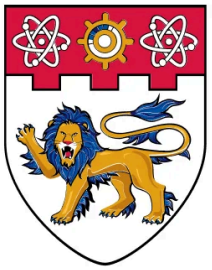


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4032 SC4020 Project 2

Members:

Poh Zi Jie Isaac
Dexter Voon Kai Xian
Lim Jun Yu
Mukherjee Tathagato

Task Description

2.1 Analysis of Co-occurrence Patterns of Points of Interest (POI)

This section explains the implementation of Apriori algorithm on the Points of Interest (POIs) distribution data of 4 cities. The execution of these techniques explained below will derive sets of POIs that frequently appear together within the same grid cell.

No online code is used in this section. The modules *pandas* and *itertools* are used for dataframe handling and subset generation respectively.

2.1.1 Data pre-processing

data_preparation() is used for the pre-processing of data. Given the POIs distribution data of a city, each grid cell is converted into a basket of POIs. The function returns *ls*, a nested list where each element in *ls* represents all the POIs found in a specific grid. Additionally, the distinct categories that exist in the city are returned as *distinct_category* from the function.

2.1.2 Apriori algorithm

2.1.2.1 Candidate Generation

candidate_generation() is used to generate candidate itemset C_{k+1} from F_k. It uses the $F_{k-1}xF_{k-1}$ method, where the union of two pairs of itemsets from F_k becomes a candidate itemset if the first *k - 1* items of the pair are identical. This is illustrated in the snippet below:

```
98          for i in range(length_f_k):
99              p = f_k_copy[i]
100             for j in range(i, length_f_k):
101                 q = f_k_copy[j]
102                 if p[:-1] == q[:-1]:
103                     itemset = list(p).copy()
104                     itemset.append(q[-1])
105                     c_kp1.append(tuple(itemset))
```

length_f_k is the number of itemsets in F_k, while *f_k_copy* is simply a copy of F_k. The two loops iterate through every possible pair of itemsets in F_k and append the union to C_{k+1} if the first *k - 1* items of the pair are identical.

2.1.2.2 Candidate Pruning

candidate_pruning() prunes candidate itemset in C_{k+1} if for any itemset, its subset of length *k* is not in F_k, in other words not frequent.

```
134     i = 0
135     count = 0
136     while i < len(c_kp1):
137         itemset = c_kp1[i]
138         subsets = generate_subset(itemset, k)
139         for subset in subsets:
140             for index, frequent_item in enumerate(f_k):
141                 if subset == frequent_item:
142                     index -= 1
143                     break
144             if index == len(f_k) - 1:
145                 c_kp1.remove(itemset)
146                 i -= 1
147                 count += 1
148                 break
149         i += 1
```

The outer loop at line 136 iterates over every candidate itemset in C_{k+1}. For each itemset, *generate_subset()* generates all possible subsets of length *k*. The two loops at line 139 and line 140 then iterates over every possible subset of the candidate

itemset in C_{k+1} and every frequent itemset in F_k. If a subset is found to be frequent, the loop iterates to the next subset and repeats the process. If any subset of a candidate itemset is found to be infrequent, the itemset will be removed from C_{k+1}.

2.1.2.3 Candidate Elimination

candidate_elimination() counts the support of each candidate itemset and eliminates any non-frequent items.

```
182     sup_count = support_counting(c_kp1, ls)
183     to_remove = []
184     for index, count in enumerate(sup_count):
185         if count < minsup: to_remove.append(index)
186     for i in sorted(to_remove, reverse=True):
187         c_kp1.pop(i)
```

support_counting() first counts the support of each itemset in C_{k+1}. The loop at line 184 iterates over all itemsets and maintains a list of itemsets where the support of these itemsets is less than the minimum support threshold. Lastly, these infrequent itemsets are eliminated from C_{k+1}.

2.1.2.4 Main Mining Loop

The main mining loop is contained in *main()*. The variable *cities* specifies all the cities to analyse, and *support_thresholds* specifies the variations of minimum support threshold. After data pre-processing, the mining loop for a city and a specified support threshold is illustrated in the snippet below. *freq_itemset* is a list to contain all frequent itemsets of any length, and is the output of the Apriori algorithm.

```
293             k = 1
294             c_kp1 = []
295             minsup = round(len(ls) * support_threshold)
296             f_k = f_1(ls, distinct_category, minsup)
297             freq_itemset = []
298
299             while f_k:
300                 print(f'\tk: {k}')
301                 c_kp1 = candidate_generation(f_k, k)
302                 candidate_pruning(c_kp1, f_k, k)
303                 candidate_elimination(c_kp1, ls, minsup)
304                 add_freq_itemset(c_kp1, freq_itemset)
305                 f_k = c_kp1
306                 k += 1
```

In the loop, *candidate_generation()*, *candidate_pruning()* and *candidate_elimination()* are run sequentially in each iteration. At the end of each iteration, $F_{k+1} \leftarrow C_{k+1}$. *add_freq_itemset* adds all new frequent itemsets into *freq_itemset*. It also removes any itemset in *freq_itemset* of length *k* where it is a subset of any new frequent itemsets from F_{k+1}. This is in accordance with Apriori principle where the subset of a frequent itemset must also be frequent.

To ensure the correctness of the algorithm, *for_checking()* can be used to run only on a small subset of a city's data specified by the variable *head*. *ls* and *freq_itemset* will be printed and the algorithm can be verified by manually counting the frequent itemsets.

2.1.2.5 Results obtained

```
CITY C:
support_threshold: 0.3
        k: 1
        k: 2
        k: 3
        frequent itemset(s) in city C for minsup of 975:
        1.      Park, Home Appliances,
        2.      Park, Building Material,
        3.      Transit Station, Elderly Care Home,
        4.      Transit Station, Home Appliances,
        5.      Transit Station, Hair Salon,
        6.      Transit Station, Accountant Office,
        7.      Transit Station, Building Material,
        8.      Transit Station, Heavy Industry,
        9.      Real Estate, Building Material,
        10.     Home Appliances, Building Material,
        11.     Building Material, Heavy Industry,
        12.     Park, Transit Station, Real Estate,
```

The output snippet below shows the frequent co-appearing POIs in city C, for a minimum support threshold of 0.3. If the code for this task is run, it will return the results for all four cities for support thresholds of 0.15, 0.2, 0.3 and 0.4.

2.2 Mining Sequential Patterns

In this section, our objective is to analyse the movement sequences of residents in each city to uncover common patterns of mobility through sequential pattern mining.

2.2.1 Data preprocessing

In this section, we cleaned our data by removing rows with invalid entries of coordinates x and y. Then, we combine our date and time into a datetime column. We drop intermediate columns and rename the columns for consistency. Lastly, we filter our data to only include the first 30 days of the data. We then save our processed data and into a file.

2.2.1.1 Load and decompress .gz files into a Dataframe

```python
# Function to load and decompress .gz files into a DataFrame
def load_compressed_csv(file_path):
    with gzip.open(file_path, 'rt') as gz_file:
        return pd.read_csv(gz_file)
```

2.2.1.2 Cleaning and preprocessing of *data process_city_data()* performs the following tasks:

```python
# Function to clean and preprocess the data
def process_city_data(city_key):
    # Load data from compressed file
    print(f"Loading data for {city_key}...")
    df = load_compressed_csv(data_paths[city_key])

    # Remove rows with invalid coordinates (-999)
    print("Cleaning data...")
    valid_data = df[(df['x'] != -999) & (df['y'] != -999)].copy()

    # Combine date ('d') and time ('t') into a 'tracked_at' datetime column
    valid_data['date'] = pd.to_datetime(valid_data['d'], format='%j', errors='coerce')
    valid_data['time_offset'] = pd.to_timedelta(valid_data['t'] * 30, unit='m')
    valid_data['tracked_at'] = valid_data['date'] + valid_data['time_offset']
    valid_data['tracked_at'] = valid_data['tracked_at'].dt.tz_localize('UTC')

    # Drop intermediate columns
    valid_data.drop(columns=['date', 'time_offset'], inplace=True)

    # Rename columns for consistency
    valid_data.rename(columns={'uid': 'user_id', 'x': 'longitude', 'y': 'latitude'}, inplace=True)

    # Filter data to only include the first 30 days (first month)
    print("Filtering data for the first 30 days...")
    start_date = valid_data['tracked_at'].min()  # Get the earliest date in the data
    end_date = start_date + pd.Timedelta(days=30)  # Set the cutoff for 30 days after the start date
    valid_data = valid_data[(valid_data['tracked_at'] >= start_date) & (valid_data['tracked_at'] < end_date)]

    # Save cleaned data
    output_file = Path(f"processed_data_{city_key}.csv")
    valid_data.to_csv(output_file, index=False)
    print(f"Processed data saved to {output_file}")
```

1. Load data for a specific city
2. Clean data by removing rows with invalid coordinates i.e. -999 values
3. Create a tracked_at timestamp by combining date and time columns
4. Drop intermediate columns (date and time_offset)
5. Rename columns for clarity (e.g., uid ⟶ user_id, x ⟶ longitude)
6. Filter data to include the first 30 days only
7. Save the cleaned data to a new CSV file

2.2.1.3 Generating triplegs

create_triplegs() performs the following tasks:

```python
# Function to preprocess data and generate triplegs
def create_triplegs(city_key):
    # Preprocess data
    process_city_data(city_key)

    # Load preprocessed data into positionfixes
    preprocessed_file = Path(f"processed_data_{city_key}.csv")
    print(f"Reading preprocessed data for {city_key}...")
    positionfixes = ti.read_positionfixes_csv(preprocessed_file)

    # Generate staypoints from positionfixes
    print("Identifying staypoints...")
    positionfixes, staypoints = positionfixes.as_positionfixes.generate_staypoints(
        method='sliding',
        dist_threshold=1,         # Distance threshold in meters
        time_threshold=90,        # Time threshold in minutes
        gap_threshold=300,        # Gap threshold in minutes
        distance_metric='haversine',
        include_last=True,
        exclude_duplicate_pfs=True,
        print_progress=True,
        n_jobs=-1
    )

    # Generate triplegs between staypoints
    print("Generating triplegs...")
    positionfixes, triplegs = ti.preprocessing.generate_triplegs(
        positionfixes, staypoints, method='between_staypoints', gap_threshold=90
    )

    # Export triplegs to CSV
    triplegs_file = Path(f"triplegs_{city_key}.csv")
    export_triplegs_to_csv(triplegs, triplegs_file, index=False)
    print(f"Triplegs exported to {triplegs_file}")
```

1. Preprocess data: Calls *process_city_data* to clean and format the raw position data
2. Load the preprocessed data into the positionfixes object
3. Generate staypoints: Identifies locations where a user stays for a certain duration and distance
4. Generate triplegs: Identifies trips made between consecutive staypoints
5. Export triplegs to CSV: Saves the generated triplegs to a CSV file for further analysis

Ultimately, this function preprocesses data, generates staypoints, creates triplegs, and exports the triplegs to a CSV file.

2.2.1.4 Converting triplegs to sequences

```python
def get_tripleg_sequences(triplegs_file):
    """
    Extracts the tripleg sequences from the triplegs CSV file by parsing the LINESTRING column.
    Each row contains a trip, and each trip is a sequence of coordinates.
    """
    # Read the triplegs CSV file into a pandas DataFrame
    triplegs = pd.read_csv(triplegs_file)

    sequences = []
    for _, row in triplegs.iterrows():
        # Extract coordinates from the LINESTRING column
        coordinates = extract_coordinates_from_linestring(row['geom'])
        sequences.append(coordinates)  # Each tripleg becomes a sequence of (x, y) pairs
    return sequences
```

After obtaining our triplegs, we iterate through each row of the dataFrame and extract the LINESTRING data from the 'geom' column. Then, we parse LINESTRING data to extract coordinates using a helper function *extract_coordinates_from_linestring()* and store the extracted coordinates (trip sequences) in a list. Finally, each tripleg becomes a sequence of (x, y) pairs.

2.2.2 Data analysis

From the triplegs file, we extracted the sequences and performed the Generalised Sequential Pattern (GSP) algorithm to obtain the frequent subsequences.

2.2.2.1 Generate candidate subsequences

```python
def generate_candidate_subsequences(frequent_seqs, length):
    """Generate candidate subsequences of a given length from frequent subsequences"""
    candidates = set()
    for seq1 in frequent_seqs:
        for seq2 in frequent_seqs:
            # Only combine sequences of length (length - 1) to create length 'length' subsequences
            if len(seq1) == length - 1 and len(seq2) == length - 1:
                # Combine the sequences if the last item of seq1 matches the first item of seq2
                if seq1[-1] == seq2[0]:
                    candidates.add(seq1 + seq2[1:])
    return candidates
```

generate_candidate_subsequences() takes in frequent sequences and iteratively combines sequences and generates candidate subsequences. This aims to create potential subsequences that might be frequent in future iterations of the mining process.

2.2.2.2 Count subsequences

```python
def count_subsequences(transactions, candidates):
    """Count the occurrences of subsequences in transactions"""
    subseq_count = defaultdict(int)
    for trans in transactions:
        for subseq in candidates:
            if is_subsequence(subseq, trans):
                subseq_count[subseq] += 1
    return subseq_count
```

count_subsequences() counts the occurrences of subsequences in transactions.

2.2.2.3 Generalized Sequential Pattern (GSP)

Our *generalized_sequential_pattern_mining()* performs several tasks:

```python
def generalized_sequential_pattern_mining(raw_transactions, min_support):
    # Step 1: Preprocess transactions to extract only item values (simplify raw data)
    transactions = [[item for item in trans] for trans in raw_transactions]

    # Step 2: Find frequent 1-item subsequences (pairs as items)
    # single_items = defaultdict(int)
    # for trans in transactions:
    #     print(trans)
    #     for item in trans:
    #         single_items[(item,)] += 1
    #         print(item)

    single_items = defaultdict(int)

    for trans in transactions:
        # Generate all possible 2-item subsequences (pairs) without duplicates and unordered
        for pair in combinations(trans, 2):
            # Ensure no duplicate items in the pair
            if pair[0] != pair[1]:  # This check ensures no duplicate items in a pair
                single_items[frozenset(pair)] += 1  # Use frozenset to ignore order and handle unique pairs

    # Step 3: Filter 1-item subsequences by min_support
    F1 = {seq: count for seq, count in single_items.items() if count >= min_support}

    # Initialize frequent subsequences with F1
    frequent_subsequences = F1.copy()

    k = 2  # Starting with length 2 sequences

    # Loop to mine subsequences of increasing length k
    while F1:
        # Step 4: Generate candidate k-sequences from Fk-1 (previous frequent subsequences)
        candidate_k_seqs = generate_candidate_subsequences(list(F1.keys()), k)
        # print(k,list(F1.keys()))
        # print()
        # Step 5: Count occurrences of candidate k-sequences in transactions
        subseq_count = count_subsequences(transactions, candidate_k_seqs)

        # Step 6: Filter candidate subsequences by min_support
        Fk = {subseq: count for subseq, count in subseq_count.items() if count >= min_support}

        # Add Fk to the set of frequent subsequences
        frequent_subsequences.update(Fk)

        # If Fk is empty, stop the mining process
        if not Fk:
            break

        # Update F1 for the next iteration (Fk becomes Fk-1)
        F1 = Fk
        k += 1  # Increase subsequence length for the next iteration

    return frequent_subsequences
```

1. Extract item values from raw transactions
2. Identify frequent 2-item subsequences and exclude those that do not meet the min_support (100) threshold
3. Iteratively generate subsequences of increasing length, followed by counting the occurrences of candidates in transactions and filtering those candidates that do not meet the min_support (100) threshold
4. Return all frequent subsequences: returns a dictionary of all frequent subsequence

2.2.2.4 Output results

Top 10 frequent subsequences for city A:

```
Frequent 2-item Subsequences (Sorted by Support)
================================================
Subsequence: frozenset({(134.0, 77.0), (135.0, 77.0)}), Support: 22859
Subsequence: frozenset({(135.0, 77.0), (135.0, 78.0)}), Support: 15009
Subsequence: frozenset({(135.0, 76.0), (135.0, 77.0)}), Support: 13061
Subsequence: frozenset({(135.0, 82.0), (134.0, 82.0)}), Support: 10100
Subsequence: frozenset({(128.0, 14.0), (129.0, 13.0)}), Support: 9099
Subsequence: frozenset({(135.0, 82.0), (135.0, 81.0)}), Support: 8621
Subsequence: frozenset({(128.0, 14.0), (129.0, 14.0)}), Support: 7969
Subsequence: frozenset({(103.0, 120.0), (103.0, 119.0)}), Support: 7878
Subsequence: frozenset({(135.0, 82.0), (135.0, 83.0)}), Support: 7314
Subsequence: frozenset({(106.0, 106.0), (106.0, 107.0)}), Support: 7314
```

Top 10 frequent subsequences for city B:

```
Frequent 2-item Subsequences (Sorted by Support)
================================================
Subsequence: frozenset({(80.0, 95.0), (80.0, 96.0)}), Support: 11569
Subsequence: frozenset({(80.0, 92.0), (80.0, 93.0)}), Support: 9865
Subsequence: frozenset({(79.0, 93.0), (79.0, 94.0)}), Support: 6702
Subsequence: frozenset({(79.0, 92.0), (79.0, 93.0)}), Support: 6601
Subsequence: frozenset({(79.0, 93.0), (80.0, 93.0)}), Support: 6244
Subsequence: frozenset({(77.0, 101.0), (76.0, 101.0)}), Support: 6019
Subsequence: frozenset({(79.0, 92.0), (80.0, 92.0)}), Support: 5979
Subsequence: frozenset({(74.0, 81.0), (75.0, 81.0)}), Support: 5412
Subsequence: frozenset({(79.0, 93.0), (80.0, 92.0)}), Support: 5327
Subsequence: frozenset({(73.0, 99.0), (72.0, 99.0)}), Support: 4768
```

Top 10 frequent subsequences for city C:

```
Frequent 2-item Subsequences (Sorted by Support)
================================================
Subsequence: frozenset({(26.0, 152.0), (26.0, 153.0)}), Support: 11302
Subsequence: frozenset({(25.0, 153.0), (26.0, 153.0)}), Support: 9381
Subsequence: frozenset({(25.0, 153.0), (24.0, 153.0)}), Support: 8254
Subsequence: frozenset({(25.0, 152.0), (25.0, 153.0)}), Support: 7770
Subsequence: frozenset({(25.0, 153.0), (25.0, 154.0)}), Support: 6390
Subsequence: frozenset({(26.0, 153.0), (24.0, 153.0)}), Support: 5700
Subsequence: frozenset({(23.0, 153.0), (24.0, 153.0)}), Support: 5590
Subsequence: frozenset({(26.0, 153.0), (26.0, 154.0)}), Support: 5549
Subsequence: frozenset({(25.0, 153.0), (26.0, 152.0)}), Support: 5545
Subsequence: frozenset({(24.0, 152.0), (24.0, 153.0)}), Support: 5437
```

Top 10 frequent subsequences for city D:

```
Frequent 2-item Subsequences (Sorted by Support)
================================================
Subsequence: frozenset({(101.0, 102.0), (101.0, 103.0)}), Support: 2383
Subsequence: frozenset({(150.0, 98.0), (151.0, 98.0)}), Support: 2206
Subsequence: frozenset({(99.0, 33.0), (98.0, 33.0)}), Support: 2175
Subsequence: frozenset({(101.0, 102.0), (100.0, 102.0)}), Support: 2060
Subsequence: frozenset({(175.0, 43.0), (174.0, 43.0)}), Support: 2038
Subsequence: frozenset({(107.0, 101.0), (107.0, 102.0)}), Support: 2035
Subsequence: frozenset({(67.0, 100.0), (67.0, 101.0)}), Support: 1968
Subsequence: frozenset({(150.0, 98.0), (152.0, 97.0)}), Support: 1872
Subsequence: frozenset({(78.0, 157.0), (78.0, 158.0)}), Support: 1861
Subsequence: frozenset({(126.0, 127.0), (126.0, 128.0)}), Support: 1843
```

2.3 LSTM to Predict Next Location

2.3.1 The Next Step

The previous tasks provided a foundation for mobility analysis. While we have already seen common movement subsequences using the GSP algorithm, we are not yet able to predict the next step of a person's journey in a given city based on their past travels. This is why the primary goal of this section is **to predict a user's next location based on their historical trajectory**, which is a natural extension of the previous tasks.

2.3.2 Choice of LSTM

The decision to use an **LSTM (Long Short-Term Memory) network** stems from its ability to model sequential data effectively, making it particularly well-suited for mobility prediction tasks. The parts of an LSTM Unit include: a Forget Gate, Input Gate, Memory Cell, and Output Gate. The Forget Gate uses a sigmoid activation function (assigns a value between zero and one) to past information and decides what to discard. The Input Gate, similarly, decides what to keep. The Memory Cell contains a combined state from the Input and Forget Gates. The Output Gate then selectively (using a sigmoid function) outputs parts of the Memory Cell. This ability to selectively retain, update, and output relevant information makes LSTM networks particularly powerful for sequential prediction tasks like mobility prediction.

2.3.3 Approach and Implementation

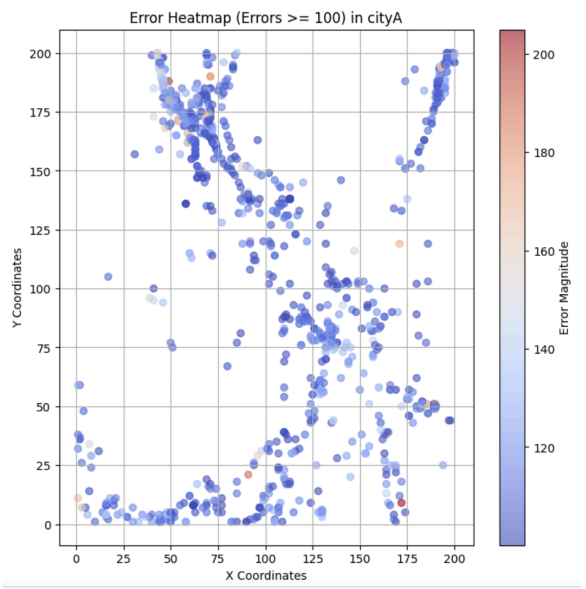
Each city was assigned a separate model because each city may have its own nuances in movement, i.e. varying distributions of POIs. After preprocessing (generating triplegs from raw trajectory data), there were sequences of varying length for each user and start-time. LSTMs require the same size of inputs, so many **subsequences of fixed length** (supposed 3, to begin with, in case most triplegs were short) would have to be generated. Every time a **subsequence (training input)** was generated, its **next location (training target)** was stored in a corresponding "targets" array. After a 80-20 train-test split of subsequences and corresponding target next locations, a single **LSTM layer with 64 units** was built with a loss function of Mean Squared Error (MSE). The model was trained with 20 epochs and a **batch size of 64**, and was validated with 20% of the train data in each epoch. Finally, a model was saved for each city and tested against the test set of subsequences and their targets.

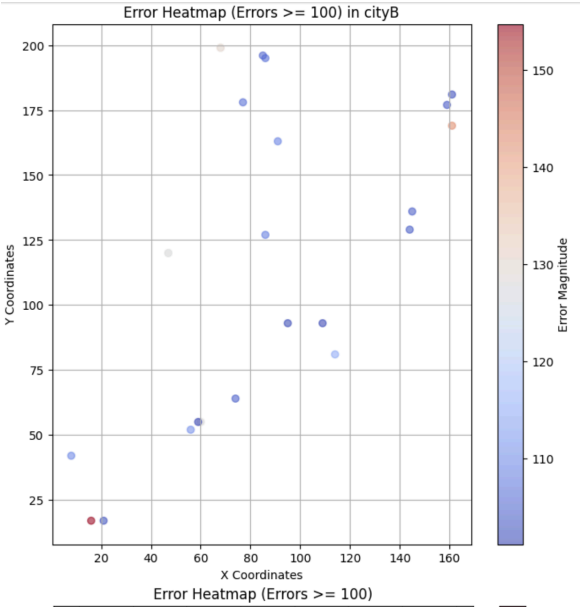
```
for seq in sequences:
    for i in range(len(seq) - sequence_length):
        input_sequences.append(seq[i:i + sequence_length])  # Input sequence
        targets.append(seq[i + sequence_length])  # Next location target

input_sequences = np.array(input_sequences)
targets = np.array(targets)
```

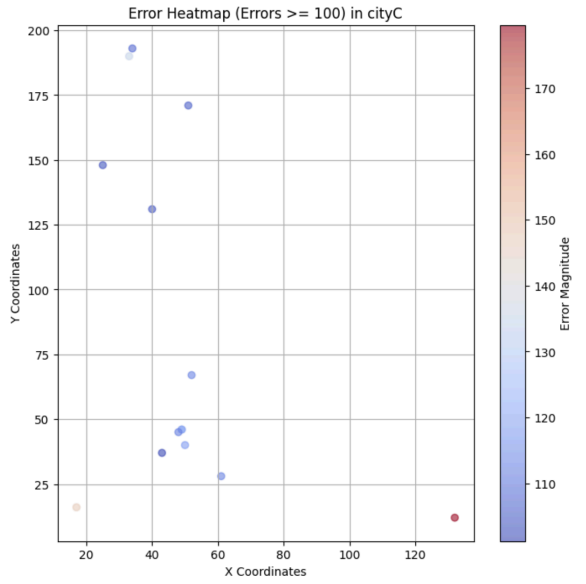
2.3.4 Results

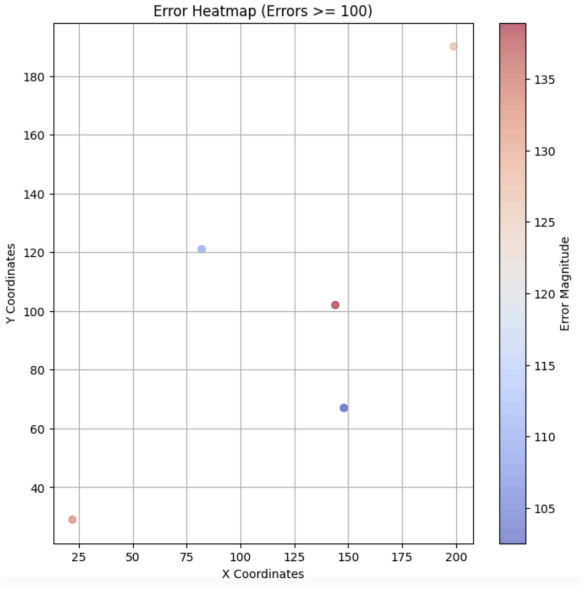
After evaluation on our test set, each city's model's MSE was reported as follows:
City A: 76.84; City B: 58.36; City C: 53.59; City D: 57.58;

Scatter plots for each city are shown to indicate errors (distance of prediction from target) greater than 100:









2.3.5 Hyperparameter Tuning

A subsequence length of 3 was initially chosen due to uncertainty of length of sequences-- if a greater subsequence length had been chosen, triplegs with 3 locations or less would have been skipped during training. Therefore, a histogram was plotted for each city to analyse the

distribution of tripleg lengths. For all of them, the vast majority contained below 4 staypoints, so a small seq_length would make sense. However, since context is important for LSTMs, more historical travel data may be worth sacrificing the quantity of training subsequences-- a quantity vs. quality tradeoff. Therefore, varying **subsequence lengths** were analysed, and the best ones were recorded based on their MSE values. Also, the **batch size** and **number of LSTM Units** were varied to find the lowest MSEs possible without overfitting, but large numbers of LSTM Units and Batch Sizes were always found to be overfitting, as the "Validation Loss" reported in each epoch of training started to fluctuate and increase for the larger combinations.

Contributions

Name	Matriculation Number	Contributions
Lim Jun Yu	U2222159F	Part 1, Report Writing
Poh Zi Jie Isaac	U2140416E	Part 2, Report Writing
Dexter Voon Kai Xian	U2120267F	Part 2, Report Writing
Mukherjee Tathagato	U2120365K	Part 3, Report Writing