FIT5196-S2-2020 Assessment 3

Student Name: Ananya Pandey

Student ID: 30757924

Date: 18/11/2020

Environment: Python 3.7.3.final.0 and Anaconda 4.8.3

Libraries used:

- pandas 1.0.3 (for data manipulation and analysis, included in Python 3.7.3)
- numpy 1.16.2 (for working with arrays, included in Python 3.7.3)
- itertools (includes a set of functions for working with iterable (sequence-like) data sets, included in Python 3.7.3)
- sklearn (it features various classification, regression and clustering algorithms, included in Python 3.7.3)
- matplotlib 3.0.3 (is a plotting library for the Python programming language and its numerical mathematics extension numpy)
- seaborn 0.9.0 (is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics)
- xml.etree.ElementTree (it is used to read and parse the xml file)
- math (it is used to apply some math funnctions)
- json (it is used in reading and parsing data from json file format)
- tabula (it is used to read and parse the pdf file format)
- shapefile (it provides read and write support for the ESRI Shapefile format. The Shapefile format is a popular Geographic Information System vector data format)
- scipy (it is used for scientific computing and technical computing)

Task 1: Data Integration

In this task, it is required to integrate the input datasets into one dataset with the given schema.

Datasets:

- **30757924**: folder containing the datasets hospitals.pdf, real_state.json, real_state.xml, shopingcenters.xlsx and supermarkets.html
- **Vic_suburb_boundary**: folder containing the shape files which are used to fetch the suburb names.
- **GTFS_Melbourne_Train_Information**: folder containing all the public transport details having routes.txt, stops.txt, stop_times.txt, agency.txt, calendar.txt, calendar_dates.txt, shapes.txt and trips.txt files.

Schema:

property_id: A unique id for the property
lat: The property latitude
lng: The property longitude
addr_street: The property address
suburb: The property suburb. Default value: “not available”
price: The property price
property_type: The type of the property
year: Year of sold
bedrooms: Number of bedrooms
bathrooms: Number of bathrooms
parking_space: The number of parking space of the property
Shopping_center_id: The closest shopping center to the property. Default value: “not available”

Distance_to_sc: The Euclidean distance from the closest shopping center to the property. Default value: 0
Train_station_id: The closest train station to the property. Default value: 0
Distance_to_train_station: The Euclidean distance from the closest train station to the property. Default value: 0
travel_min_to_CBD: The average travel time (minutes) from the closest train station to the “Flinders street” station on weekdays (i.e. Monday-Friday) departing between 7 to 9 am. For example, if there are 3 trip departing from the closest train station to the Flinders street station on weekdays between 7-9am and each take 6, 7, and 8 minutes respectively, then the value of this column for the property should be (6+7+8)/3. If there are any direct transfers between the closest station and Flinders street station, only the average of direct transfers should be calculated . Default value: 0
Transfer_flag: A Boolean attribute indicating whether there is a direct trip to the Flinders street station from the closest station between 7-9am on the weekdays. This flag is 0 if there is a direct trip (i.e. no transfer between trains is required to get from the closest train station to the Flinders station) and 1 otherwise. Default value: -1
Hospital_id: The closest hospital to the property. Default value: “not available”
Distance_to_hospital: The Euclidean distance from the closest hospital to the property. Default value: 0
Supermarket_id: The closest supermarket to the property. Default value: “not available”
Distance_to_supermaket: The Euclidean distance from the closest supermarket to the property. Default value: 0

Importing the libraries

In [ ]:

```
import pandas as pd
import json
import tabula
import xml.etree.ElementTree as etree
import numpy
import matplotlib
%matplotlib inline
import shapefile
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
from math import radians, cos, sin, asin, sqrt
from matplotlib import pyplot as plt
from sklearn import preprocessing
import math
import seaborn as sb
from sklearn.linear_model import LinearRegression
import numpy as np
from scipy import stats

import warnings
warnings.filterwarnings('ignore')

import os
os.chdir('C:\\Users\\ananya\\Desktop\\Wrangling ass3')
os.getcwd()
```

1. Read the data files

The files `real_state.json` , `real_state.xml` , `hospitals.pdf` , `shopingcenters.xlsx` , `supermarkets.html` from the folder of the student id `30757924` are read into dataframes using the pandas library in python.

1.1. Read the real_state.json file and remove the duplicate rows

The `real_state.json` file from the 30757924 folder is read as a dataframe using the pandas library and the json module. This file contains the information of the properties i.e. the property_id, latitude, longitude, address of the property, price, type of property, the year, no. of bathrooms, no. of bedrooms and no. of parking spaces of each of the property.

After reading the data into a pandas dataframe, the rows containing NA are then removed from the dataframe. In order to make the data complete, the null rows are removed.

The rows having duplicate property_id are found by using the value_counts() function on each of the property and the duplicated rows are then dropped from the dataframe.

In [ ]:

```
#load the json file
with open('./30757924/real_state.json') as f:
    real_state_json_data = json.load(f)
```

In [ ]:

```
real_state_json_data = pd.DataFrame( real_state_json_data)
```

In [ ]:

```
real_state_json_data
```

In [ ]:

```
real_state_json_data['property_id'].nunique()
```

In [ ]:

```
#remove na rows
real_state_json_data = real_state_json_data.dropna()
```

In [ ]:

```
real_state_json_data
```

In [ ]:

```
duplicates = real_state_json_data['property_id'].value_counts !=1
```

In [ ]:

```
real_state_json_data.drop_duplicates(['property_id'], keep='first', inplace=True)
```

In [ ]:

```
real_state_json_data.shape
```

In [ ]:

```
real_state_json_data.dtypes
```

In [ ]:

```
real_state_json_data = real_state_json_data.astype({"property_id": int, "lat": float,
"lng": float, "price": int, "year": int, "bedrooms": int, "bathrooms": int, "parking_sp
ace": int})
```

In [ ]:

```
real_state_json_data.dtypes
```

1.2. Read the shopingcenters.xlsx file

The `shopingcenters.xlsx` file from the 30757924 folder is read as a dataframe using the pandas library. The shopingcenters.xlsx file contains information about the shopping centers i.e. the shoppingcenter id, the latitude and longitude of each of the shopping center.

For reading into a dataframe, the columns are renamed according to the names in the excel file. There is an extra column in the file which has the index of each of the row. This extra column is unnecessary and not required so it has been dropped and the final dataframe for the shopping centers contains the id, latitude and longitude of each of the shopping centers.

In [ ]:

```
#read the excel data file
excel_data = pd.ExcelFile("./30757924/shopingcenters.xlsx")
shopingcenters_df=pd.DataFrame()
```

In [ ]:

```
for i in excel_data.sheet_names:
    df = excel_data.parse(i, header=None)
    df.columns=[" ", "sc_id", "lat", "lng"]
    df = df.drop(df.index[0])
    df = df.drop(df.columns[0], axis=1)
    shopingcenters_df = shopingcenters_df.append(df, ignore_index=True)
```

1.3. Read the hospitals.pdf file

The `hospitals.pdf` file from the 30757924 folder is read as a dataframe using the tabula module and pandas library. The hospitals.pdf file contains information about the hospitals i.e. the hospital_id, the latitude, longitude and the name of each of the hospital.

For reading into a dataframe, the hospitals.pdf file is read as a list using the tabula module and then each of the page is appended together in one single dataframe. Then the unnecesary columns which are not useful are dropped. The rows having null values are also dropped from the final dataframe having the hospitals data.

In [ ]:

```
#!pip install tabula-py
```

In [ ]:

```
#declare the path of the file
file_path = "./30757924/hospitals.pdf"
#Convert the file
pdf_list = tabula.read_pdf(file_path, pages="all")
```

In [ ]:

```
#initialise the dataframe for saving the pdf
pdf_df=pd.DataFrame()
pdf_df = pdf_list[0].append([pdf_list[1], pdf_list[2], pdf_list[3], pdf_list[4]], ignor
e_index=True)
```

In [ ]:

```
pdf_df = pdf_df.drop(pdf_df.columns[5:], axis=1)
```

In [ ]:

```
pdf_df = pdf_df.drop(pdf_df.columns[0], axis=1)
```

In [ ]:

```
#drop the rows with null values
pdf_df = pdf_df.dropna()
```

In [ ]:

```
pdf_df=pdf_df.reset_index(drop=True)
```

In [ ]:

```
pdf_df
```

1.4. Read the real_state.xml file

The `real_state.xml` file from the 30757924 folder is read as a dataframe using the pandas library and the xml module. This file contains the information of the properties i.e. the property_id, latitude, longitude, address of the property, price, type of property, the year, no. of bathrooms, no. of bedrooms and no. of parking spaces of each of the property.

After reading the data into a pandas dataframe, the rows containing NA are then removed from the dataframe. In order to make the data complete, the null rows are removed.

The rows having duplicate property_id are found by using the value_counts() function on each of the property and the duplicated rows are then dropped from the dataframe.

In [ ]:

```
#declare the path of xml file
xml_file_path = "./30757924/real_state.xml"
xml_data = open(xml_file_path).readline()[2:][:-1]
```

In [ ]:

```
xml_data_text = etree.fromstring(xml_data)
```

In [ ]:

```
for i in range(len(xml_data_text)):
    xml_dict = dict()
    for elem in xml_data_text[i]:
        xml_dict[elem.tag] = elem.text
    xml_df = pd.DataFrame.from_dict(xml_dict, orient="index")
    xml_df.columns = [xml_data_text[i].tag]
    if i==0:
        xml_df_final = xml_df
    else:
        xml_df_final = xml_df_final.join(xml_df, how="outer")
```

In [ ]:

```
xml_df_final
```

In [ ]:

```
xml_df_final['property_id'].nunique()
```

In [ ]:

```
#remove rows having null values
xml_df_final = xml_df_final.dropna()
```

In [ ]:

```
xml_df_final
```

In [ ]:

```
duplicates = xml_df_final['property_id'].value_counts !=1
xml_df_final.drop_duplicates(['property_id'], keep='first', inplace=True)
xml_df_final.shape
```

In [ ]:

```
xml_df_final.dtypes
```

In [ ]:

```
xml_df_final = xml_df_final.astype({"property_id": int, "lat": float, "lng": float, "pr
ice": int, "year": int, "bedrooms": int, "bathrooms": int, "parking_space": int})
```

In [ ]:

```
xml_df_final.dtypes
```

1.5. Read the supermarkets.html file

The `supermarkets.html` file from the 30757924 folder is read as a dataframe using the pandas library. The supermarkets.html file contains information about the supermarkets i.e. the supermarket id, the latitude and longitude of each of the supermarket.

For reading into a dataframe, the html file is read using the read_html() function of the pandas library. This extra column is unnecessary and not required so it has been dropped and the final dataframe for the supermarkets contains the id, latitude and longitude of each of the supermarkets.

In [ ]:

```
#read the .html file into a dataframe
html_df = pd.read_html('./30757924/supermarkets.html')
html_df = html_df[0]
```

In [ ]:

```
html_df = html_df.drop(html_df.columns[0], axis=1)
```

In [ ]:

```
html_df
```

1.6. Read the vic_suburb_boundary shapefiles

The Shapefile format is a popular Geographic Information System vector data format created by Esri.

The `vic_suburb_boundary` folder contains the shapefiles for Victoria's suburb boundary. The shapefiles can be read using the shapefile module in python. The shapefile.Reader function helps in reading the shapefiles. This data is going to be further used in order to find the suburb for each of the property according to the address of the property.

In [ ]:

```
#!pip install pyshp
```

In [ ]:

```
#read the shapefile
sf = shapefile.Reader("./vic_suburb_boundary/VIC_LOCALITY_POLYGON_shp")
```

1.7. Merge the real_state files

Two files having the real_state data have been provided namely `real_state.json` and `real_state.xml` . As both of the files reflect on the same set of data, thus they have been merged into one single datframe containing the entire real_state data.

In [ ]:

```
#merge the real state files
real_state_df = real_state_json_data.append(xml_df_final, ignore_index=True)
```

In [ ]:

```
real_state_df
```

In [ ]:

```
duplicates = real_state_df['property_id'].value_counts !=1
```

In [ ]:

```
real_state_df.shape
```

In [ ]:

```
real_state_df.drop_duplicates(['property_id'], keep='first', inplace=True, ignore_index
=True)
real_state_df.shape
```

In [ ]:

```
real_state_df
```

2. Integrating the input datasets

Now, as all the data files of various formats have been read into dataframes, the data is now ready to be integrated into one according to the schema provided.

2.1. Find suburb for each of the property

A new column `suburb` has been added to the real_state data. This new column contains the suburb of each of the property which has been calculated using the shapefiles and the latitude and longitude of each of the property.

suburb is added to the real state data

In [ ]:

```
recs = sf.records()
shapes=sf.shapes()
s = shapes[0]
```

In [ ]:

```
Nshp=len(shapes)
```

In [ ]:

```
ptchs=[]
for nshp in range(Nshp):
    pts=np.array(shapes[nshp].points)
    prt=shapes[nshp].parts
    par=list(prt)+[pts.shape[0]]

    for pij in range(len(prt)):
        ptchs.append((recs[nshp][6], Polygon(pts[par[pij]:par[pij+1]])))
```

In [ ]:

```
subs = []
for index,row in real_state_df.iterrows():
    for i in range(len(ptchs)):
        if ptchs[i][1].contains_point((float(row['lng']), float(row['lat']))):
            subs.append(ptchs[i][0])
```

In [ ]:

```
#append the values of the subs list to the column suburb of the real state dataframe
real_state_df['suburb'] = subs
```

In [ ]:

```
#apply the title case to the names of the suburbs
real_state_df['suburb'] = real_state_df['suburb'].apply(lambda x: x.title())
```

In [ ]:

```
real_state_df
```

2.2. The nearest shopping_center and its distance is computed

In this section, the distance to the nearest shopping center and the id of the nearest shopping center is computed.

Firstly, we calculate the haversine distance using the haversine function, this function calculates the distance between two points for which the latitude and the longitude is known. The radius of the earth is taken as 6378 km.

A new column $dists$ is made which stores a list of distances between a property and each shopping center for all the properties. For each property, from the list of distances, the minimum distance is placed in the `Distance_to_sc` column and the corresponding index is placed in the sc_id column. Now that the index for the nearest shopping center is found, the id of the shopping center is then placed in the `Shopping_center_id` column.

The extra columns made for some calculations are then removed.

In [ ]:

```
#calculate distance using latitude and longitude
def haversine(lat1, lon1, lat2, lon2):

    R = 6378 #radius of earth

    dLat = radians(lat2 - lat1)
    dLon = radians(lon2 - lon1)
    lat1 = radians(lat1)
    lat2 = radians(lat2)

    a = sin(dLat/2)**2 + cos(lat1)*cos(lat2)*sin(dLon/2)**2
    c = 2*asin(sqrt(a))

    return R * c *1000 #to get in metres
```

In [ ]:

```
ss=[] #list of distances from all the shopping centers for that property
dists=[] #list of distances from all the shopping centers for all the properties
for i in range(len(real_state_df.lat)):
    ss=[]
    for j in range(len(shopingcenters_df.lat)):
        ss.append(haversine(float(real_state_df.lat[i]), float(real_state_df.lng[i]), f
loat(shopingcenters_df.lat[j]), float(shopingcenters_df.lng[j])))
    dists.append(ss)
```

In [ ]:

```
real_state_df['dists']=dists
```

In [ ]:

```
def minimum(a):

    # inbuilt function to find the position of minimum
    minpos = a.index(min(a))
    return minpos
```

In [ ]:

```
x=[]
y=[]
for i in range(len(real_state_df.dists)):
    cc=min(real_state_df.dists[i])
    x.append(int(cc))
    y.append((minimum(real_state_df.dists[i])))
```

In [ ]:

```
real_state_df['Distance_to_sc']=x
real_state_df['sc_id']=y
```

In [ ]:

```
xx=[] #stores the index of the shopping center
for i in range(len(real_state_df.sc_id)):
    xx.append(real_state_df.sc_id[i])
```

In [ ]:

```
xc=[]#stores the id of the shopping center
for i in range(len(xx)):
    xc.append(shopingcenters_df.sc_id[xx[i]])
```

In [ ]:

```
real_state_df['Shopping_center_id']=xc
```

In [ ]:

```
#remove the extra columns
del real_state_df['dists']
del real_state_df['sc_id']
```

In [ ]:

```
real_state_df
```

2.2. The nearest hospital and its distance is computed

In this section, the distance to the nearest hospital and the id of the nearest hospital is computed.

Firstly, we calculate the haversine distance using the haversine function, this function calculates the distance between two points for which the latitude and the longitude is known. The radius of the earth is taken as 6378 km.

A new column hospital_dists is made which stores a list of distances between a property and each hospital for all the properties. For each property, from the list of distances, the minimum distance is placed in the `Distance_to_hospital` column and the corresponding index is placed in the hosp_id column. Now that the index for the nearest hospital is found, the id of the hospital is then placed in the `Hospital_id` column.

The extra columns made for some calculations are then removed.

In [ ]:

```
ss=[] #list of distances from all the hospitals for that property
hosp_distances=[] #list of distances from all the hospitals for all the properties
for i in range(len(real_state_df.lat)):
    ss=[]
    for j in range(len(pdf_df.id)):

        ss.append(haversine(float(real_state_df.lat[i]),float(real_state_df.lng[i]), fl
oat(pdf_df.lat[j]), float(pdf_df.lng[j])))
    hosp_distances.append(ss)

real_state_df['hospital_dists'] = hosp_distances
```

In [ ]:

```
def minimum(a):

    # inbuilt function to find the position of minimum
    minpos = a.index(min(a))
    return minpos
```

In [ ]:

```
x=[]
y=[]
for i in range(len(real_state_df.hospital_dists)):
    cc=min(real_state_df.hospital_dists[i])
    x.append(int(cc))
    y.append((minimum(real_state_df.hospital_dists[i])))

real_state_df['Distance_to_hospital']=x
real_state_df['hosp_id']=y
```

In [ ]:

```
xx=[] # stores the index for that hospital
for i in range(len(real_state_df.hosp_id)):
    xx.append(real_state_df.hosp_id[i])

xc=[] # stores the id of the hospital
for i in range(len(xx)):
    xc.append(pdf_df.id[xx[i]])

real_state_df['Hospital_id']=xc
```

In [ ]:

```
#remove the extra columns
del real_state_df['hospital_dists']
del real_state_df['hosp_id']
```

In [ ]:

```
real_state_df
```

2.3. The nearest supermarket and its distance is computed

In this section, the distance to the nearest supermarket and the id of the nearest supermarket is computed.

Firstly, we calculate the haversine distance using the haversine function, this function calculates the distance between two points for which the latitude and the longitude is known. The radius of the earth is taken as 6378 km.

A new column html_dists is made which stores a list of distances between a property and each supermarket for all the properties. For each property, from the list of distances, the minimum distance is placed in the `Distance_to_supermarket` column and the corresponding index is placed in the html_dists_point column. Now that the index for the nearest supermarket is found, the id of the supermarket is then placed in the `Supermarket_id` column.

The extra columns made for some calculations are then removed.

In [ ]:

```
ss=[]  #list of distances from all the supermarkets for that property
html_distances=[]  #list of distances from all the hospitals for all the properties
for i in range(len(real_state_df.lat)):
    ss=[]
    for j in range(len(html_df.lat)):

        ss.append(haversine(float(real_state_df.lat[i]),float(real_state_df.lng[i]), fl
oat(html_df.lat[j]), float(html_df.lng[j])))
    html_distances.append(ss)
```

In [ ]:

```
real_state_df['html_dists']=html_distances
```

In [ ]:

```
def minimum(a):

    # inbuilt function to find the position of minimum
    minpos = a.index(min(a))
    return minpos

x=[]
y=[]
for i in range(len(real_state_df.html_dists)):
    cc=min(real_state_df.html_dists[i])
    x.append(int(cc))
    y.append((minimum(real_state_df.html_dists[i]))+1)

real_state_df['Distance_to_supermarket']=x
real_state_df['html_dists_point']=y
```

In [ ]:

```
xx=[] #stores the index of the supermarket
for i in range(len(real_state_df.html_dists_point)):
    xx.append(real_state_df.html_dists_point[i])

xc=[] #stores the id of the supermarket
for i in range(len(xx)):
    xc.append(html_df.id[xx[i]])

real_state_df['Supermarket_id']=xc
```

In [ ]:

```
#remove the extra columns
del real_state_df['html_dists']
del real_state_df['html_dists_point']
```

In [ ]:

```
real_state_df
```

2.4. Calculating the nearest train_station and its distance

In this section, the distance to the nearest train station and the id of the nearest supermarket is computed.

Firstly, we load the train information data using the pandas library. We calculate the haversine distance using the haversine function, this function calculates the distance between two points for which the latitude and the longitude is known. The radius of the earth is taken as 6378 km.

A new column train_dists is made which stores a list of distances between a property and each train station for all the properties. For each property, from the list of distances, the minimum distance is placed in the `Distance_to_train_station` column and the corresponding index is placed in the train_dists_point column. Now that the index for the nearest station is found, the id of the train station is then placed in the `Train_station_id` column.

The extra columns made for some calculations are then removed.

In [ ]:

```
#read the file
train_info = pd.read_csv("./GTFS_Melbourne_Train_Information/1. GTFS - Melbourne Train
 Information - From PTV (9 Oct 2015)/GTFS - Melbourne Train Information/stops.txt", sep
= ",")
```

In [ ]:

```
train_info
```

In [ ]:

```
ss=[] #list of distances from all the train_stations for that property
train_distances=[]#list of distances from all the train_stations for all the properties
for i in range(len(real_state_df.lat)):
    ss=[]
    for j in range(len(train_info.stop_lat)):

        ss.append(haversine(float(real_state_df.lat[i]),float(real_state_df.lng[i]), fl
oat(train_info.stop_lat[j]), float(train_info.stop_lon[j])))
    train_distances.append(ss)

real_state_df['train_dists']=train_distances
```

In [ ]:

```
def minimum(a):

    # inbuilt function to find the position of minimum
    minpos = a.index(min(a))
    return minpos
```

In [ ]:

```
x=[]
y=[]
for i in range(len(real_state_df.train_dists)):
    cc=min(real_state_df.train_dists[i])
    x.append(int(cc))
    y.append((minimum(real_state_df.train_dists[i])))

real_state_df['Distance_to_train_station']=x
real_state_df['train_dists_point']=y
```

In [ ]:

```
real_state_df
```

In [ ]:

```
xx=[] #stores the index of the train station
for i in range(len(real_state_df.train_dists_point)):
    xx.append(real_state_df.train_dists_point[i])
```

In [ ]:

```
xc=[] #stores the id of the train station
for i in range(len(xx)):
    xc.append(train_info.stop_id[xx[i]])

real_state_df['Train_station_id']=xc
```

In [ ]:

```
#remove unwanted columns
del real_state_df['train_dists']
del real_state_df['train_dists_point']
```

In [ ]:

```
real_state_df
```

2.5. Calculating the travel_min_to_CBD and Transfer_flag

In this section, the average travel time (minutes) from the closest train station to the 'Flinders street' station on weekdays (i.e. Monday-Friday) departing between 7 to 9 am is computed which is the travel_min_to_CBD and Transer_flag is a boolean attribute indicating whether there is a direct trip to the Flinders street station from the closest station between 7-9am on the weekdays. This flag is 0 if there is a direct trip (i.e. no transfer between trains is required to get from the closest train station to the Flinders station) and 1 otherwise.

2.5.1. Read the GTFS_Melbourne_Train_Information data

Firstly, we load the train information data using the pandas library from the provided GTFS_Melbourne_Train_Information. This folder contains various files in csv format namely routes.csv, stops.csv, agency.csv, calendar.csv, calendar_dates.csv, shapes.csv, stop_times.csv and trips.csv.

In [ ]:

```
routes = pd.read_csv("./GTFS_Melbourne_Train_Information/1. GTFS - Melbourne Train Info
rmation - From PTV (9 Oct 2015)/GTFS - Melbourne Train Information/routes.txt", sep =
",")
stops = pd.read_csv("./GTFS_Melbourne_Train_Information/1. GTFS - Melbourne Train Infor
mation - From PTV (9 Oct 2015)/GTFS - Melbourne Train Information/stops.txt", sep = ","
)
agency = pd.read_csv("./GTFS_Melbourne_Train_Information/1. GTFS - Melbourne Train Info
rmation - From PTV (9 Oct 2015)/GTFS - Melbourne Train Information/agency.txt", sep =
",")
calendar = pd.read_csv("./GTFS_Melbourne_Train_Information/1. GTFS - Melbourne Train In
formation - From PTV (9 Oct 2015)/GTFS - Melbourne Train Information/calendar.txt", sep
= ",")
calendar_dates = pd.read_csv("./GTFS_Melbourne_Train_Information/1. GTFS - Melbourne Tr
ain Information - From PTV (9 Oct 2015)/GTFS - Melbourne Train Information/calendar_dat
es.txt", sep = ",")
shapes = pd.read_csv("./GTFS_Melbourne_Train_Information/1. GTFS - Melbourne Train Info
rmation - From PTV (9 Oct 2015)/GTFS - Melbourne Train Information/shapes.txt", sep =
",")
stop_times = pd.read_csv("./GTFS_Melbourne_Train_Information/1. GTFS - Melbourne Train
 Information - From PTV (9 Oct 2015)/GTFS - Melbourne Train Information/stop_times.txt"
, sep = ",")
trips = pd.read_csv("./GTFS_Melbourne_Train_Information/1. GTFS - Melbourne Train Infor
mation - From PTV (9 Oct 2015)/GTFS - Melbourne Train Information/trips.txt", sep = ","
)
```

We calculate the haversine distance using the haversine function, this function calculates the distance between two points for which the latitude and the longitude is known. The radius of the earth is taken as 6378 km.

A new column train_dists is made which stores a list of distances between a property and each train station for all the properties. For each property, from the list of distances, the minimum distance is placed in the `Distance_to_train_station` column and the corresponding index is placed in the train_dists_point column. Now that the index for the nearest station is found, the id of the train station is then placed in the `Train_station_id` column.

The extra columns made for some calculations are then removed.

2.5.2. Merge the necessary data

In order to retrieve important information, the data should be in a proper format, in this case the data which has to be used to calculate the travel_min_to_CBD and the Transfer_flag is in three separate files i.e. stops, stop_times and the trips data. So, the three dataframes are merged into one dataframe.

In [ ]:

```
stops_merged = pd.merge(stops, stop_times, on='stop_id')
```

In [ ]:

```
stops_times_trips = pd.merge(stops_merged, trips, on='trip_id')
```

In [ ]:

```
full_df=stops_times_trips
```

2.5.3. Extract the services running on all weekdays

Here, the services which run on all weekdays i.e. Monday to Friday are found from the calendar data which contains information about the service_id, the days on which it runs, start_date and end_date. As we only need the services that run on all weekdays, the services running on monday, tuesday, wednesday, thursday and friday are checked.

Then, the data of the services running on weekdays is filtered out from the main merged train information dataframe.

In [ ]:

```
service_id = []
for index,row in calendar.iterrows():
    if (row['monday'] == 1 & row['tuesday'] == 1 & row['wednesday'] == 1 & row['thursda
y'] == 1 & row['friday'] == 1):
        service_id.append(row['service_id'])
```

In [ ]:

```
stops_times_trips = stops_times_trips[stops_times_trips['service_id'].isin(service_id)]
```

2.5.4. Filter out the departure time and the stop id

In this section, we filter out those trips that have departed between 7 am and 9 am so a condition of the departure time has been applied in order to get the correct results. The rows with stop id of Flinders station is only needed and thus extracted from the dataframe. The rows having stop sequence as 1 are also removed as it means that it is the start station.

Only the rows with the stop id of Flinders station are required and thus filtered from the data.

In [ ]:

```
stops_times_trips  = stops_times_trips[(stops_times_trips['departure_time'] >= '07:00:0
0' ) & (stops_times_trips['departure_time'] <= '09:00:00')]
stops_times_trips = stops_times_trips[(stops_times_trips['stop_sequence'])!=1]
```

In [ ]:

```
train_trips_flinders = stops_times_trips[(stops_times_trips['stop_id'] == 19854)]
flinders_trip_id = train_trips_flinders['trip_id'].tolist() #getting trip id's which ha
ve flinders as stop
len(flinders_trip_id)
```

In [ ]:

```
stops_times_trips = stops_times_trips[stops_times_trips['trip_id'].isin(flinders_trip_i
d)]
stops_times_trips
```

2.5.5. Filter the conditions from the main merged dataframe

In order to get all the rows that fullfil the conditions, the main merged dataset (stops, trips and stop_times) is filtered based on the rows that have departure between 7 am to 9am and those with destination as Flinders street station and those services that run on all weekdays.

In [ ]:

```
full_df = full_df[full_df['trip_id'].isin(flinders_trip_id)] # dataframe which has flin
ders station as stop
```

In [ ]:

```
full_df
```

2.5.6. Calculate Transfer_flag column

Transer_flag is a boolean attribute indicating whether there is a direct trip to the Flinders street station from the closest station between 7-9am on the weekdays. This flag is 0 if there is a direct trip (i.e. no transfer between trains is required to get from the closest train station to the Flinders station) and 1 otherwise. The stop_id of the rows fulfilling all the conditions are extracted and stored in a list and then the unique elements of this list having the stop id's have been found using the set function.

If the stop_id in this list is present in the real_state data's Train station id then the transfer_flag will be zero which means there is a direct train to Flinders from that station otherwise it will be 1 which means there is no direct train from that station to Flinders.

In [ ]:

```
stop_id=full_df['stop_id'].tolist()
```

In [ ]:

```
unique_stop_id=list(set(stop_id))
len(unique_stop_id)
```

In [ ]:

```
real_state_df['Transfer_flag']=0
```

In [ ]:

```
for i in range(len(real_state_df.Train_station_id)):
    if(real_state_df.Train_station_id[i] in unique_stop_id):
        real_state_df['Transfer_flag'][i]=0
    else:
        real_state_df['Transfer_flag'][i]=1
```

In [ ]:

```
real_state_df
```

2.5.7. Compute the travel_min_to_CBD column

It is the average travel time (minutes) from the closest train station to the “Flinders street” station on weekdays (i.e. Monday-Friday) departing between 7 to 9 am. For example, if there are 3 trip departing from the closest train station to the Flinders street station on weekdays between 7-9am and each take 6, 7, and 8 minutes respectively, then the value of this column for the property should be (6+7+8)/3. If there are any direct transfers between the closest station and Flinders street station, only the average of direct transfers should be calculated.

In [ ]:

```
flinders_stop_time = []
for stopid in unique_stop_id:

    trips_list = full_df[full_df['stop_id']== stopid]['trip_id'].tolist()  #trip id's f
or given stop id
    travel_times = []

    for trip in trips_list:
        flinders_stop = full_df[full_df['trip_id']==trip]  #extracting only data for se
lected trip ID's
        start_stopid = flinders_stop[flinders_stop['stop_id']== stopid] # data for the
 given source stopid
        flinders_stopid = flinders_stop[(flinders_stop['stop_id'] == 19854)] # for flin
ders street station
        departure_time = start_stopid['departure_time'].values # departure time from to
given stopID
        arrival_time = flinders_stopid['arrival_time'].values # Arrival time at flinder
s station
        departure_time = pd.to_timedelta(departure_time[0]).seconds/60 #converting into
seconds
        arrival_time = pd.to_timedelta(arrival_time[0]).seconds/60
        time_taken = abs(arrival_time - departure_time)
        travel_times.append(time_taken)

    mean_time = int(np.mean(travel_times)) #mean of time taken
    flinders_stop_time.append((stopid, int(mean_time))) #appending tuple of stopid and
 meantime
flinders_stop_time_dict = dict(flinders_stop_time) # dictionary of stopid and meantime
```

In [ ]:

```
real_state_df['travel_min_to_CBD'] = real_state_df['Train_station_id'].apply(lambda sto
pid: flinders_stop_time_dict[stopid])
```

In [ ]:

```
real_state_df
```

2.6 Arrange the columns according to the specifications

Here, the columns are being arranged according to the specifications provided. This is to ensure that there is consistency in the output produced.

In [ ]:

```
# arranging the columns as asked in the specification of the task.
real_state_df = real_state_df[['property_id','lat','lng','addr_street','suburb','price'
,'property_type','year','bedrooms','bathrooms','parking_space','Shopping_center_id','Di
stance_to_sc','Train_station_id','Distance_to_train_station','travel_min_to_CBD','Trans
fer_flag','Hospital_id','Distance_to_hospital','Supermarket_id','Distance_to_supermarke
t']]
```

3. Converting the final dataframe into csv file

In this section, the final dataframe having all the required columns in a proper format is being written to a csv file named "30757924_A3_solution.csv".

In [ ]:

```
# converting the final dataframe into csv file for final submission.
real_state_df.to_csv('30757924_A3_solution.csv',index=False)
```

Task 2: Data Reshaping

1. Introduction

In this task, the effect of different normalization/transformation methods (i.e. standardization, minmax normalization, log, power, box-cox transformation) on the “price” , “Distance_to_sc”, “travel_min_to_CBD” , and “Distance_to_hospital” attributes are studied and observed to explain their effect assuming we want to develop a linear model to predict the “price” using “Distance_to_sc”, “travel_min_to_CBD”, and “Distance_to_hospital” attributes . The linear regression assumptions are to be studied in this task are: Normality and Linearity.

Data Normalisation:

It is usually the case that raw data is not in good shape to be processed by data mining techniques. Data normalisation is the process of transforming raw data values to another form with better properties that are more suitable for modelling and analysis.

The normalisation process focuses on scaling data in terms of range and distribution. Therefore, it consists of two main processes:

1. Scaling normalisation: where the focus is on rescaling data range to a specific interval/range. This type includes min-max normalisation and decimal scaling normalisation.
2. Standardisation (z-score normalisation): where the focus is on shifting the distribution of data to have mean of 0 and standard deviation of 1.

Both standardisation and normalisation have its own use cases where it is important to be applied for data preparation.

In this task, the main linear regression assumptions are Normality and Linearity.

2. Store the final dataframe in another dataframe

Here, the final dataframe of the real_state data is stored in another dataframe so as to retain the original one completely.

In [ ]:

```
#storing the final dataframe in another dataframe so as to retain the original one comp
letely.
df_for_reshaping = real_state_df
```

3. Normalisations/Transformations

As mentioned before, in this task, the effect of different normalization/transformation methods (i.e. standardization, minmax normalization, log, power, box-cox transformation) on the “price” , “Distance_to_sc”, “travel_min_to_CBD” , and “Distance_to_hospital” attributes are studied and observed to explain their effect assuming we want to develop a linear model to predict the “price” using “Distance_to_sc”, “travel_min_to_CBD”, and “Distance_to_hospital” attributes . The linear regression assumptions are to be studied in this task are: Normality and Linearity.

The normalisations performed are:

- Standardisation: The scikit-learn library is used for standardisation of the data and the main motive is to have mean 0 and standard deviation 1.
- Min - Max Normalisation : Min-max normalisation is a different type of normalization used for reshaping the range of data i.e. it transforms the actual original range to another range of the data.

The transformations performed are:

- Box Cox Transformation : A Box Cox transformation is a transformation of a non-normal dependent variables into a normal shape. For applying this transformation, the data should be positive.
- Square Power Transformation : In this transformation, the values are squared. The aim of this transformation was to stretch the values of the data.
- Log Transformation: The log transformation reduces all values, and values between 0 and 1 become negative. Large values are reduced much more than small values.

3.1. Transformations for `price`

3.1.1. Normalisations/ Transformations

In the next few blocks, the normalisations and transformations have been performed on the 'price' column.

In [ ]:

```
# Z-Score Normalisation
std_scale = preprocessing.StandardScaler().fit(df_for_reshaping[['price']])
df_std = std_scale.transform(df_for_reshaping[['price']])
df_std[0:5]
df_for_reshaping['price_z_normalised'] = df_std[:,0]
```

In [ ]:

```
# MinMax Normalisation
minmax_scale = preprocessing.MinMaxScaler().fit(df_for_reshaping[['price']])
df_minmax = minmax_scale.transform(df_for_reshaping[['price']])
df_for_reshaping['price_minmax_normalised'] = df_minmax[:,0]
```

In [ ]:

```
# Box Cox Transformation
import numpy as np
from scipy import stats
df_for_reshaping['price_box_cox_transformed'],_ = stats.boxcox(df_for_reshaping['price'
])
```

In [ ]:

```
# Square Power Transformation
df_for_reshaping['price_power_transformed'] = df_for_reshaping['price'].apply(lambda x:
x**2 )
```

In [ ]:

```
# Log Transformation
df_for_reshaping['price_log_transformed'] = np.log(df_for_reshaping.price)
```

3.1.2. Plot the histograms for the transformations performed

The transformations that have been applied are now shown visually with the help of boxplots.

In [ ]:

```
plt.figure(1,figsize=(15,12))
plt.subplot(3,2,1)
df_for_reshaping['price'].hist(bins=50)
plt.title('Price')
plt.grid(b=None)
plt.subplot(3,2,2)
df_for_reshaping['price_z_normalised'].hist(bins=50)
plt.title('price_z_normalised')
plt.grid(b=None)
plt.subplot(3,2,3)
df_for_reshaping['price_minmax_normalised'].hist(bins=50)
plt.title('price_minmax_normalised')
plt.grid(b=None)
plt.subplot(3,2,4)
df_for_reshaping['price_box_cox_transformed'].hist(bins=50)
plt.title('price_box_cox_transformed')
plt.grid(b=None)
plt.subplot(3,2,5)
df_for_reshaping['price_power_transformed'].hist(bins=50)
plt.title('price_power_transformed')
plt.grid(b=None)
plt.subplot(3,2,6)
df_for_reshaping['price_log_transformed'].hist(bins=50)
plt.title('price_log_transformed')
plt.grid(b=None)
```

3.1.3. Summary:

In this task, the transformations have been applied on the 'price' column and then the histograms have been plotted for the transformations. The graph for which the data is most linearly distributed is the 'price_log_transformed' which shows the log transformation on the column 'price'. Thus, this column will be used for the building up of the linear model.

3.2. Transformations for `Distance_to_sc`

3.2.1. Normalisations/ Transformations

In the next few blocks, the normalisations and transformations have been performed on the 'Distance_to_sc' column.

In [ ]:

```
# Z-Score Normalisation
std_scale = preprocessing.StandardScaler().fit(df_for_reshaping[['Distance_to_sc']])
df_std = std_scale.transform(df_for_reshaping[['Distance_to_sc']])
df_std[0:5]
df_for_reshaping['Distance_to_sc_z_normalised'] = df_std[:,0]
```

In [ ]:

```
# MinMax Normalisation
minmax_scale = preprocessing.MinMaxScaler().fit(df_for_reshaping[['Distance_to_sc']])
df_minmax = minmax_scale.transform(df_for_reshaping[['Distance_to_sc']])
df_for_reshaping['Distance_to_sc_minmax_normalised'] = df_minmax[:,0]
```

In [ ]:

```
# Box Cox Transformation
df_for_reshaping['Distance_to_sc_box_cox_transformed'],_ = stats.boxcox(df_for_reshapin
g['Distance_to_sc'])
```

In [ ]:

```
# Square Power Transformation
df_for_reshaping['Distance_to_sc_power_transformed'] = df_for_reshaping['Distance_to_s
c'].apply(lambda x: x**2 )
```

In [ ]:

```
# Log Transformation
df_for_reshaping['Distance_to_sc_log_transformed'] = np.log(df_for_reshaping['Distance_
to_sc'])
```

3.2.2. Plot the histograms for the transformations performed

The transformations that have been applied are now shown visually with the help of boxplots.

In [ ]:

```
plt.figure(1,figsize=(15,12))

plt.subplot(3,2,1)
df_for_reshaping['Distance_to_sc'].hist(bins=50)
plt.title('Distance_to_sc')
plt.grid(b=None)
plt.subplot(3,2,2)
df_for_reshaping['Distance_to_sc_z_normalised'].hist(bins=50)
plt.title('Distance_to_sc_z_normalised')
plt.grid(b=None)
plt.subplot(3,2,3)
df_for_reshaping['Distance_to_sc_minmax_normalised'].hist(bins=50)
plt.title('Distance_to_sc_minmax_normalised')
plt.grid(b=None)
plt.subplot(3,2,4)
df_for_reshaping['Distance_to_sc_box_cox_transformed'].hist(bins=50)
plt.title('Distance_to_sc_box_cox_transformed')
plt.grid(b=None)
plt.subplot(3,2,5)
df_for_reshaping['Distance_to_sc_power_transformed'].hist(bins=50)
plt.title('Distance_to_sc_power_transformed')
plt.grid(b=None)
plt.subplot(3,2,6)
df_for_reshaping['Distance_to_sc_log_transformed'].hist(bins=50)
plt.title('Distance_to_sc_log_transformed')
plt.grid(b=None)
```

3.2.3. Summary:

In this task, the transformations have been applied on the 'Distance_to_sc' column and then the histograms have been plotted for the transformations. The graph for which the data is most linearly distributed is the 'Distance_to_sc' which shows the original 'Distance_to_sc'column. Thus, this column will be used for the building up of the linear model.

3.3. Transformations for `travel_min_to_CBD`

3.3.1. Normalisations/ Transformations

In the next few blocks, the normalisations and transformations have been performed on the 'travel_min_to_CBD' column.

In [ ]:

```
# Z-Score Normalisation
std_scale = preprocessing.StandardScaler().fit(df_for_reshaping[['travel_min_to_CBD']])
df_std = std_scale.transform(df_for_reshaping[['travel_min_to_CBD']])
df_std[0:5]
df_for_reshaping['travel_min_to_CBD_z_normalised'] = df_std[:,0]
```

In [ ]:

```
# MinMax Normalisation
minmax_scale = preprocessing.MinMaxScaler().fit(df_for_reshaping[['travel_min_to_CBD'
]])
df_minmax = minmax_scale.transform(df_for_reshaping[['travel_min_to_CBD']])
df_for_reshaping['travel_min_to_CBD_minmax_normalised'] = df_minmax[:,0]
```

In [ ]:

```
df_for_reshaping['travel_min_to_CBD'] = df_for_reshaping['travel_min_to_CBD'].apply(lam
bda x : 0.0001 if x == 0 else x)
#df_for_reshaping['travel_min_to_CBD'] = df_for_reshaping['travel_min_to_CBD'].apply(la
mbda x : 0.0001 if x < 0 else x)
```

In [ ]:

```
# Box Cox Transformation
import numpy as np
from scipy import stats
df_for_reshaping['travel_min_to_CBD_box_cox_transformed'],_ = stats.boxcox(df_for_resha
ping['travel_min_to_CBD'])
```

In [ ]:

```
# Square Power Transformation
df_for_reshaping['travel_min_to_CBD_power_transformed'] = df_for_reshaping['travel_min_
to_CBD'].apply(lambda x: x**2 )
```

In [ ]:

```
# Log Transformation
df_for_reshaping['travel_min_to_CBD_log_transformed'] = np.log(df_for_reshaping['travel
_min_to_CBD'])
```

3.3.2. Plot the histograms for the transformations performed

The transformations that have been applied are now shown visually with the help of boxplots.

In [ ]:

```
plt.figure(1,figsize=(15,12))
plt.subplot(3,2,1)
df_for_reshaping['travel_min_to_CBD'].hist(bins=50)
plt.title('travel_min_to_CBD')
plt.grid(b=None)
plt.subplot(3,2,2)
df_for_reshaping['travel_min_to_CBD_z_normalised'].hist(bins=50)
plt.title('travel_min_to_CBD_z_normalised')
plt.grid(b=None)
plt.subplot(3,2,3)
df_for_reshaping['travel_min_to_CBD_minmax_normalised'].hist(bins=50)
plt.title('travel_min_to_CBD_minmax_normalised')
plt.grid(b=None)
plt.subplot(3,2,4)
df_for_reshaping['travel_min_to_CBD_box_cox_transformed'].hist(bins=50)
plt.title('travel_min_to_CBD_box_cox_transformed')
plt.grid(b=None)
plt.subplot(3,2,5)
df_for_reshaping['travel_min_to_CBD_power_transformed'].hist(bins=50)
plt.title('travel_min_to_CBD_power_transformed')
plt.grid(b=None)
plt.subplot(3,2,6)
df_for_reshaping['travel_min_to_CBD_log_transformed'].hist(bins=50)
plt.title('travel_min_to_CBD_log_transformed')
plt.grid(b=None)
```

3.3.3. Summary:

In this task, the transformations have been applied on the 'travel_min_to_CBD' column and then the histograms have been plotted for the transformations. The graph for which the data is most linearly distributed is the 'travel_min_to_CBD_power_transformed' which shows the power transformation on the 'travel_min_to_CBD' column. Thus, this column will be used for the building up of the linear model.

3.4. Transformation for `Distance_to_hospital`

3.4.1. Normalisations/ Transformations

In the next few blocks, the normalisations and transformations have been performed on the 'Distance_to_hospital' column.

In [ ]:

```
# Z-Score Normalisation
std_scale = preprocessing.StandardScaler().fit(df_for_reshaping[['Distance_to_hospital'
]])
df_std = std_scale.transform(df_for_reshaping[['Distance_to_hospital']])
df_std[0:5]
df_for_reshaping['Distance_to_hospital_z_normalised'] = df_std[:,0]
```

In [ ]:

```
# MinMax Normalisation
minmax_scale = preprocessing.MinMaxScaler().fit(df_for_reshaping[['Distance_to_hospita
l']])
df_minmax = minmax_scale.transform(df_for_reshaping[['Distance_to_hospital']])
df_for_reshaping['Distance_to_hospital_minmax_normalised'] = df_minmax[:,0]
```

In [ ]:

```
# Box Cox Transformation
df_for_reshaping['Distance_to_hospital_box_cox_transformed'],_ = stats.boxcox(df_for_re
shaping['Distance_to_hospital'])
```

In [ ]:

```
# Square Power Transformation
df_for_reshaping['Distance_to_hospital_power_transformed'] = df_for_reshaping['Distance
_to_hospital'].apply(lambda x: x**2 )
```

In [ ]:

```
# Log Transformation
df_for_reshaping['Distance_to_hospital_log_transformed'] = np.log(df_for_reshaping['Dis
tance_to_hospital'])
```

3.4.2. Plot the histograms for the transformations performed

The transformations that have been applied are now shown visually with the help of boxplots.

In [ ]:

```
plt.figure(1,figsize=(15,12))
plt.subplot(3,2,1)
df_for_reshaping['Distance_to_hospital'].hist(bins=50)
plt.title('Distance_to_hospital')
plt.grid(b=None)
plt.subplot(3,2,2)
df_for_reshaping['Distance_to_hospital_z_normalised'].hist(bins=50)
plt.title('Distance_to_hospital_z_normalised')
plt.grid(b=None)
plt.subplot(3,2,3)
df_for_reshaping['Distance_to_hospital_minmax_normalised'].hist(bins=50)
plt.title('Distance_to_hospital_minmax_normalised')
plt.grid(b=None)
plt.subplot(3,2,4)
df_for_reshaping['Distance_to_hospital_box_cox_transformed'].hist(bins=50)
plt.title('Distance_to_hospital_box_cox_transformed')
plt.grid(b=None)
plt.subplot(3,2,5)
df_for_reshaping['Distance_to_hospital_power_transformed'].hist(bins=50)
plt.title('Distance_to_hospital_power_transformed')
plt.grid(b=None)
plt.subplot(3,2,6)
df_for_reshaping['Distance_to_hospital_log_transformed'].hist(bins=50)
plt.title('Distance_to_hospital_log_transformed')
plt.grid(b=None)
```

3.4.3. Summary:

In this task, the transformations have been applied on the 'Distance_to_hospital' column and then the histograms have been plotted for the transformations. The graph for which the data is most linearly distributed is the 'Distance_to_hospital_box_cox_transformed' which shows the box cox transformation on the 'Distance_to_hospital' column. Thus, this column will be used for the building up of the linear model.

4. Scaling

As the columns selected for linear modelling are not having the same range of values, hence scaling is performed. Scaling is done for those columns which are not already transformed and here it can be seen that the 'Distance_to_sc' column ranges from approximately 200 to 6000. So, in this case the 'Distance_to_sc' column has been scaled down to a range of 1 to 20.

4.1. Filter the required columns

The columns that have been selected for the linear models are filtered from the dataframe and stored in a new dataframe called 'df_filtered'. This dataframe contains the 'price_log_transformed', 'Distance_to_sc', 'travel_min_to_CBD_power_transformed', 'Distance_to_hospital_box_cox_transformed' columns.

In [ ]:

```
df_filtered = df_for_reshaping.loc[:,['price_log_transformed','Distance_to_sc', 'travel
_min_to_CBD_power_transformed',  'Distance_to_hospital_box_cox_transformed']]
```

4.2. Scale the 'Distance_to_sc' column

The 'Distance_to_sc' column has been scaled down in order to get it in the range of 1 to 20.

In [ ]:

```
# scaling Distance_to_sc column
scaler = preprocessing.MinMaxScaler(feature_range=(1, 20))
df_filtered['Distance_to_sc'] = scaler.fit_transform(df_filtered[['Distance_to_sc']])
```

In [ ]:

```
# plot to show the scaling for 'Distance_to_sc' column
df_for_reshaping["Distance_to_sc"].plot(), df_filtered["Distance_to_sc"].plot()
```

5. Checking for Linearity

5.1. Plotting of scatterplots

In order to show whether there is any linear relationship between the columns, scatterplots have been used in order to identify the realtionship between the variables. Here, 'Distance_to_sc', 'travel_min_to_CBD_power_transformed', and 'Distance_to_hospital_box_cox_transformed' with 'price_log_transformed' have been plotted with the help of scatterplots to see the linear relationship.

In [ ]:

```
# scatter plot between 'travel_min_to_CBD' and 'price_log_transformed'
x = np.array(df_filtered['travel_min_to_CBD_power_transformed'])
y = np.array(df_filtered['price_log_transformed'])
plt.plot(x, y, 'o')
slope, intercept = np.polyfit(x, y, 1)
plt.plot(x, slope*x + intercept)
```

In [ ]:

```
x = np.array(df_filtered['Distance_to_sc'])
y = np.array(df_filtered['price_log_transformed'])
plt.plot(x, y, 'o')
slope, intercept = np.polyfit(x, y, 1)
plt.plot(x, slope*x + intercept)
```

In [ ]:

```
x = np.array(df_for_reshaping['Distance_to_hospital_box_cox_transformed'])
y = np.array(df_for_reshaping['price_log_transformed'])
plt.plot(x, y, 'o')
slope, intercept = np.polyfit(x, y, 1)
plt.plot(x, slope*x + intercept)
```

5.2. Correlation matrix

As from the above scatterplots, there is not much clarity whether there is any linear relationship between 'Distance_to_sc', 'travel_min_to_CBD_power_transformed', and 'Distance_to_hospital_box_cox_transformed' with 'price_log_transformed'. So, the correlation matrix has been used to check for the linear relationship between these variables.

5.2.1. Filter the required columns

The columns that have been selected for the linear models are filtered from the dataframe and stored in a new dataframe called 'df_linearity'. This dataframe contains the 'price_log_transformed', 'Distance_to_sc', 'travel_min_to_CBD_power_transformed', 'Distance_to_hospital_box_cox_transformed' columns.

In [ ]:

```
df_linearity = df_for_reshaping.loc[:,['price_log_transformed','Distance_to_sc', 'trave
l_min_to_CBD_power_transformed',  'Distance_to_hospital_box_cox_transformed']]
```

5.2.2. Correlation matrix for the df_linearity dataframe

A correlation matrix has been made to check for linearity between the variables.

In [ ]:

```
df_linearity.corr()
```

5.2.3. Filter the required columns

The columns that have been selected for the linear models are filtered from the dataframe and stored in a new dataframe called 'org_values'. This dataframe contains the 'price', 'Distance_to_sc', 'travel_min_to_CBD', 'Distance_to_hospital' columns.

In [ ]:

```
# creating a new dataframe from the original dataframe that contains original values wi
thout transformation.
org_values = df_for_reshaping.loc[:,['price','Distance_to_sc', 'travel_min_to_CBD',  'D
istance_to_hospital']]
```

5.2.4. Correlation matrix for the org_values dataframe

A correlation matrix has been made to check for linearity between the variables.

In [ ]:

```
org_values.corr()
```

5.3. Explanation:

From the above correlation matrices, on comparing the results it is found that after the transformation of the columns, the relation between the 'price' columnand the other columns namely 'Distance_to_sc', 'travel_min_to_CBD', and 'Distance_to_hospital' has improved as for Distance_to_sc the value changed from -0.115261 to -0.119456; for travel_min_to_CBD the value changed from -0.359875 to -0.360352 and for Distance_to_hospital the value changed from -0.312859 to -0.347233 with respect to the price.

Now, it can be said that the linear relationship between the variables has increased.

6. Checking for Normality

In order to show for normality between the variables, the multiple regression assumes that the residuals are normally distributed. So, for checking normality, a linear model is trained using the 'price_log_transformed','Distance_to_sc', 'travel_min_to_CBD_power_transformed' and 'Distance_to_hospital_box_cox_transformed' variables of the dataframe.

6.1. Filter the required columns

The columns that have been selected for the linear model are filtered from the dataframe and stored in a new dataframe called 'df_normality'. This dataframe contains the 'price_log_transformed', 'Distance_to_sc', 'travel_min_to_CBD_power_transformed', 'Distance_to_hospital_box_cox_transformed' columns.

In [ ]:

```
df_normality = df_for_reshaping.loc[:,['price_log_transformed','Distance_to_sc', 'trave
l_min_to_CBD_power_transformed',  'Distance_to_hospital_box_cox_transformed']]
```

6.2. Linear models

Linear models have been fitted on this filtered dataframe with the 'price' as the value to be predicted and 'Distance_to_sc', 'travel_min_to_CBD', 'Distance_to_hospital' as the predictor variables and also on the original data values of 'price', 'Distance_to_sc', 'travel_min_to_CBD' and 'Distance_to_hospital'.

Then using these models, the predictions of the predicted value i.e. the price are made and then a histogram has been created for plotting the residuals for both the models.

According to the plotted histograms, it is very evident that the plot for the transformed variables has a better normal distribution than the one with the original values without any form of transformation. This shows that the dataframe with the transformed variables follows the normality and is better for making any type of predictions in this case.

In [ ]:

```
# training a linear model transformed variables dataframe
model = LinearRegression()
model.fit(df_normality.iloc[:,1:],df_normality.iloc[:,0])
```

In [ ]:

```
# making predictions using the model
predictions = model.predict(df_normality.iloc[:,1:])
```

In [ ]:

```
# plotting the histogram of the residuals to check for normality
plt.hist(predictions - df_normality.iloc[:,0])
```

In [ ]:

```
# training a linear model variables without transformation dataframe
model = LinearRegression()
model.fit(org_values.iloc[:,1:],org_values.iloc[:,0])
```

In [ ]:

```
# making predictions using the model developed in the above cell
predictions = model.predict(org_values.iloc[:,1:])
```

In [ ]:

```
# plotting the histogram of the residuals to check for Normality
plt.hist(predictions - org_values.iloc[:,0])
```

Conclusion:

Task 1:

In Task 1, the data from different file formats is to be integrated. The different formats that are provided are pdf, xml, json, excel file and html format file. All different types of files are read in different ways into a dataframe so that later they can be merged according to the specifications provided. The shapefiles are read in order to find the exact suburb of the addresses. The haversine distance is used to get distances based on latitude and longitude.

Task 2:

In Task 2, some normalisations and transformations have been performed on some of the variables. The task included studying the effects of linearity and normality of linear regression by the models. Using the histogram, the variables following the normal distribution are chosen for the linear model. Using the correlation matrix, the linearity between the variables is shown. The normality is shown by plotting the residuals. For normality, the residuals should be normally distributed and from the graph of the transformed variables, it can be clearly seen that the model with the transfromed variables perform better than the ones with the original variables.

References:

1. Harris, C. (2020). fitting data with numpy. Retrieved 18 November 2020, from https://stackoverflow.com/questions/18767523/fitting-data-with-numpy (https://stackoverflow.com/questions/18767523/fitting-data-with-numpy)
2. Box-Cox Transformation using Python - GeeksforGeeks. (2020). Retrieved 18 November 2020, from https://www.geeksforgeeks.org/box-cox-transformation-using-python/ (https://www.geeksforgeeks.org/box-cox-transformation-using-python/)
3. Week 11 - Lecture Slides. Retrieved from https://d3cgwrxphz0fqu.cloudfront.net/e6/92/e692e400132b29a70ced037902e0b2f113df1f21?response-content-disposition=inline%3Bfilename%3D%225.-Data-Reshaping.pdf%22&response-content-type=application%2Fpdf&Expires=1605693040&Signature=KWf1G6CjoaTKPkTN2uSrJwPRWsZxaK3XV1A76ssUFTuc87DI5wXPvXYvhdsyjuRwJkNAtFe5FwSDEjrqV5ykLcKXWuB9S23G~cwipy34A3ACM6bPWYvy9R2PSsw90iGTaZJKVOS6laVQZhL1S-OFi75lT9Y8~lbG8kqkdf56pIet7yg1bXf06agQ__&Key-Pair-Id=APKAJRIEZFHR4FGFTJHA (https://d3cgwrxphz0fqu.cloudfront.net/e6/92/e692e400132b29a70ced037902e0b2f113df1f21?response-content-disposition=inline%3Bfilename%3D%225.-Data-Reshaping.pdf%22&response-content-type=application%2Fpdf&Expires=1605693040&Signature=KWf1G6CjoaTKPkTN2uSrJwPRWsZxaK3XV1A76ssUFTuc87DI5wXPvXYvhdsyjuRwJkNAtFe5FwSDEjrqV5ykLcKXWuB9S23G~cwipy34A3ACM6bPWYvy9R2PSsw90iGTaZJKVOS6laVQZhL1S-OFi75lT9Y8~lbG8kqkdf56pIet7yg1bXf06agQ__&Key-Pair-Id=APKAJRIEZFHR4FGFTJHA)
4. (2020). Retrieved 18 November 2020, from http://downloads.cambridge.edu.au/education/extra/209/PageProofs/Further%20Maths%20TINCP/Core%2 (http://downloads.cambridge.edu.au/education/extra/209/PageProofs/Further%20Maths%20TINCP/Core%
5. Prakash, J., & Kaushik, N. (2020). How to convert an XML file to nice pandas dataframe?. Retrieved 18 November 2020, from https://stackoverflow.com/questions/28259301/how-to-convert-an-xml-file-to-nice-pandas-dataframe (https://stackoverflow.com/questions/28259301/how-to-convert-an-xml-file-to-nice-pandas-dataframe)
6. Junior, I., & Han, M. (2020). Opening a pdf and reading in tables with python pandas. Retrieved 18 November 2020, from https://stackoverflow.com/questions/23284759/opening-a-pdf-and-reading-in-tables-with-python-pandas (https://stackoverflow.com/questions/23284759/opening-a-pdf-and-reading-in-tables-with-python-pandas)
7. Here's How To Calculate Distance Between 2 Geolocations in Python. (2020). Retrieved 18 October 2020, from https://towardsdatascience.com/heres-how-to-calculate-distance-between-2-geolocations-in-python93ecab5bbba4 (https://towardsdatascience.com/heres-how-to-calculate-distance-between-2-geolocations-in-python93ecab5bbba4) (https://towardsdatascience.com/heres-how-to-calculate-distance-between-2-geolocationsin-python-93ecab5bbba4 (https://towardsdatascience.com/heres-how-to-calculate-distance-between-2-geolocationsin-python-93ecab5bbba4))
8. Haversine formula. (2020). Retrieved 18 October 2020, from https://en.wikipedia.org/wiki/Haversine_formula (https://en.wikipedia.org/wiki/Haversine_formula) (https://en.wikipedia.org/wiki/Haversine_formula (https://en.wikipedia.org/wiki/Haversine_formula))