distance_to_hospital')
```

We can see that from the histograms shown above, Input scale, Z scale and Min-Max scale standardisation do not change the distribution of predictors. **Price and Distance_to_Hospital** is still **right skewed**

### 6.2.4 Check Linearity : distribution of Distance_to_sc, travel_min_to_CBD, Distance_to_hospital with Y= Price

In [ ]:
```
# Create a Figure of size 20, 20 to plot the graphs.
fig, axs = plt.subplots(3, 3,figsize=(20,20))

# use axs to denote where the plot is drawn.
# input scale with out any standardisation.
axs[0, 0].scatter(price_model_data.price,price_model_data.Distance_to_hos
axs[0, 0].set_title('Input scale y=Price vs x=Distance_to_hospital')
axs[0, 0].grid()
axs[0, 1].scatter(price_model_data.price,price_model_data.Distance_to_sc,
axs[0, 1].set_title('Input scale y=Price vs x=Distance_to_sc')
axs[0, 1].grid()
axs[0, 2].scatter(price_model_data.price,price_model_data.travel_min_to_C
axs[0, 2].set_title('Input scale y=Price vs x=travel_min_to_CBD')
axs[0, 2].grid()
```

```
# Z-scale standardasation on all columns.
axs[1, 0].scatter(price_model_data.Z_Price,price_model_data.Z_Distance_to_
axs[1, 0].set_title('Z-scale y=Price vs x=Distance_to_hospital')
axs[1, 0].grid()
axs[1, 1].scatter(price_model_data.Z_Price,price_model_data.Z_Distance_to_
axs[1, 1].set_title('Z-scale y=Price vs x=Distance_to_sc')
axs[1, 1].grid()
axs[1, 2].scatter(price_model_data.Z_Price,price_model_data.Z_travel_min_t
axs[1, 2].set_title('Z-scale y=Price vs x=travel_min_to_CBD')
axs[1, 2].grid()

# minmax standardasation on all columns.
axs[2, 0].scatter(price_model_data.MM_Price,price_model_data.MM_Distance_t
axs[2, 0].set_title('Min-Max scale y=Price vs x=Distance_to_hospital')
axs[2, 0].grid()
axs[2, 1].scatter(price_model_data.MM_Price,price_model_data.MM_Distance_t
axs[2, 1].set_title('Min-Max scale y=Price vs x=Distance_to_sc')
axs[2, 1].grid()
axs[2, 2].scatter(price_model_data.MM_Price,price_model_data.MM_travel_mi
axs[2, 2].set_title('Min-Max scale y=Price vs x=travel_min_to_CBD')
axs[2, 2].grid()
```

#### 6.2.5 Inference from graph

- From this scatter plot above we can see that even though standardisation has been done to price, Distance_to_sc, travel_min_to_CBD, Distance_to_hospital columns, the distribution remains the same. There is no loss of information to the overall spread of data.
- When creating the linear regression model the coefficients of Distance_to_sc, travel_min_to_CBD, Distance_to_hospital as predictors will denote the contribution of a predictor in predicting Price.
- Initially the scale of all the columns where different and after performing the standardisation columns are either within mean =0 and sd=1 or within min=0 or max=1.
- Using Z-scale or Min-Max to create the model will help understand the linearlity aspect of Distance_to_sc, travel_min_to_CBD, Distance_to_hospital better.

# 7.Data Transformation to check normality and linearity

In order make sure that the normality condition of lineat regression is met we will to make changes/ tranformation to predictor and check if applying the following transformation makes the predictors ditribution into normal/guassian. We will apply the following transformations.

- Root Transformation
- Square/Power Transformation
- Log transformation
- Box-Cox transformation

Of all the predictors we could see from the histogram that **Price and Distance_to_Hospital** are **right skewed** and we will apply the transformation to these fields and see the changes mainly, however we would still want to see if there are any changes to normality for travel_min_to_CBD

and Distance_to_sc after applying these transformations.

## 7.1 Root transformation.

This transformation is lossy in nature, meaning once the date is converted to root form the original information is now in a different dimension on which the relationship and pattern is studied. In most cases this can be use to fix right skewed data.

In [ ]:

**Perform root transformation on columns.**

In [ ]:
```
# using the Numpy sqrt function create new columns.
price_model_data['root_price']=np.sqrt(price_model_data.price)
price_model_data['root_Distance_to_sc']=np.sqrt(price_model_data.Distance_
price_model_data['root_travel_min_to_CBD']=np.sqrt(price_model_data.trave
price_model_data['root_Distance_to_hospital']=np.sqrt(price_model_data.Di
```

**Check Normality and Linearity after Root Tranformation on all columns**

In [ ]:
```
# PRICE before and after root transformation. Normality
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['price'])
axs[0].set_title('Price Before transform')
axs[0].grid()
axs[1].hist(price_model_data['root_price'])
axs[1].set_title('Price After Root Tranform')
axs[1].grid()
```

The right skew is has reduced and the data seems to look a lot like Normal distribution as Price is transformed into root of price. Also the scale seems to reduced and looks more interpretable compared to Input Price field. It is worth noting that if we used Min-Max price or Z-scale stadardised price and applied the root transformation the data of input will still be right skewed and the transformed data will look a lot closer to normal distribution.

In [ ]:
```
# Distance_to_hospital before and after root transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['Distance_to_hospital'])
axs[0].set_title('Distance_to_hospital Before transform')
axs[0].grid()
axs[1].hist(price_model_data['root_Distance_to_hospital'])
axs[1].set_title('Distance_to_hospital After Root Tranform')
axs[1].grid()
```

Similar to Price, applying root transformation on Distance_to_hospital column has removed the right skew from this predictor and transfored data looks a lot closed to normal. It is worth to keep root transformation in mind while developing the linear model for both Price and distance_to Hospital columns.

In [ ]:
```
# travel_min_to_CBD before and after root transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['travel_min_to_CBD'])
axs[0].set_title('travel_min_to_CBD Before transform')
axs[0].grid()
axs[1].hist(price_model_data['root_travel_min_to_CBD'])
axs[1].set_title('travel_min_to_CBD After Root Tranform')
axs[1].grid()
```

In [ ]:
```
# travel_min_to_CBD before and after root transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['Distance_to_sc'])
axs[0].set_title('Distance_to_sc Before transform')
axs[0].grid()
axs[1].hist(price_model_data['root_Distance_to_sc'])
axs[1].set_title('Distance_to_sc After Root Tranform')
axs[1].grid()
```

**Check linearity**

we will plot a scatter plot and understand if there is improvement to columns after performing root tranformation. Although Normality is improved by using Root tranformation we need to understand if the linearity is improved compated to the initial spread of data.

In [ ]:
```
# Create a Figure of size 20, 20 to plot the graphs.
fig, axs = plt.subplots(1, 3,figsize=(20,5))

# use axs to denote where the plot is drawn.
# input scale with out any standardisation.
axs[0].scatter(price_model_data.root_price,price_model_data.root_Distance_
axs[0].set_title('y=root_Price vs x=root_Distance_to_hospital')
axs[0].grid()
axs[1].scatter(price_model_data.root_price,price_model_data.root_Distance_
axs[1].set_title('y=root_Price vs x=root_Distance_to_sc')
axs[1].grid()
axs[2].scatter(price_model_data.root_price,price_model_data.root_travel_m
axs[2].set_title('y=root_Price vs x=root_travel_min_to_CBD')
```

**Inference from Root transformation.**

1. Applying root transformation is certainly a good option for right skewed columns Price and Distance_to_Hospital.
2. It is worth to note that Normality is better using Root transformation on Distance_to_sc and travel_min_to_CBD. This is because Root reduce the peaks and levels the columns making it appear more normal.
3. Linearity has taken a hit by using root transformation and there are no visible linear patterns between transfored predictors, which is an important assumption for linear regression model(linear additive summation).

## 7.2 Power transformation.

Power transformation helps to amplify the data where there are uneven spread to data. In our predictors as there right skewed information this will worsen the spread and mostly likely be a bad choice to transform the data.

**Perform root transformation on columns.**

In [ ]:
```
# using the Numpy power function create new columns.
price_model_data['pow_price']=np.power(price_model_data.price,2)
price_model_data['pow_Distance_to_sc']=np.power(price_model_data.Distance_
price_model_data['pow_travel_min_to_CBD']=np.power(price_model_data.trave
```

**Check Normality and Linearity after Root Tranformation on all columns**

In [ ]:
```
# PRICE before and after square transformation. Normality
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['price'])
axs[0].set_title('Price Before transform')
axs[0].grid()
axs[1].hist(price_model_data['pow_price'])
axs[1].set_title('Price After Square Tranform')
```

We can see that the right skew has worsened after applying power transformation on already right skewed data. distance_to_Hospital will have the same effect as this columns is also right skewed. We will still compare the effect on travel_min_to_CBD and Distance_to_sc.

In [ ]:
```
# travel_min_to_CBD before and after square transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['travel_min_to_CBD'])
axs[0].set_title('travel_min_to_CBD Before transform')
axs[0].grid()
axs[1].hist(price_model_data['pow_travel_min_to_CBD'])
axs[1].set_title('travel_min_to_CBD After Square Tranform')
```

In [ ]:
```
# travel_min_to_CBD before and after square transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['Distance_to_sc'])
axs[0].set_title('Distance_to_sc Before transform')
axs[0].grid()
axs[1].hist(price_model_data['pow_Distance_to_sc'])
axs[1].set_title('Distance_to_sc After Square Tranform')
```

In [ ]:
```
# Create a Figure of size 20, 20 to plot the graphs.
fig, axs = plt.subplots(1, 3,figsize=(20,5))

# use axs to denote where the plot is drawn.
# input scale with out any standardisation.
axs[0].scatter(price_model_data.pow_price,price_model_data.pow_Distance_t
axs[0].set_title('y=pow_Price vs x=pow_Distance_to_hospital')
```

```
axs[0].grid()
axs[1].scatter(price_model_data.pow_price,price_model_data.pow_Distance_t
axs[1].set_title('y=pow_Price vs x=pow_Distance_to_sc')
axs[1].grid()
axs[2].scatter(price_model_data.pow_price,price_model_data.pow_travel_min
axs[2].set_title('y=pow_Price vs x=pow_travel_min_to_CBD')
```

**Inference from Square transformation.**

1. Right skewed data becomes more right skewed which make this transformation useless to make columns to normal.
2. Linerity has not improved using the square transformation for these columns.

## 7.3 Log transformation.

Log transformation is handy in handling right skewed data as the log of huge numbers are smaller this way the distribution becomes close to normal. Its work noting that there is no value defined for 0 , so in order to make sure log runs without any issues we will subsitite a very small value rows with 0 in them. Also there are many log that can be applied to data like log to base 2, log to base 10, natural log. Here we will use natural log.

**Update zero in columns as log 0 is not defined.**

In [ ]:
```
# subsitute small value close to zero like 0.00001 as log(0) is undefined
price_model_data.price.loc[(price_model_data.price == 0)]=0.00001
price_model_data.Distance_to_sc.loc[(price_model_data.Distance_to_sc == 0
price_model_data.Distance_to_hospital.loc[(price_model_data.Distance_to_h
```

**Perform root transformation on columns.**

In [ ]:
```
# using the Numpy log function create new columns.
price_model_data['log_price']=np.log(price_model_data.price)
price_model_data['log_Distance_to_sc']=np.log(price_model_data.Distance_t
price_model_data['log_travel_min_to_CBD']=np.log(price_model_data.travel_
```

**Visualization of before and after log transformation.**

In [ ]:
```
# PRICE before and after log transformation. Normality
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['price'])
axs[0].set_title('Price Before transform')
axs[0].grid()
axs[1].hist(price_model_data['log_price'])
axs[1].set_title('Price After Log Tranform')
```

It clearly evident the right skewness removed by using the log, in this case natural log. Price data is transformed into normal distribution after applying log transformation on it. We will do the same

to other right skewed data

In [ ]:
```
# Distance_to_hospital before and after log transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['Distance_to_hospital'])
axs[0].set_title('Distance_to_hospital Before transform')
axs[0].grid()
axs[1].hist(price_model_data['log_Distance_to_hospital'])
axs[1].set_title('Distance_to_hospital After Log Tranform')
axs[1].grid()
```

Similary to price, distance_to_hospital column is transfored into a normal distribution by applying log transformation. This is much better than root transformation as the ditribution is almost normal/ gaussian which is needed to create the linear model.

In [ ]:
```
# travel_min_to_CBD before and after square transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['travel_min_to_CBD'])
axs[0].set_title('travel_min_to_CBD Before transform')
axs[0].grid()
axs[1].hist(price_model_data['log_travel_min_to_CBD'])
axs[1].set_title('travel_min_to_CBD After Log Tranform')
```

It is evident that using log transformation on travel_min_to_CBD columns has adverse effect, this transforms the data to left skewed which makes this not feasible for travel_min_to_CBD predictor.

In [ ]:
```
# travel_min_to_CBD before and after square transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['Distance_to_sc'])
axs[0].set_title('Distance_to_sc Before transform')
axs[0].grid()
axs[1].hist(price_model_data['log_Distance_to_sc'])
axs[1].set_title('Distance_to_sc After Log Tranform')
```

For Distance_to_sc which was right heavy right tailed get converted into almost normal distribution using the log transformation.

In [ ]:
```
# Check for Linearity.
# Create a Figure of size 20, 20 to plot the graphs.
fig, axs = plt.subplots(1, 3,figsize=(20,5))

# use axs to denote where the plot is drawn.
# input scale with out any standardisation.
axs[0].scatter(price_model_data.log_price,price_model_data.log_Distance_t
axs[0].set_title('y=log_Price vs x=log_Distance_to_hospital')
axs[0].grid()
axs[1].scatter(price_model_data.log_price,price_model_data.log_Distance_t
axs[1].set_title('y=log_Price vs x=log_Distance_to_sc')
axs[1].grid()
axs[2].scatter(price_model_data.log_price,price_model_data.log_travel_min
axs[2].set_title('y=log_Price vs x=log_travel_min_to_CBD')
axs[2].grid()
```

**Inference from Log transformation.**

1. Normality is restored using Log transformation on right skewed data, this includes Price, Distance_to_hospital and Distance_to_sc.
2. travel_min_to_CBD tranformed into Leftskewed which incidcates this should not be applied to this predictor.
3. There is no evidence on linearity although data is evenly spread for log of predictor with log of price.

## 7.4 Box-Cox transformation.

Right skewed data or power law function which has heavy tails to the right and high peaks on the left can be easily converted to normal/guassian distribution using the box-cox transformation. We will apply this transformation on Price, Distance_to_Hospital, Distance_to_Sc and Travel_time_CBD columns and understand the effect of this transformation on them. In mathematical terms it is given by

$y(\lambda) = \frac{y^{\lambda}-1}{\lambda}$ for $\lambda \neq 0$ this function converges to Log(y) if $\lambda = 0$

IMPORTANT: For box_cox to work the data points should be positive and it does not work otherwise. All our data points are positive and we wont have any problems in implementing this. We cannot have log(0) so we will include a small value close to 0 like 0.00001 to 0 values in all the columns before applying box cox tranformation.

**Perform root transformation on columns.**

In [ ]:
```
# using the Numpy log function create new columns.
price_model_data['box_price'],l1=stats.boxcox(price_model_data.price)
price_model_data['box_Distance_to_sc'],l2=stats.boxcox(price_model_data.D
price_model_data['box_travel_min_to_CBD'],l3=stats.boxcox(price_model_dat
```

**Display Lambda values after transformation.**

In [ ]:
```
# Display lambda values for columns.
print( 'Lambda for Box-Cox transformation is given below:', '\nprice:',l1
```

**Visualization of before and after Box-Cox transformation.**

In [ ]:
```
# PRICE before and after Box-Cox transformation. Normality
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['price'])
axs[0].set_title('Price Before transform')
axs[0].grid()
axs[1].hist(price_model_data['box_price'])
axs[1].set_title('Price After Box-Cox Tranform')
```

On comparision to log transformation box-cox transformation is able to tranform the rightskewed Price column data into a Gaussian/Normal distribution. We will apply the same to distance_to_Hospital which is also right-Skewed.

In [ ]:
```
# Distance_to_hospital before and after Box-Cox transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['Distance_to_hospital'])
axs[0].set_title('Distance_to_hospital Before transform')
axs[0].grid()
axs[1].hist(price_model_data['box_Distance_to_hospital'])
axs[1].set_title('Distance_to_hospital After Box-Cox Tranform')
axs[1].grid()
```

Box-Cox transformation on distance_to_Hospital is also very close to normal distribution. We will use this transformed data while building the linear model.

In [ ]:
```
# travel_min_to_CBD before and after Box-Cox transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['travel_min_to_CBD'])
axs[0].set_title('travel_min_to_CBD Before transform')
axs[0].grid()
axs[1].hist(price_model_data['box_travel_min_to_CBD'])
axs[1].set_title('travel_min_to_CBD After Box-Cox Tranform')
```

There is significant difference to distribution of travel_min_to_CBD which had high peak after Box-Cox tranformation the data is evenly distributed and looks more like a normal distribution.

In [ ]:
```
# travel_min_to_CBD before and after Box-Cox transformation.
fig, axs = plt.subplots(1, 2,figsize=(15,5))
axs[0].hist(price_model_data['Distance_to_sc'])
axs[0].set_title('Distance_to_sc Before transform')
axs[0].grid()
axs[1].hist(price_model_data['box_Distance_to_sc'])
axs[1].set_title('Distance_to_sc After Box-Cox Tranform')
```

The right skew that was visible with the input scale of Distance_to_SC is absent in Box-Cox tranformed data. Tranformed data looks more Normal/Guassian which notes that on comparision to all tranformation done previously box-cox is highly effective in converting right-skewed data to more normally distributed.

In [ ]:
```
# Check for Linearity.
# Create a Figure of size 20, 20 to plot the graphs.
fig, axs = plt.subplots(1, 3,figsize=(20,5))
```

```
# use axs to denote where the plot is drawn.
# input scale with out any standardisation.
axs[0].scatter(price_model_data.box_price,price_model_data.box_Distance_t
axs[0].set_title('y=box_Price vs x=box_Distance_to_hospital')
axs[0].grid()
axs[1].scatter(price_model_data.box_price,price_model_data.box_Distance_t
axs[1].set_title('y=box_Price vs x=box_Distance_to_sc')
axs[1].grid()
axs[2].scatter(price_model_data.box_price,price_model_data.box_travel_min
axs[2].set_title('y=box_Price vs x=box_travel_min_to_CBD')
```

**Inference from Box-Cox transformation.**

1. On comparision with all the tranforamtion, root, square/power, Log , Box-Cox is able to completely convert the right skewed data to a almost normal distribution. The normality of columns is acheived by using Box-Cox tranformation which is recommended to be used while developing the linear model.
2. Linearity is still unclear as non of the transformation is able to clearly provide a linear relationship with Price. This could mean that the linearity may be acheived while using better features or doing feature engineering that helps to identify tranformed feature that could add value to linear model.

# Reference


1. Pandas-read-xml. (n.d.). Retrieved from https://pypi.org/project/pandas-read-xml/ (https://pypi.org/project/pandas-read-xml/)
2. Varun. (2019, January 11). Pandas : How to create an empty DataFrame and append rows & columns to it in python. Retrieved from https://thispointer.com/pandas-how-to-create-an-empty-dataframe-and-append-rows-columns-to-it-in-python/ (https://thispointer.com/pandas-how-to-create-an-empty-dataframe-and-append-rows-columns-to-it-in-python/)
3. Pulo, N., Bugis, R. A., & Jamak, A. (2020, April 24). Importance and goals of filtering GTFS. Retrieved from https://www.atlantbh.com/importance-and-goals-of-filtering-gtfs/ (https://www.atlantbh.com/importance-and-goals-of-filtering-gtfs/)
4. Reference | Static Transit | Google Developers. (n.d.). Retrieved from https://developers.google.com/transit/gtfs/reference#tripstxt (https://developers.google.com/transit/gtfs/reference#tripstxt)
5. Point in Polygon & Intersect¶. (n.d.). Retrieved from https://automating-gis-processes.github.io/2016/Lesson3-point-in-polygon.html (https://automating-gis-processes.github.io/2016/Lesson3-point-in-polygon.html)
6. Soma, J. (Director). (n.d.). Spatial joins in geopandas [Video file]. Retrieved from https://www.youtube.com/watch?v=y85IKthrV-s&feature=youtu.be (https://www.youtube.com/watch?v=y85IKthrV-s&feature=youtu.be)
7. Bigbugbigbug 36.4k3434 gold badges6969 silver badges9191 bronze badges, GarrettGarrett 32.4k55 gold badges5151 silver badges4747 bronze badges, Cs95cs95 240k6262 gold badges414414 silver badges479479 bronze badges, JeffJeff 102k1717 gold badges190190 silver badges163163 bronze badges, Firelynxfirelynx 22.6k44 gold badges7878 silver badges8787 bronze badges, User443854user443854 5, . . . M-dzm-dz 2. (1963, February

01). How to deal with SettingWithCopyWarning in Pandas. Retrieved from https://stackoverflow.com/questions/20625582/how-to-deal-with-settingwithcopywarning-in-pandas (https://stackoverflow.com/questions/20625582/how-to-deal-with-settingwithcopywarning-in-pandas)

8. Scipy.stats.boxcox. (n.d.). Retrieved from https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.boxcox.html (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.boxcox.html)
9. Australia Victoria. (2016, January 31). Retrieved from https://en.wikipedia.org/wiki/Module:Location_map/data/Australia_Victoria (https://en.wikipedia.org/wiki/Module:Location_map/data/Australia_Victoria)
10. Managing Projections. (n.d.). Retrieved from https://geopandas.org/projections.html (https://geopandas.org/projections.html)