HDB Resale Price Predictor & Visualisation

This project aims to create a data pipeline with the help of availale APIs (Data.gov.sg and OneMap) to build a web-based application for

1. HDB Price visualisation
2. HDB Price prediction

The prototype aims to read latest data directly from data.gov.sg and perform ETL (Extract, Transform, and Load) to a local/web database of choice.

In [19]:

```
import requests
import numpy as np
import pandas as pd
import json
import logging
import time
from requests.exceptions import HTTPError
from pprint import pprint
from functools import wraps
from geopy.distance import geodesic as GD
```

Contents

1. API call data
2. Data Wrangling
3. Feature Engineering

1. Getting the data through API call

Wrapper functions

- To time function calls
- To error handle HTTPerrors and other Exceptions
- To cache API calls

In [20]:

```
logging.basicConfig(filename='wrangling.log', filemode='a', format='%(asctime)s - %(name)s - %(l
logging.warning(f"{'-'*20}New run started {'-'*100}")
```

In [21]:

```
# Wrapper for timing function calls:
def timeit(func):
    '''
    Wrapper to time function call
    '''
    @wraps(func)
    def timeit_wrapper(*args, **kwargs):
        '''
        *args and **kwargs here allow parameters for the original function to be taken in
        and passed to the function contained in the wrapper.
        '''
        current_time = time.strftime("%H:%M:%S", time.localtime())
        start = time.perf_counter()
        result = func(*args, **kwargs)
        end = time.perf_counter()
        time_taken = end-start
        print(f'{func.__name__}() called at \t{current_time} \texecution time: {time_taken:.4f}
        logging.info(f'{func.__name__}() called at \texecution time: {time_taken:.4f} seconds')
        return result
    return timeit_wrapper

def error_handler(func, max_attempts=3, delay=120):
    '''
    Wrapper to catch and handle errors
    '''
    @wraps(func)
    def error_handler_wrapper(*args, **kwargs):
        '''
        *args and **kwargs here allow parameters for the original function to be taken in
        and passed to the function contained in the wrapper, without needed to declare them in t
        '''
        for i in range(max_attempts):
            try:
                result = func(*args, **kwargs)
            except HTTPError as err:
                logging.error(f'{func.__name__}() encountered {err}')
                # Raise exception if we reach max tries
                if i == max_attempts:
                    raise HTTPError(f'Exceeded max tries of {max_attempts}')
                print(f'{func.__name__}() encountered {err}')

                # err.response gives us the Response object from requests module, we can call .s
                if err.response.status_code == 429:
                    print(f'Sleeping for {delay} seconds', end = '\t')
                    time.sleep(delay)
                    print('Retrying...', end='\t')
            except Exception as err:
                logging.error(f'{func.__name__}() encountered {err}')
                print(f'{func.__name__}() encountered {err}')
                break
            else:
                return result
    return error_handler_wrapper

def cache_calls():
    pass
```

Details for Data.gov.sg API call can be found at

https://data.gov.sg/dataset/ckan-datastore-search

In [22]:

```
@timeit
@error_handler
def get_token(location: str):
    '''
    Function to check if API token is still valid and updates API token if outdated
    ##Parameters
        location: filepath (str)
    Returns API token : str
    '''
    with open(location, 'r+') as fp:
        file = fp.read()
        data = json.loads(file)
        response = requests.post("https://developers.onemap.sg/privateapi/auth/post/getToken", d
        token = response.json()
        if token['access_token'] != data['access_token']:
            print(f"New token found")
            data['access_token'] = token['access_token']
            data['expiry_timestamp'] = token['expiry_timestamp']
            fp.seek(0)
            json.dump(data, fp = fp, indent=4)
            print('Updated token json')
            data = json.loads(file)
        return data['access_token']

@timeit
@error_handler
def datagovsg_api_call(url: str, sort: str = 'month desc', limit: int = 100,
                       months:list =[1,2,3,4,5,6,7,8,9,10,11,12],
                       years:list =["2023"]) -> pd.DataFrame:
    '''
    Function to build the API call and construct the pandas dataframe
    ## Parameters
    url: str
        url for API, with resource_id parameters
    sort: str
        field, by ascending/desc, default by Latest month
    limit: int
        maximum entries (API default by OneMap is 100, if not specified)
    months: list
        months desired, int between 1-12
    years: list
        months desired , int
    Returns Dataframe of data : pd.DataFrame
    '''
    month_dict = '{"month":['
    for year in years:
        for month in months: # months 1-12
            month_dict = month_dict + f'"{year}-{str(month).zfill(2)}", '
    month_dict = month_dict[:-2] # Cancel out extra strings <, >
    month_dict = month_dict + ']}'
    url = url+f'&sort={sort}&filters={month_dict}'
    if limit: # API call's default is 100 even without specifying
        print(f'Call limit : {limit}')
        url = url+f'&limit={limit}'
    pprint(f'API call = {url}')
    response = requests.get(url)
    response.raise_for_status()
    data = response.json()
    df = pd.DataFrame(data['result']['records'])
    return df
```

In [23]:
```
df = datagovsg_api_call('https://data.gov.sg/api/action/datastore_search?resource_id=f1765b54-a2
                        sort='month desc',
                        limit = 100000,
                        months = [1,2,3,4,5],
                        years=[2023])
df
```

```
Call limit : 100000
('API call = '
 'https://data.gov.sg/api/action/datastore_search?resource_id=f1765b54-a209-4718-8d38-a39237f502
b3&sort=month '
 'desc&filters={"month":["2023-01", "2023-02", "2023-03", "2023-04", '
 '"2023-05"]}&limit=100000')
datagovsg_api_call() called at  15:21:06        execution time: 1.5891 seconds
```

Out[23]:

	town	flat_type	flat_model	floor_area_sqm	street_name	resale_price	month	remaining_lease	lease_
0	YISHUN	4 ROOM	Model A	104	YISHUN AVE 2	521000	2023-05	64 years 09 months	
1	YISHUN	4 ROOM	Model A	108	YISHUN AVE 2	505000	2023-05	63 years 07 months	
2	YISHUN	4 ROOM	Model A	105	YISHUN AVE 11	500000	2023-05	64 years 04 months	
3	YISHUN	4 ROOM	Model A	104	YISHUN AVE 11	475000	2023-05	64 years 05 months	
4	YISHUN	4 ROOM	Model A	92	YISHUN AVE 11	490000	2023-05	88 years 03 months	
...	...	...	...	...	...	...	...	...	
11110	CLEMENTI	3 ROOM	Model A	69	CLEMENTI AVE 3	670000	2023-01	94 years 08 months	
11111	CLEMENTI	3 ROOM	New Generation	67	CLEMENTI AVE 3	385000	2023-01	54 years 05 months	
11112	CLEMENTI	3 ROOM	New Generation	67	CLEMENTI AVE 2	320000	2023-01	54 years 06 months	
11113	CLEMENTI	3 ROOM	New Generation	67	CLEMENTI AVE 2	430000	2023-01	54 years 08 months	
11114	CLEMENTI	3 ROOM	New Generation	67	CLEMENTI AVE 2	385000	2023-01	54 years	

11115 rows × 12 columns

In [24]:

```
# from dataprep.eda import create_report
# create_report(df).show()
```

2. Data wrangling steps

1. Reindexed dataframe using _id (unique to every resale transaction)
2. Changed room types into float values, with Executive as 5.5 rooms (extra study/balcony/bathroom)
3. Storey range was converted to avg_storey, the avg floor would be used (every value is a difference of 3 storeys)
4. Resale_price, Floor area converted to float values
5. Month was converted into datetime format, to be used to detrend the time series moving average
6. Year/Month was separated into Year and Month for visualisation purposes
7. Remaining lease was converted into remaining months (float)
8. Update capitalisation and street naming conventions (for purpose of API call later)
9. Categorised towns into regions (North, West, East, North-East, Central) https://www.hdb.gov.sg/about-us/history/hdb-towns-your-home

In [25]:

```
@timeit
def clean_df(df: pd.DataFrame):
    '''
    Function to clean the raw dataframe
    ##Parameters
    pd.DataFrame
    ##Cleaning done
```

```
        1. Reindexed dataframe using _id (unique to every resale transaction)
        2. Changed room types into float values, with Executive as 4.5 rooms (extra study/balcony
        3. Storey range was converted to avg_storey, the avg floor would be used (every value is
        4. Resale_price, Floor area converted to float values
        5. Month was converted into datetime format, to be used to detrend the time series moving
        6. Year/Month was separated into Year and Month for visualisation purposes
        7. Remaining lease was converted into remaining months (float)
        8. Update capitalisation and street naming conventions (for purpose of API call later)
        9. Categorised towns into regions (North, West, East, North-East, Central)
    Returns the cleaned dataframe
    '''
    try:
        # Start
        # Step 1: set index to overall id
        step = 1
        df.set_index('_id', inplace=True)

        # Step 2: Create feature "rooms", "avg_storey"
        def categorise_rooms(flat_type):
            '''
            Helper function for categorising number of rooms
            '''
            if flat_type[0] == 'E' or flat_type[0] == 'M':
                return 5.5
            else:
                return float(flat_type[0])

        step = 2
        df['rooms'] = df['flat_type'].apply(categorise_rooms)
        step = 3
        df['avg_storey'] = df['storey_range'].apply(lambda x: (int(x[:2])+int(x[-2:]))/2)

        # Step 4-6: Change dtypes
        df['resale_price'] = df['resale_price'].astype('float')
        df['floor_area_sqm'] = df['floor_area_sqm'].astype('float')
        step = 5
        df['timeseries_month'] = pd.to_datetime(df['month'], format="%Y-%m")
        step = 6
        df['year'] = df['timeseries_month'].dt.year
        df['month'] = df['timeseries_month'].dt.month
        step = 7
        df['lease_commence_date'] = df['lease_commence_date'].astype('int')

        # Calculate remaining_lease
        def year_month_to_year(remaining_lease):
            '''
            Helper function to change year & months, into years (float)
            '''
            remaining_lease = remaining_lease.split(' ')
            if len(remaining_lease) > 2:
                year = float(remaining_lease[0]) + float(remaining_lease[2])/12
            else:
                year = float(remaining_lease[0])
            return year

        df['remaining_lease'] = df['remaining_lease'].apply(year_month_to_year)

        step = 8
        # Step 8: Change capitalization of strings
        for column in df.columns:
            if df[column].dtype == 'O':
                df[column] = df[column].str.title()

        # Update address abbreviations for onemap API call
        abbreviations = {'Sth':'South',
                        '[S][t][^.ri]':'Street ',
                        '[S][t]$':'Street',
                        '[S][t][.]':'Saint',
                        'Nth':'North',
                        'Ave':'Avenue',
                        'Dr':'Drive',
                        'Rd':'Road'}
        for abbreviation, full in abbreviations.items():
            df['street_name'] = df['street_name'].str.replace(abbreviation, full, regex=True)

        # Step 9: Categorise town regions
        step = 9
        town_regions = {'Sembawang' : 'North',
                    'Woodlands' : 'North',
                    'Yishun' : 'North',
                    'Ang Mo Kio' : 'North-East',
                    'Hougang' : 'North-East',
                    'Punggol' : 'North-East',
                    'Sengkang' : 'North-East',
                    'Serangoon' : 'North-East',
                    'Bedok' : 'East',
                    'Pasir Ris' : 'East',
                    'Tampines' : 'East',
                    'Bukit Batok' : 'West',
                    'Bukit Panjang' : 'West',
                    'Choa Chu Kang' : 'West',
                    'Clementi' : 'West',
                    'Jurong East' : 'West',
                    'Jurong West' : 'West',
                    'Tengah' : 'West',
                    'Bishan' : 'Central',
                    'Bukit Merah' : 'Central',
                    'Bukit Timah' : 'Central',
                    'Central Area' : 'Central',
                    'Geylang' : 'Central',
                    'Kallang/Whampoa' : 'Central',
                    'Marine Parade' : 'Central',
                    'Queenstown' : 'Central',
                    'Toa Payoh' : 'Central'}
        df['region'] = df['town'].map(town_regions)
    except Exception as err:
        print(f"Error at step {step}, error message: {err}")
    else:
        # Reorder columns
        temp_df = df[['block', 'street_name']]
        df = df[['resale_price', 'year', 'month', 'timeseries_month', 'region', 'town', 'rooms',
                # Unused columns - 'lease_commence_date', 'flat_model', 'storey_range', 'flat_ty
    return df, temp_df
```

In [26]:

```
df, address_df = clean_df(df)
display(df.dtypes)
df
```

```
clean_df() called at    15:21:07        execution time: 0.1702 seconds
```

```
resale_price                float64
year                          int32
month                         int32
timeseries_month    datetime64[ns]
region                       object
town                         object
rooms                       float64
avg_storey                  float64
floor_area_sqm              float64
remaining_lease             float64
dtype: object
```

Out[26]:

	resale_price	year	month	timeseries_month	region	town	rooms	avg_storey	floor_area_sqm	remai
_id										
154409	521000.0	2023	5	2023-05-01	North	Yishun	4.0	11.0	104.0	
154408	505000.0	2023	5	2023-05-01	North	Yishun	4.0	2.0	108.0	
154407	500000.0	2023	5	2023-05-01	North	Yishun	4.0	8.0	105.0	
154406	475000.0	2023	5	2023-05-01	North	Yishun	4.0	11.0	104.0	
154405	490000.0	2023	5	2023-05-01	North	Yishun	4.0	2.0	92.0	
...	...	...	...	...	...	...	...	...	...	
144115	670000.0	2023	1	2023-01-01	West	Clementi	3.0	17.0	69.0	
144114	385000.0	2023	1	2023-01-01	West	Clementi	3.0	5.0	67.0	
144113	320000.0	2023	1	2023-01-01	West	Clementi	3.0	2.0	67.0	
144112	430000.0	2023	1	2023-01-01	West	Clementi	3.0	11.0	67.0	
144111	385000.0	2023	1	2023-01-01	West	Clementi	3.0	2.0	67.0	

11115 rows × 10 columns

3. Feature Engineering (Geodata)

Lastly, location plays a huge role in house pricing, hence

3.1 Obtaining latitude, longitude, postal codes

3.2 Distance to city center

3.3 Obtaining MRT locations

3.4 Determine nearest MRT and traveling time

3.1 Latitude & longitude from address

Using street name and block, I utilized OneMap API to obtain the latitude, longitude, and postal codes of each flat https://www.onemap.gov.sg/docs

In [27]:

```
@error_handler
def get_location_data(address_df: pd.DataFrame):
    # Getting latitude, longitude, postal code
    @timeit
    def get_lat_long(address_df : pd.DataFrame, sleeptime : float =0.15):
        '''
        API call to get latitude, longitude, and postal code
        ## Parameters
        df : pd.DataFrame
            dataframe for cleaning, should contain columns ['block'] and ['street_name]
        sleeptime : float
            Incorporates sleep time to not exceed a max of 250 calls per min
            Default 0.15s
        '''
        # Lag time between calls
        time.sleep(sleeptime)

        # API call
        address = address_df['block'] + ', ' + address_df['street_name']
        try:
            call = f'https://developers.onemap.sg/commonapi/search?searchVal={address}&returnGeo
            response = requests.get(call)
            response.raise_for_status()
            data = response.json()
            return data['results'][0]['LATITUDE'] + ',' + data['results'][0]['LONGITUDE'] + ' '
        except Exception as err:
            print(f'Error occurred - get_lat_long() API call: {err} on the following call:')
            pprint(call)
            return '0,0 0' # Still return 0 values

    def to_numpy_array(lat_long_df):
        # Build a numpy array from latitude and longitude
        combi = np.array([lat_long_df[0], lat_long_df[1]])
        return combi


    # This calls the API call function row wise
    position = address_df.apply(get_lat_long, axis=1)

    try:
        temp_df = position.str.split(expand=True)
        temp_df.iloc[:,1] = temp_df.iloc[:,1].apply(lambda x: 0 if x=='NIL' else x)
        temp_df.iloc[:,1] = temp_df.iloc[:,1].astype('int')
        lat_long_df = temp_df.iloc[:,0].str.split(pat=',', expand=True)
        lat_long_df = lat_long_df.astype('float')
        numpy_array = lat_long_df.apply(to_numpy_array, axis=1)

    except Exception as err:
        print(f"Error occurred - Splitting data : {err}")
    else:
        geo_data_df = pd.concat([temp_df, lat_long_df, numpy_array], axis=1)
        geo_data_df.columns = ['lat_long', 'postal_code', 'latitude', 'longitude', 'numpy_array'
        return geo_data_df
```

In [ ]:

```
geo_data_df= get_location_data(address_df)
display(geo_data_df.dtypes)
geo_data_df
```

3.2 Distance to city center

The central district of Singapore has the highest housing prices. Property nearer to the city centre tend to have a higher price.

We will make use of this to create a new feature to test if it is significant in model building.

In [29]:

```
@error_handler
def distance_to(df_series : pd.Series, to_address : str , dist_type : str='latlong', verbose : i
    '''
    Function to determine distance to a location (from a series of locations in a dataframe
    ## Parameters
    df_series : pd.Series contains numpy array containing [latitude, longitude]
    to_address : str
        place and streetname
    dist_type : str
        type of distance (latlong, or geodesic)
    verbose : int
        whether to show the workings of the function

    Returns np.Series of distance between input and location
    '''
    # if an address is given
    if isinstance(to_address, str):
        call = f'https://developers.onemap.sg/commonapi/search?searchVal={to_address}&returnGeom
        response = requests.get(call)
        response.raise_for_status()
        data = response.json()
        to_coordinates = np.array([float(data['results'][0]['LATITUDE']), float(data['results'][

    if verbose==1:
        print(f'Coordinates of {to_address} : {to_coordinates}')

    def matrix_operations(from_coordinates, to_coordinates):
        # Matrix substraction to get difference
        distance_diff = from_coordinates - to_coordinates
        absolute_dist = np.absolute(distance_diff)

        #Matrix sum over latitude and longitude of each entry
        sum_of_distances = np.sum(absolute_dist)

        if verbose==2:
            print(f'Difference in distances: \n{distance_diff}')
            print()
            print(f'Absolute difference: \n{absolute_dist}')
            print()
            print(f'Sum of distances \n {sum_of_distances}')

        return sum_of_distances

    def geodesic_operations(from_coordinates, coordinates):
        from_coordinates = tuple(from_coordinates)
        coordinates = tuple(coordinates)
        geodesic_dist = GD(from_coordinates, coordinates).kilometers
        return np.round(geodesic_dist,2)

    if dist_type == 'geodesic':
        diff_dist = df_series.apply(geodesic_operations, coordinates=to_coordinates)
    else:
        diff_dist = df_series.apply(matrix_operations, coordinates=to_coordinates)

    return diff_dist
```

In [30]:

```
dist_to_marina_bay = distance_to(geo_data_df['numpy_array'], 'Marina Bay', dist_type='geodesic',
dist_to_marina_bay = pd.Series(dist_to_marina_bay, name='dist_to_marina_bay')
df = pd.concat([df, dist_to_marina_bay, geo_data_df['latitude'], geo_data_df['longitude']], axis
df
```

```
Coordinates of Marina Bay : [  1.2834542  103.86080905]
```

Out[30]:

	resale_price	year	month	timeseries_month	region	town	rooms	avg_storey	floor_area_sqm	remai
_id										
154409	521000.0	2023	5	2023-05-01	North	Yishun	4.0	11.0	104.0	
154408	505000.0	2023	5	2023-05-01	North	Yishun	4.0	2.0	108.0	
154407	500000.0	2023	5	2023-05-01	North	Yishun	4.0	8.0	105.0	
154406	475000.0	2023	5	2023-05-01	North	Yishun	4.0	11.0	104.0	
154405	490000.0	2023	5	2023-05-01	North	Yishun	4.0	2.0	92.0	
...	...	...	...	...	...	...	...	...	...	
144115	670000.0	2023	1	2023-01-01	West	Clementi	3.0	17.0	69.0	
144114	385000.0	2023	1	2023-01-01	West	Clementi	3.0	5.0	67.0	
144113	320000.0	2023	1	2023-01-01	West	Clementi	3.0	2.0	67.0	
144112	430000.0	2023	1	2023-01-01	West	Clementi	3.0	11.0	67.0	
144111	385000.0	2023	1	2023-01-01	West	Clementi	3.0	2.0	67.0	

11115 rows × 13 columns

3.3 MRT Locations

The location of all MRT stations was also obtained using OneMap API and saved as a json file locally

In [31]:

```
@timeit
@error_handler
def update_mrt_coordinates(mrt_stations=None, filepath='static/mrt_dict.json'):
    '''
    Function to API call for MRT station coordinates and write to json file
    ## Parameters
    mrt_stations : list
        list of mrt station names, default to All stations if nothing is given
    filepath : str
        filepath and name of json file to write to, should end with .json
    Returns None
    '''
    if not mrt_stations:
        mrt_stations = ['Admiralty MRT', 'Aljunied MRT', 'Ang Mo Kio MRT', 'Bakau LRT', 'Bangkit
                         'Bayshore MRT', 'Beauty World MRT', 'Bedok MRT', 'Bedok North MRT', 'Bed
                         'Bendemeer MRT', 'Bishan MRT', 'Boon Keng MRT', 'Boon Lay MRT', 'Botanic
                         'Bras Basah MRT', 'Buangkok MRT', 'Bugis MRT', 'Bukit Batok MRT', 'Bukit
                         'Bukit Panjang MRT', 'Buona Vista MRT', 'Caldecott MRT', 'Cashew MRT', '
                         'Chinatown MRT', 'Chinese Garden MRT', 'Choa Chu Kang MRT', 'City Hall M
                         'Clementi MRT', 'Commonwealth MRT', 'Compassvale LRT', 'Cove LRT', 'Dako
                         'Downtown MRT', 'Xilin MRT', 'Tampines East MRT', 'Mayflower MRT', 'Uppe
                         'Lentor MRT', 'Woodlands North MRT', 'Woodlands South MRT', 'Esplanade M
                         'Expo MRT', 'Fajar LRT', 'Farmway LRT', 'Farrer Park MRT', 'Fort Canning
                         'Gardens by the Bay MRT', 'Geylang Bahru MRT', 'HarbourFront MRT', 'Haw
                         'Holland Village MRT', 'Hougang MRT', 'Jalan Besar MRT', 'Joo Koon MRT',
                         'Jurong West MRT', 'Kadaloor LRT', 'Kaki Bukit MRT', 'Kallang MRT', 'Kem
                         'King Albert Park MRT', 'Kovan MRT', 'Kranji MRT', 'Labrador Park MRT',
                         'Layar LRT', 'Little India MRT', 'Lorong Chuan MRT', 'MacPherson MRT', '
                         'Marsiling MRT', 'Marymount MRT', 'Mattar MRT', 'Meridian LRT', 'Mountba
                         'Newton MRT', 'Nibong LRT', 'Nicoll Highway MRT', 'Novena MRT', 'Oasis L
                         'Outram Park MRT', 'Paya Lebar MRT', 'Pasir Ris MRT', 'Paya Lebar MRT',
                         'Pioneer MRT', 'Potong Pasir MRT', 'Promenade MRT', 'Punggol MRT', 'Quee
                         'Riviera LRT', 'Rochor MRT', 'Sembawang MRT', 'Sengkang MRT', 'Serangoon
```

```
                        'Somerset MRT', 'Springleaf MRT', 'Stadium MRT', 'Stevens MRT', 'Sumang
                        'Tampines East MRT', 'Tampines West MRT', 'Tanah Merah MRT', 'Tanjong Pa
                        'Telok Ayer MRT', 'Telok Blangah MRT', 'Thanggam LRT', 'Tiong Bahru MRT'
                        'Tuas Crescent MRT', 'Tuas Link MRT', 'Tuas West Road MRT', 'Ubi MRT', '
                        'Woodlands MRT', 'Woodlands South MRT', 'Woodlands North MRT', 'Yew Tee 
    # Future stations - 'Tampines North MRT', 'Tengah MRT'

    mrt_coordinates = {}
    for mrt in mrt_stations:
        response = requests.get(f"https://developers.onemap.sg/commonapi/search?searchVal={mrt}&
        response.raise_for_status()
        data = response.json()
        # string (lat,long) as key
        # mrt_coordinates[f"{data['results'][0]['LATITUDE']},{data['results'][0]['LONGITUDE']}"]
        mrt_coordinates[mrt] = (float(data['results'][0]['LATITUDE']),float(data['results'][0]['

    with open(filepath, 'w')as f:
        json.dump(mrt_coordinates, f, indent=4)

@timeit
@error_handler
def get_mrt_coordinates(filepath = 'static/mrt_dict.json'):
    '''
    Function to read saved mrt_coordinates from json file
    ## Parameters
    filepath : str
        filepath to json file
    Returns data : dictionary
    '''
    with open(filepath, 'r') as f:
        file = f.read()
        data = json.loads(file)
        return data
```

Load Json file and convert to numpy array to utilize matrix operations.

In [32]:

```
mrt_coordinates_dict = get_mrt_coordinates()

# Convert coordinates into numpy arrays
mrt_stations = np.array(list(mrt_coordinates_dict.keys()))
mrt_coordinates = np.array(list(mrt_coordinates_dict.values()))
```

```
get_mrt_coordinates() called at         16:04:03        execution time: 0.6115 seconds
```

3.4 Nearest MRT stations and Minimum distance/time

- Using the matrix operations, we are able to find the nearest MRT station by absolute distance
- Then use OneMap's route_api_call() to get distance/time to MRT stations

In [33]:

```
@error_handler
def find_nearest_stations(geo_data_df : pd.DataFrame, mrt_stations : np.array=mrt_stations, mrt_
                          n_nearest_stations: int=2, verbose : int=0):
    '''
    Function to determine nearest MRT station of the resale_flat based on latitude and longitude
    ## Parameters
        geo_data_df : pd.DataFrame
        mrt_stations : np.array
        mrt_coordinates : np.array
        n_nearest_stations: int=2
        verbose : int=0

    Returns a list of n_nearest stations
    '''
    # Matrix substraction to get difference with each MRT, convert to absolute values
    distance_diff = geo_data_df['numpy_array'] - mrt_coordinates
    absolute_dist = np.absolute(distance_diff)

    # Matrix sum over latitude and longitude of each entry
    sum_of_distances = np.sum(absolute_dist, axis=1)

    # Sort and search based on desired n_nearest_stations
    sorted_distances = np.sort(sum_of_distances)
    nearest_stations = []
    for n in range(n_nearest_stations):
        idx = np.where(sum_of_distances==sorted_distances[n])
        from_coordinates = tuple(geo_data_df['numpy_array'])
        to_coordinates = tuple(mrt_coordinates[idx][0])
        geodesic_dist = GD(from_coordinates, to_coordinates).kilometers
        nearest_stations.append(mrt_stations[idx][0])
        nearest_stations.append(np.round(geodesic_dist,2))

    if verbose==1:
        print(f'Difference in distances: \n{distance_diff[:5]}')
        print()
        print(f'Absolute difference: \n{absolute_dist[:5]}')
        print()
        print(f'Sum of distances \n {sum_of_distances[:5]}')
        print()
        print(f'Sorted distances\n{sorted_distances[:5]}')
        print()
        print(f'Top {n_nearest_stations}')
        print(nearest_stations)

    return nearest_stations
```

In [34]:

```
n_nearest_stations = 1
# Matrix operations to find nearest MRT stations for each row
nearest_stations = geo_data_df.apply(find_nearest_stations, n_nearest_stations=n_nearest_station
nearest_stations_df = pd.DataFrame(nearest_stations.tolist(), index=geo_data_df.index, columns=[
nearest_stations_df
```

Out[34]:

	nearest_station_0	dist_to_station_0
_id		
154409	Yishun MRT	0.81
154408	Yishun MRT	0.85
154407	Yishun MRT	1.01
154406	Yishun MRT	1.49
154405	Yishun MRT	1.27
...	...	...
144115	Clementi MRT	0.15
144114	Clementi MRT	0.36
144113	Clementi MRT	0.32
144112	Clementi MRT	0.63
144111	Clementi MRT	0.50

11115 rows × 2 columns

In [35]:
```
df = pd.concat([df, nearest_stations_df], axis=1)
df
```

Out[35]:

_id	resale_price	year	month	timeseries_month	region	town	rooms	avg_storey	floor_area_sqm	remai
154409	521000.0	2023	5	2023-05-01	North	Yishun	4.0	11.0	104.0	
154408	505000.0	2023	5	2023-05-01	North	Yishun	4.0	2.0	108.0	
154407	500000.0	2023	5	2023-05-01	North	Yishun	4.0	8.0	105.0	
154406	475000.0	2023	5	2023-05-01	North	Yishun	4.0	11.0	104.0	
154405	490000.0	2023	5	2023-05-01	North	Yishun	4.0	2.0	92.0	
...	...	...	...	...	...	...	...	...	...	
144115	670000.0	2023	1	2023-01-01	West	Clementi	3.0	17.0	69.0	
144114	385000.0	2023	1	2023-01-01	West	Clementi	3.0	5.0	67.0	
144113	320000.0	2023	1	2023-01-01	West	Clementi	3.0	2.0	67.0	
144112	430000.0	2023	1	2023-01-01	West	Clementi	3.0	11.0	67.0	
144111	385000.0	2023	1	2023-01-01	West	Clementi	3.0	2.0	67.0	

11115 rows × 15 columns

In [36]:
```
name = input('Name save file: e.g. <2023_apr>\n')
if name != '':
    filename= f'static/{name}.csv'
    df.to_csv(filename)
    print(f'File saved as {filename}')
```

```
File saved as static/2023_till_05.csv
```

Retired code below, too slow due to numerous API calls

Get minimum distance/time using OneMap API call

def route_api_call(routeType: str, start: str, end: str, metric: str, credentials : str, date = '01-26-2023', time_start = '07:35:00', mode = 'TRANSIT', maxWalkDistance = 1000, numItineraries = 2, verbose=0, recursive_call=None): ''' Function to api call OneMap for routing ## Parameters routeType : str option between ['walk','drive','cycle', 'pt] Below only applicable if routeType == 'pt' date : str MM-DD-YYYY default '01-26-2023' time : str HH:MM:SS default '07:35:00' mode : str choose between TRANSIT, BUS, RAIL default 'TRANSIT' maxWalkDistance : int max walking distance allowed, in meters default 1000 numItineraries : int number of suggested routes default 2 verbose : int 1 to print time and distance, 2 for the whole json response default 0 ### Returns (time, distance) for chosen routeType time is in seconds total_distance is in metres. ''' # Lag time between calls to ensure we stay within 250 calls per minute, 0.24 is calculated time # Removed, server lag response gives us an average of about 0.7s per call already, no need to slow down somemore # time.sleep(0.24) # Walk if routeType in ['walk','drive','cycle']: response = requests.get(f"https://developers.onemap.sg/privateapi/routingsvc/route?start={start}&end={end}&routeType={routeType}&token={credentials}") response.raise_for_status() data = response.json() time_taken = data['route_summary']['total_time'] distance = data['route_summary']['total_distance'] if verbose==1: print(f'Walking time: {time_taken}') print(f'Walking distance: {distance}') # Public transport elif routeType == 'pt': response = requests.get(f"https://developers.onemap.sg/privateapi/routingsvc/route?start={start}&end={end}&routeType={routeType}&token={credentials}&date={date}&time={time_start}&mode={mode}&maxWalkDistance={maxWalkDistance}&numItineraries={numItineraries}") response.raise_for_status() data = response.json() summary =

```
{'walkTime': data['plan']['itineraries'][0]['walkTime'], 'transitTime': data['plan']['itineraries'][0]['transitTime'], 'waitingTime':
data['plan']['itineraries'][0]['waitingTime'] } distance = time_taken = sum(summary.values()) pt_walk_distance = data['plan']
['itineraries'][0]['walkDistance'] if verbose==1: pprint(summary) print(f'Total public transport time: {time_taken}') print(f'Walk
distance to public transport: {pt_walk_distance}') else: raise KeyError("Enter valid routeType, choose between
'walk','drive','cycle', 'pt'") # To end the call if verbose==2: pprint(data) '''# To Let us know if the retry on recursive call is
successful if recursive_call: print('\tRetry successful')''' return time_taken if metric=='time' else distance @timeit
@error_handler def time_taken_to_station(geo_data_df, credentials, mrt_coordinates_dict=mrt_coordinates_dict,
n_nearest_stations=n_nearest_stations): ''' Function to coordinate route_api_call() to build walking distance and minimum time
to nearest mrts ''' start = geo_data_df['lat_long'] # Columns will depend on how many columns of nearest_stations we
obtained previously, defaulted to 2 columns = geo_data_df[['nearest_station_'+ str(x) for x in range(n_nearest_stations)]]
time_distance = [] for index, mrt_station in enumerate(columns): # List comprehension to build latitude and longitude in string
(1.121231,102.123123) list_of_strings = [str(x) for x in mrt_coordinates_dict[mrt_station]] end = ','.join(list_of_strings) # Only
return closest station's walking distance if index==0: walk= route_api_call('walk', start, end, 'distance', credentials) if walk:
time_distance.append(walk) else: time_distance.append(0) # Return time for each station pt = route_api_call('pt', start, end,
'time', credentials, numItineraries = 1) if pt: time_distance.append(pt) else: time_distance.append(0) return time_distance
```

Due to the large amount of API calls, we will split the data into batches to extract the data.

```
@error_handler def split_df(geo_data_df: pd.DataFrame, interval: int=500): splitted_df_list = [] for start in range(0,
len(geo_data_df.index), interval): splitted_df_list.append(geo_data_df.iloc[start:start+interval , :]) print(f'Number of dataframes
split into: {len(splitted_df_list)}') return splitted_df_list def iterate_function(splitted_df_list: list, results: list, func: function, start:
int, stop: int): ''' Appends to results (list) in place. ''' print(f'Writing to {id(results)} with {len(results)} elements already present')
for index, splitted_df in enumerate(splitted_df_list): if index >= start and index < stop: time_distance = splitted_df.apply(func,
credentials=credentials, n_nearest_stations=n_nearest_stations, axis=1) results.append(time_distance) cont = input(f'Done with
index {index}, continue? Y/N \n') if cont.lower() == 'n': break print(f'Length of updated results list: {len(results)}') splitted_df_list
= split_df(geo_data_df, interval=400)
```

Run the code by batches while appending the results to a list inplace

```
credentials=get_token("venv/onemap.json") time_distance_list = [] iterate_function(splitted_df_list, time_distance_list,
time_taken_to_station, 0, len(splitted_df_list))
```

Put the DataFrame back together if all runs successful

```
if len(splitted_df_list) == len(time_distance_list): time_distance =
pd.DataFrame(pd.concat(time_distance_list).to_dict()).transpose() time_distance.columns=['dist_to_station']+['time_route_'+
str(x) for x in range(n_nearest_stations)] display(time_distance) else: raise IndexError('Mismatch in length of starting and results
list')
```

Determine minimum time

```
# temporary df to find minimum time among public transport times temp_df = time_distance.drop(labels=['dist_to_station'],
axis=1) min_pt_time = temp_df.min(axis=1).rename('min_pt_time') geo_data_df =
pd.concat([time_distance.loc[:,'dist_to_station'],min_pt_time], axis=1) # Unused columns ['lat_long', 'latitude', 'longitude',
'postal_code']+ geo_data_df
```

Tidying up the full dataframe

```
df = pd.concat([df, geo_data_df], axis=1) df
```