AY 2024/25 SEMESTER 1

SC4020 DATA ANALYTICS & MINING

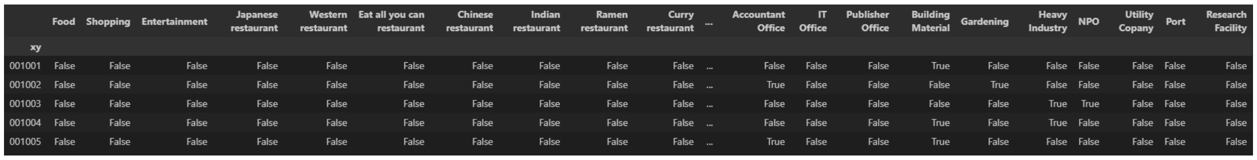
Project 2 - Report

By Group 9

Task 1: Analysis of Co-occurrence Patterns of Points of Interest

Data Preparation for Apriori Algorithm

To prepare our data, we first checked for any null values and then proceeded to transform it into a suitable form for use by the Apriori Algorithm. We decided to ignore the specific quantities of each POI, focusing only on the presence of POI categories.

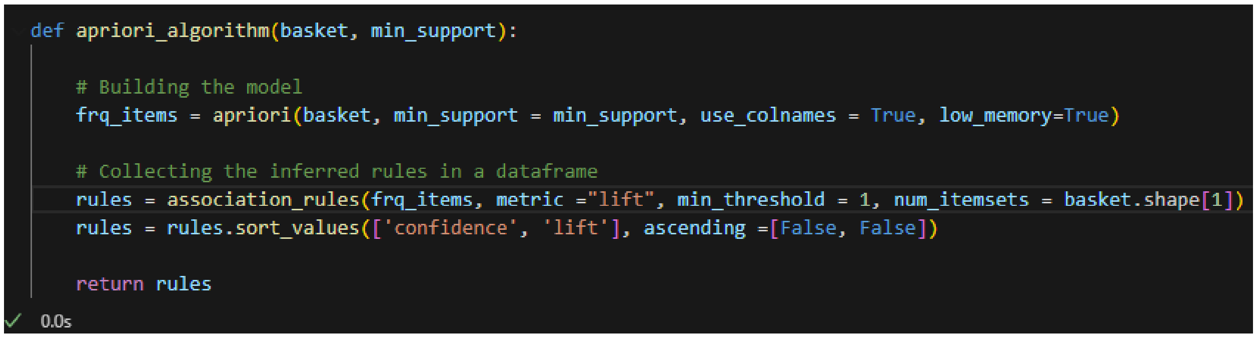


	Food	Shopping	Entertainment	Japanese restaurant	Western restaurant	Eat all you can restaurant	Chinese restaurant	Indian restaurant	Ramen restaurant	Curry restaurant	...	Accountant Office	IT Office	Publisher Office	Building Material	Gardening	Heavy Industry	NPO	Utility Copany	Port	Research Facility
xy																					
001001	False	False	False	False	False	False	False	False	False	False	...	False	False	False	True	False	False	False	False	False	False
001002	False	False	False	False	False	False	False	False	False	False	...	True	False	False	False	True	False	False	False	False	False
001003	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	True	True	False	False	False
001004	False	False	False	False	False	False	False	False	False	False	...	False	False	False	True	False	True	False	False	False	False
001005	False	False	False	False	False	False	False	False	False	False	...	True	False	False	True	False	False	False	False	False	False

Each grid cell represents a “basket” while the POI categories that appear in that grid are the “items”. We also renamed the columns using the POI_datacategories.csv for easy analysis.

Analysis of Frequent Itemsets

We then utilised the Apriori Algorithm from the MLXtend Python Package..



```python
def apriori_algorithm(basket, min_support):

    # Building the model
    frq_items = apriori(basket, min_support = min_support, use_colnames = True, low_memory=True)

    # Collecting the inferred rules in a dataframe
    rules = association_rules(frq_items, metric ="lift", min_threshold = 1, num_itemsets = basket.shape[1])
    rules = rules.sort_values(['confidence', 'lift'], ascending =[False, False])

    return rules
```
```
✓ 0.0s
```

Here we used the minimum min_support value we could for each city, while having a reasonable computational time.

min_support values for **City A**: 0.06, **City B**: 0.03, **City C**: 0.09, **City D**: 0.03

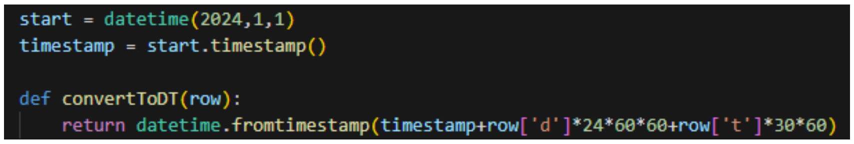
We then focused on 4 features, support, confidence, lift and interest to analyse the results. A short analysis of each feature for each city can be found in the code, as well as the results.

In general, the observations that stood out were:

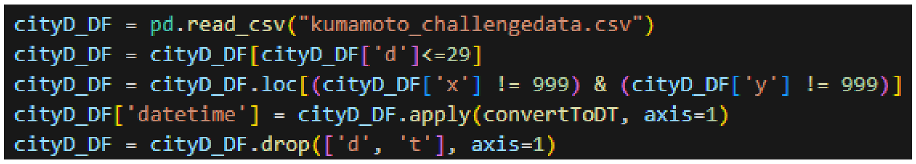
- ‘Hair Salon’ had a near 1.0 confidence when health and leisure related places (Chiropractic, Japanese restaurant, Fishing, Hospital, Café, Clothes Store, etc) were visited for City A, B and D
- City C had a very high occurrence of the item ‘Elderly Care Home’ in the antecedents

Task 2

Firstly, we went ahead to convert the ‘d’ and ‘t’ columns into datetime format and proceeded to drop the ‘d’ and ‘t’ columns. We also limited the data analysis to the first 30 days. Finally, for city B, C and D datasets we excluded the rows with missing data.

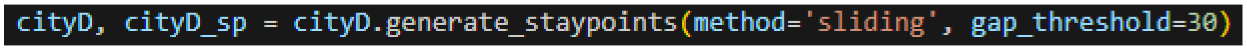


```python
start = datetime(2024,1,1)
timestamp = start.timestamp()

def convertToDT(row):
    return datetime.fromtimestamp(timestamp+row['d']*24*60*60+row['t']*30*60)
```

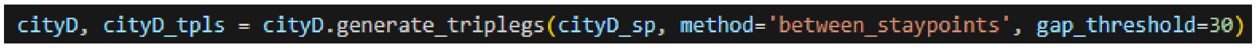


```python
cityD_DF = pd.read_csv("kumamoto_challengedata.csv")
cityD_DF = cityD_DF[cityD_DF['d']<=29]
cityD_DF = cityD_DF.loc[(cityD_DF['x'] != 999) & (cityD_DF['y'] != 999)]
cityD_DF['datetime'] = cityD_DF.apply(convertToDT, axis=1)
cityD_DF = cityD_DF.drop(['d', 't'], axis=1)
```

We then proceed with the generation of triplegs using the trackintel library which requires us to generate staypoints then using those staypoints to generate triplegs. The gap_threshold is set to 30 since the gap between temporal gap of the given dataset is 30 minutes

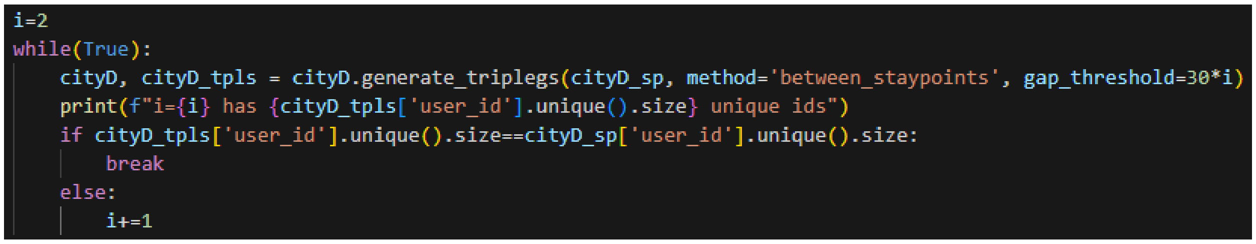


```python
cityD, cityD_sp = cityD.generate_staypoints(method='sliding', gap_threshold=30)
```



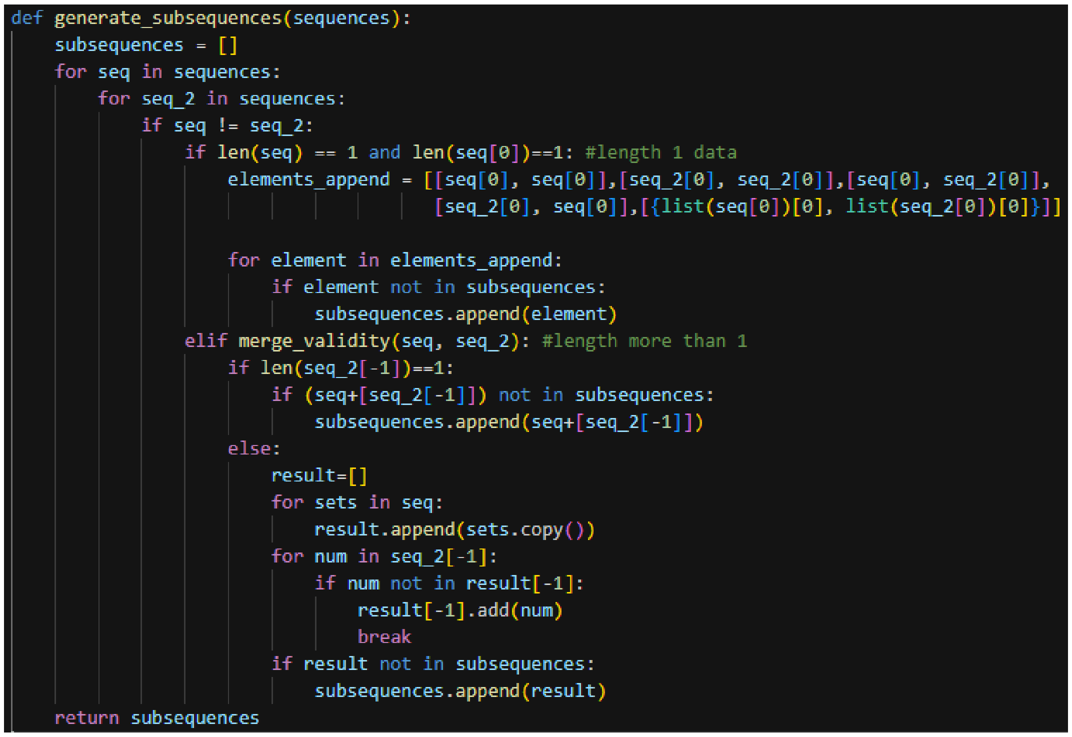
```python
cityD, cityD_tpls = cityD.generate_triplegs(cityD_sp, method='between_staypoints', gap_threshold=30)
```

After generating triplegs for all the cities with gap_threshold being 30 minutes, we discovered that there are quite missing user_ids from the triplegs generated, so we ran the code below till about i=200 to get the smallest possible i with the biggest possible number of unique user_ids so that we have analysis on as many user_ids as possible



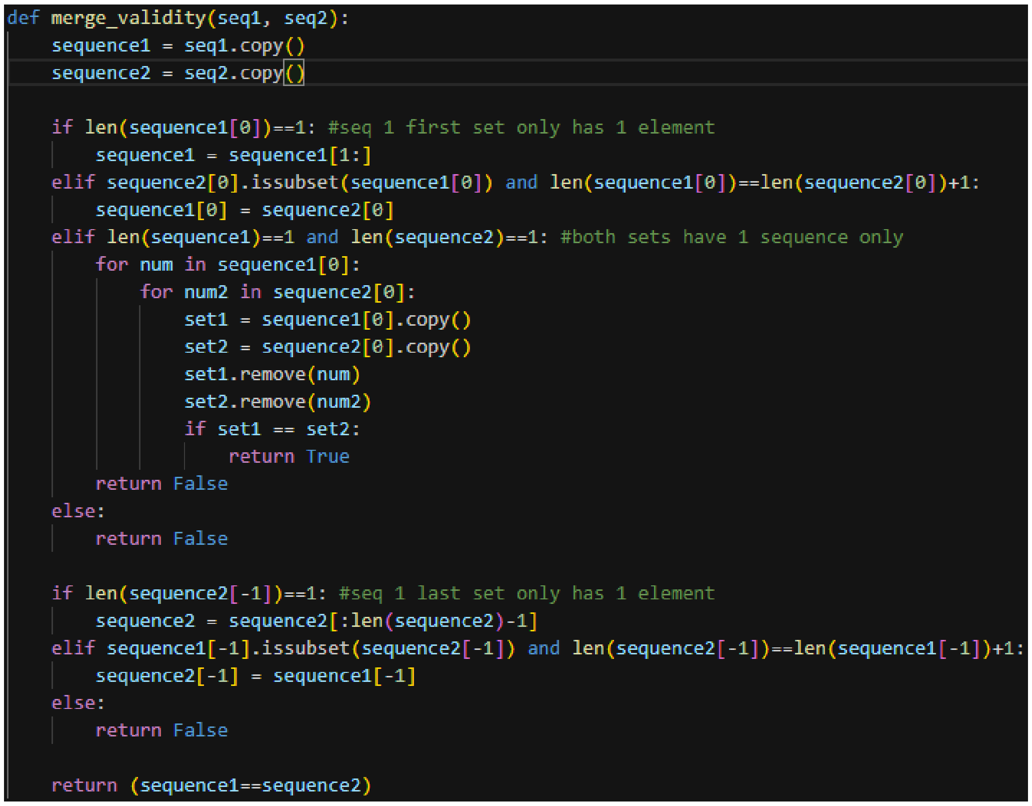
```python
i=2
while(True):
    cityD, cityD_tpls = cityD.generate_triplegs(cityD_sp, method='between_staypoints', gap_threshold=30*i)
    print(f"i={i} has {cityD_tpls['user_id'].unique().size} unique ids")
    if cityD_tpls['user_id'].unique().size==cityD_sp['user_id'].unique().size:
        break
    else:
        i+=1
```

After all the data preprocessing, we proceed with constructing the appropriate algorithm to be used. Here we have the function to generate subsequences from k length frequent subsequences to k+1 length subsequences where for k>2 it will apply the rules of merging subsequences by comparing the first k-1 length elements of the 2 distinct sequences. The k=2 case is detected by the fact that it is a 1 length subsequence so it has only 1 set with 1 element. Then, everything else is k>2 where we will check for the validity of merging with another function.



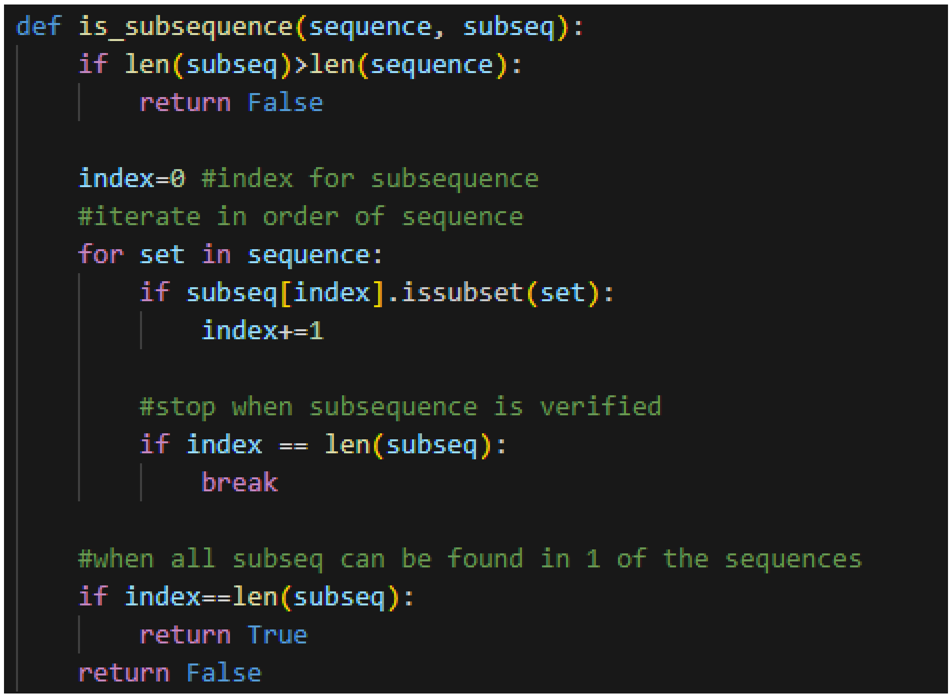
```python
def generate_subsequences(sequences):
    subsequences = []
    for seq in sequences:
        for seq_2 in sequences:
            if seq != seq_2:
                if len(seq) == 1 and len(seq[0])==1: #length 1 data
                    elements_append = [[seq[0], seq[0]],[seq_2[0], seq_2[0]],[seq[0], seq_2[0]],
                                       [seq_2[0], seq[0]],[{list(seq[0])[0], list(seq_2[0])[0]}]]

                    for element in elements_append:
                        if element not in subsequences:
                            subsequences.append(element)
                elif merge_validity(seq, seq_2): #length more than 1
                    if len(seq_2[-1])==1:
                        if (seq+[seq_2[-1]]) not in subsequences:
                            subsequences.append(seq+[seq_2[-1]])
                    else:
                        result=[]
                        for sets in seq:
                            result.append(sets.copy())
                        for num in seq_2[-1]:
                            if num not in result[-1]:
                                result[-1].add(num)
                                break
                        if result not in subsequences:
                            subsequences.append(result)
    return subsequences
```

Here is the function to check for validity of merge. It checks whether removing an element in the first set of seq1 matches removing an element in the last set of seq2 which can be broken down into several cases such as either or both of the sets having 1 set of sequences only or both having multiple sets.



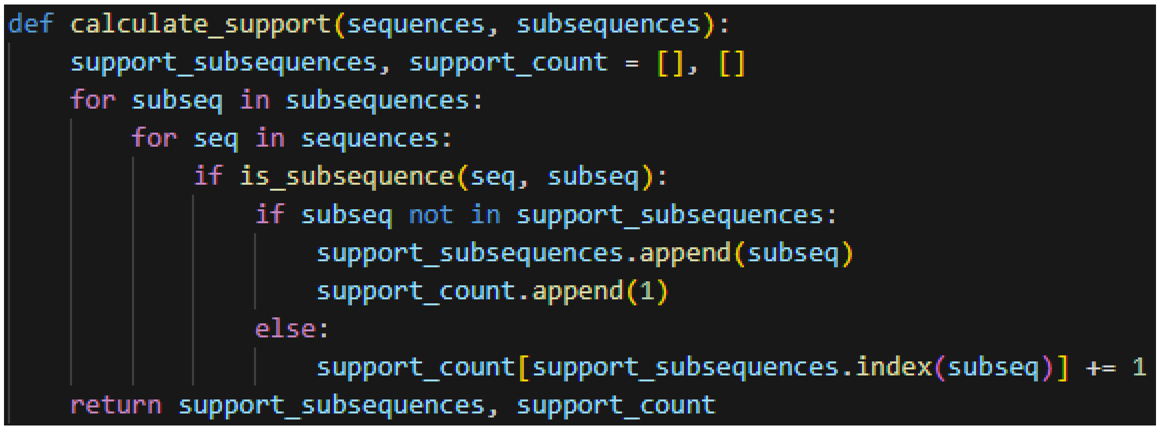
```python
def merge_validity(seq1, seq2):
    sequence1 = seq1.copy()
    sequence2 = seq2.copy()

    if len(sequence1[0])==1: #seq 1 first set only has 1 element
        sequence1 = sequence1[1:]
    elif sequence2[0].issubset(sequence1[0]) and len(sequence1[0])==len(sequence2[0])+1:
        sequence1[0] = sequence2[0]
    elif len(sequence1)==1 and len(sequence2)==1: #both sets have 1 sequence only
        for num in sequence1[0]:
            for num2 in sequence2[0]:
                set1 = sequence1[0].copy()
                set2 = sequence2[0].copy()
                set1.remove(num)
                set2.remove(num2)
                if set1 == set2:
                    return True
        return False
    else:
        return False

    if len(sequence2[-1])==1: #seq 1 last set only has 1 element
        sequence2 = sequence2[:len(sequence2)-1]
    elif sequence1[-1].issubset(sequence2[-1]) and len(sequence2[-1])==len(sequence1[-1])+1:
        sequence2[-1] = sequence1[-1]
    else:
        return False

    return (sequence1==sequence2)
```

Here is the function to check whether a sequence subseq is a subsequence of seq which checks whether all sets of subseq is a subset of sets in sequences in the correct order.



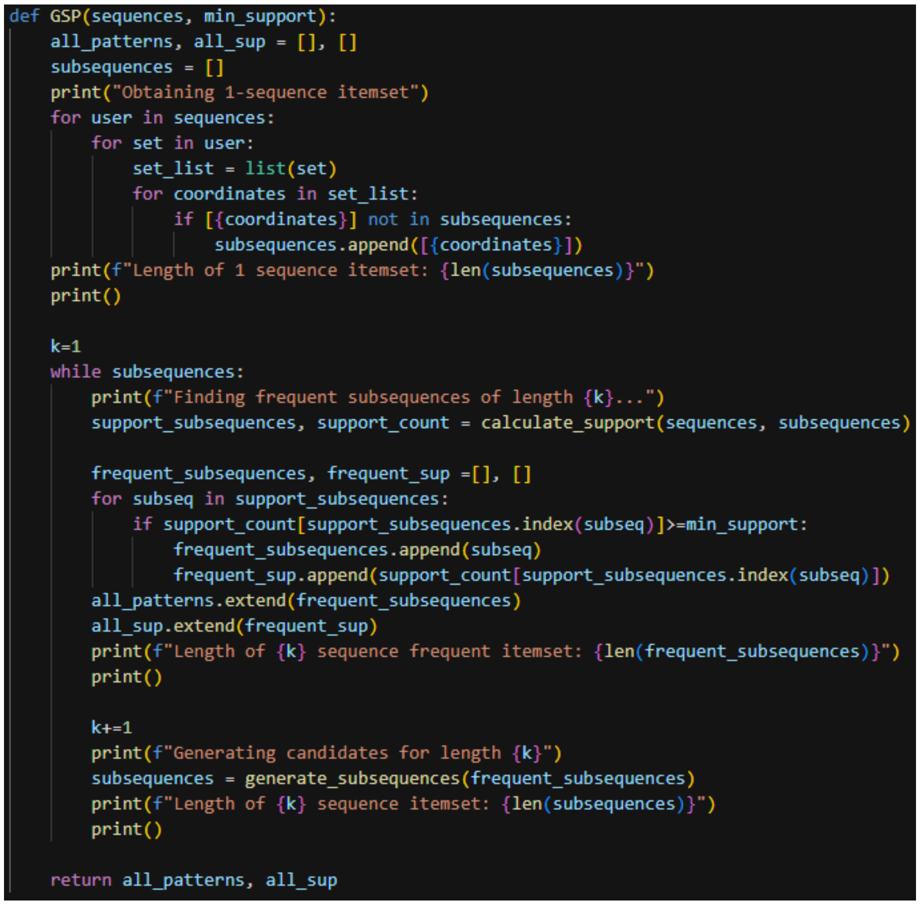
```python
def is_subsequence(sequence, subseq):
    if len(subseq)>len(sequence):
        return False

    index=0 #index for subsequence
    #iterate in order of sequence
    for set in sequence:
        if subseq[index].issubset(set):
            index+=1

        #stop when subsequence is verified
        if index == len(subseq):
            break

    #when all subseq can be found in 1 of the sequences
    if index==len(subseq):
        return True
    return False
```

And the final helper function here helps to calculate the support of a list of subsequences by using the is_subsequence function above. It outputs the list of support count in order of the subsequences list



```python
def calculate_support(sequences, subsequences):
    support_subsequences, support_count = [], []
    for subseq in subsequences:
        for seq in sequences:
            if is_subsequence(seq, subseq):
                if subseq not in support_subsequences:
                    support_subsequences.append(subseq)
                    support_count.append(1)
                else:
                    support_count[support_subsequences.index(subseq)] += 1
    return support_subsequences, support_count
```

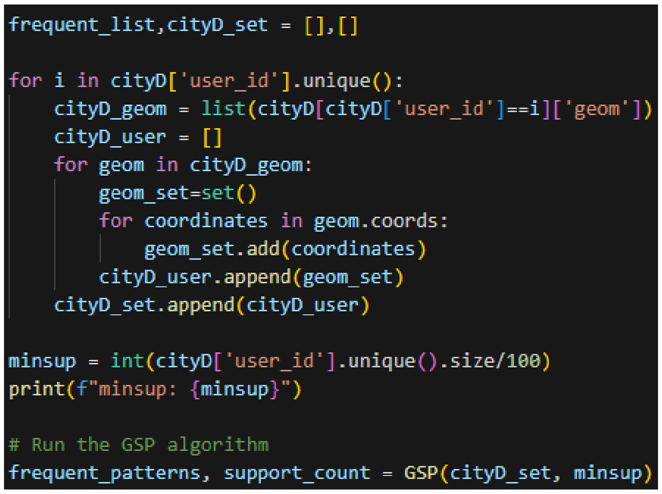
Then putting this all together we have the GSP algorithm function which first parses through the data to get all the 1-length subsequence, then continuously finds frequent patterns.



```python
def GSP(sequences, min_support):
    all_patterns, all_sup = [], []
    subsequences = []
    print("Obtaining 1-sequence itemset")
    for user in sequences:
        for set in user:
            set_list = list(set)
            for coordinates in set_list:
                if [{coordinates}] not in subsequences:
                    subsequences.append([{coordinates}])
    print(f"Length of 1 sequence itemset: {len(subsequences)}")
    print()

    k=1
    while subsequences:
        print(f"Finding frequent subsequences of length {k}...")
        support_subsequences, support_count = calculate_support(sequences, subsequences)

        frequent_subsequences, frequent_sup =[], []
        for subseq in support_subsequences:
            if support_count[support_subsequences.index(subseq)]>=min_support:
                frequent_subsequences.append(subseq)
                frequent_sup.append(support_count[support_subsequences.index(subseq)])
        all_patterns.extend(frequent_subsequences)
        all_sup.extend(frequent_sup)
        print(f"Length of {k} sequence frequent itemset: {len(frequent_subsequences)}")
        print()

        k+=1
        print(f"Generating candidates for length {k}")
        subsequences = generate_subsequences(frequent_subsequences)
        print(f"Length of {k} sequence itemset: {len(subsequences)}")
        print()

    return all_patterns, all_sup
```

Result

Then we proceeded to use the GSP algorithm above to find the frequent subsequences for each city's unique user_id. For each city, each unique user_id and their data points will be iterated, with minimum support set as 1% of the number of user_ids of the city dataset. The results can be found on the following CSVs: cityA_task2_result.csv, cityB_task2_result.csv, cityC_task2_result.csv & cityD_task2_result.csv



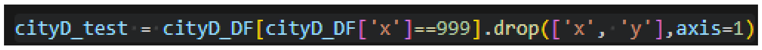
```python
frequent_list,cityD_set = [],[]

for i in cityD['user_id'].unique():
    cityD_geom = list(cityD[cityD['user_id']==i]['geom'])
    cityD_user = []
    for geom in cityD_geom:
        geom_set=set()
        for coordinates in geom.coords:
            geom_set.add(coordinates)
        cityD_user.append(geom_set)
    cityD_set.append(cityD_user)

minsup = int(cityD['user_id'].unique().size/100)
print(f"minsup: {minsup}")

# Run the GSP algorithm
frequent_patterns, support_count = GSP(cityD_set, minsup)
```

Task 3

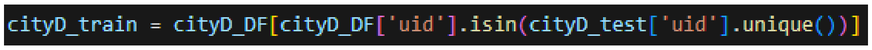
For this task, we took on predicting the missing data of cities B, C and D using the K Nearest Neighbours model.

Dataset Preparation

For each city, we prepared the datasets by separating rows with missing coordinates $(x == 999)$ into a test set, while the remaining rows were designated as the training set. This process was performed individually for each city. Additionally, the data was processed using unique user IDs (uid), ensuring that the model captured the specific movement patterns of each user.



```
cityD_test = cityD_DF[cityD_DF['x']==999].drop(['x', 'y'],axis=1)
```



```
cityD_train = cityD_DF[cityD_DF['uid'].isin(cityD_test['uid'].unique())]
```

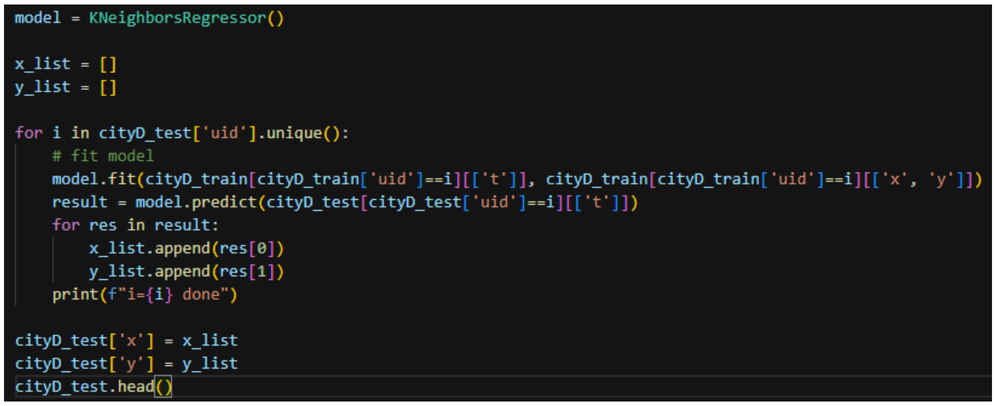
Model

The K Nearest Neighbours (KNN) algorithm was employed to predict the missing spatial coordinates (x, y) for each city. KNN is a non-parametric method that relies on proximity in the feature space to make predictions. The algorithm was implemented using the KNeighborsRegressor class from the scikit-learn library.

The model took the timestamp (t) as its input feature and predicted the missing spatial coordinates (x, y). The Euclidean distance metric was used to identify the nearest neighbours for each prediction. Once again, the training process was performed separately for each user ID (uid) in the training set, ensuring that the model learned user-specific movement patterns.

Results

The trained KNN model was applied to the test dataset to predict missing spatial coordinates for each user ID. The model used time values as features and generated corresponding predictions for the missing x and y coordinates. The results were saved as updated datasets for each city, with the missing values replaced by the predicted coordinates. The outputs were stored in the following CSV files: cityB_pred.csv, cityC_pred.csv and cityD_pred.csv.



```
model = KNeighborsRegressor()

x_list = []
y_list = []

for i in cityD_test['uid'].unique():
    # fit model
    model.fit(cityD_train[cityD_train['uid']==i][['t']], cityD_train[cityD_train['uid']==i][['x', 'y']])
    result = model.predict(cityD_test[cityD_test['uid']==i][['t']])
    for res in result:
        x_list.append(res[0])
        y_list.append(res[1])
    print(f"i={i} done")

cityD_test['x'] = x_list
cityD_test['y'] = y_list
cityD_test.head()
```

Name	Matric No.	Contributions
Wong Zi Lun	U2240220D	1. Task 1 2. Report Writing 3. Video
Javen Ong Jing Hian	U2240610E	1. Task 2 2. Report Writing 3. Video
Thejesvi Ramesh	U2240554H	1. Task 2 2. Report Writing 3. Video
Florian Goering	U2220036A	1. Task 3 2. Report Writing 3. Video