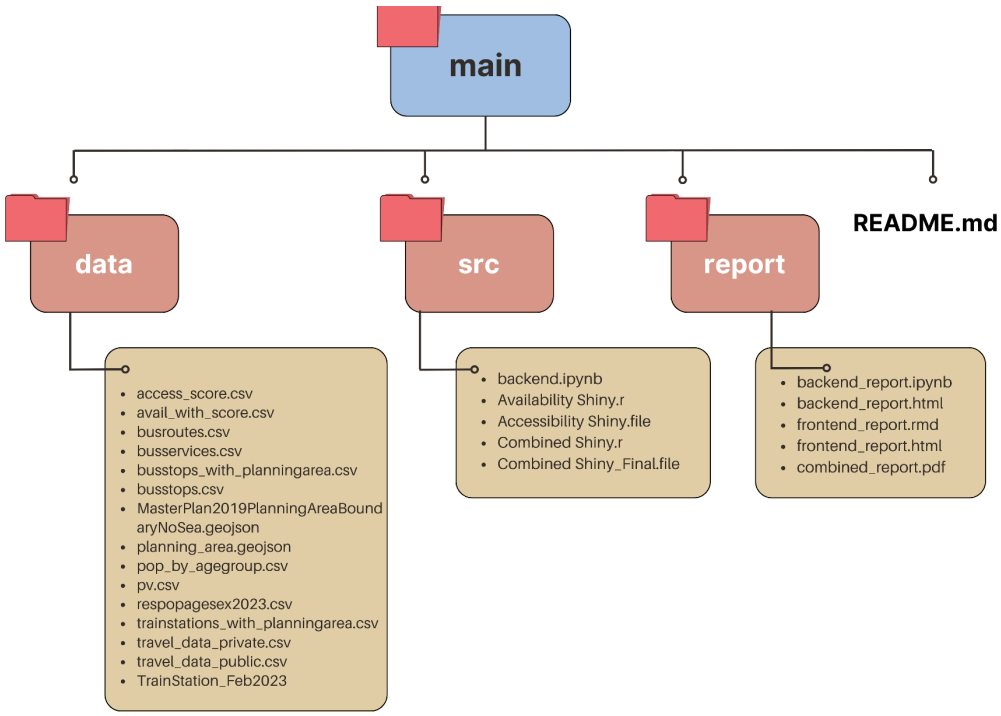
Analysing Connectivity of Public Transport in Singapore

Bernice Ong Hwee Yee Gao Yuchen Luo Xinming Tiffany Irene Prasetio

Below is an overview of the repository structure:



Shiny App Code Execution and Testing

You can find the code to run the Shiny App in the "src/Combined Shiny_Final" directory. Please ensure that you have the necessary libraries and packages ('shiny`, `leaflet`, `sf`, `dplyr`, `DT`) installed before running the code. Load the datasets by executing the code provided under the #load the data section. The datasets used are listed in the "data" folder. Open RStudio or any other environment where you can run R code to execute the file.

To test the Shiny application, which involves both a user interface (UI) and server logic components, you can follow these steps:

1. Test UI Elements:
   - Interact with the UI elements like sliders and buttons to ensure they modify the application state as expected. For example, adjust the sliders and observe if the map and data tables update accordingly.
   - Use the action buttons to generate maps and ensure the maps display the correct data and legends.
2. Validate the calculations:
   - Verify that the application correctly calculates the total weights and updates inputs when weights are changed. For instance, when you modify one weight slider, the application should adjust others to ensure the sum remains equal to 1.
3. Review error handling:
   - Test how the application handles errors, such as when the sum of weights does not equal 1. Ensure that the error notifications are clear.

By following these steps, you can comprehensively test the Shiny application to ensure it performs as intended and provides a robust user experience.

Frontend Documentation

This document provides a comprehensive guide to the Shiny application interface designed for analyzing connectivity in 55 regions of Singapore.

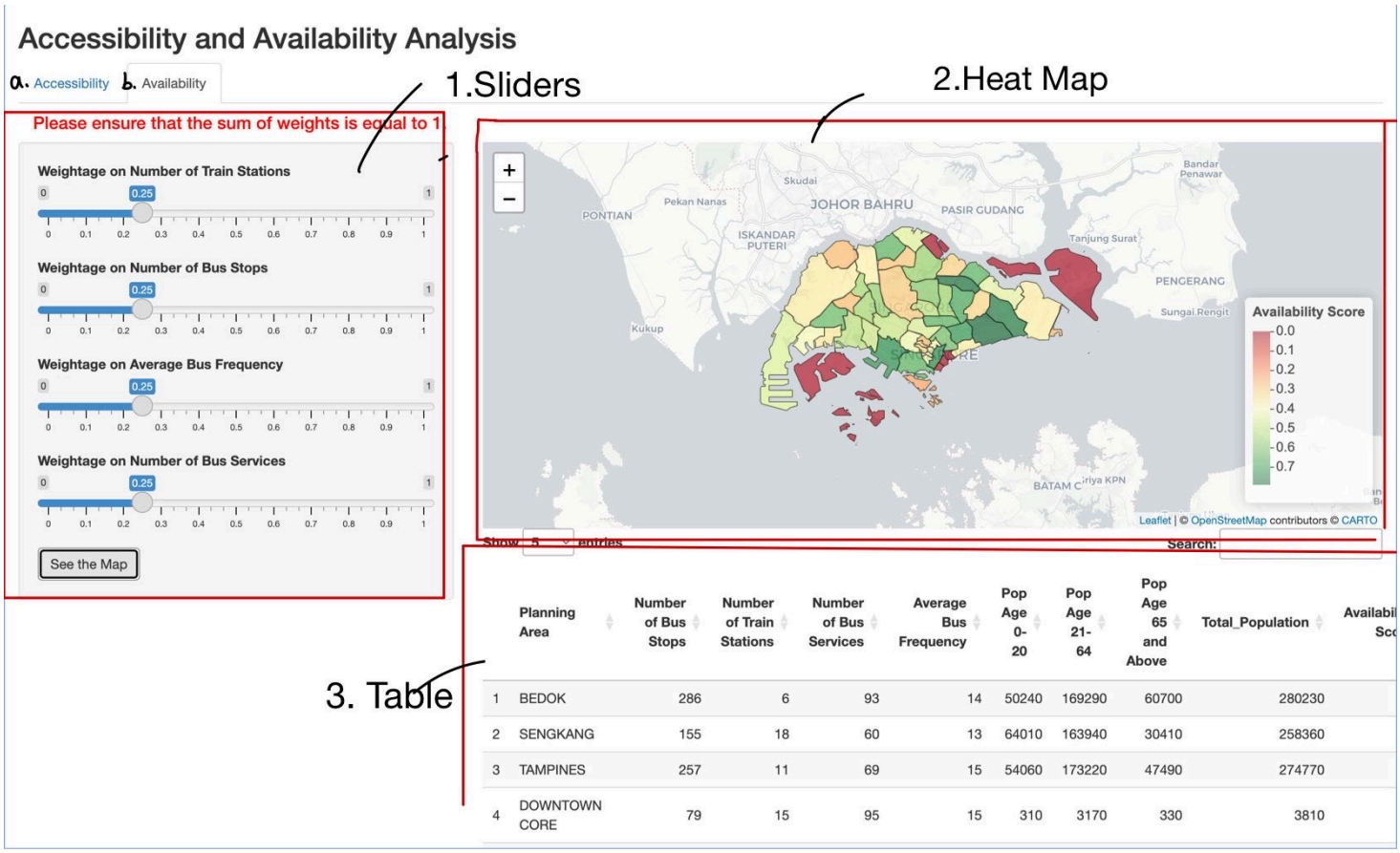
What does this app do?

The app is intended to help people who rely heavily on public transport and intend to buy or rent a new house. For this group of people, transportation is one of their key considerations when buying or renting a house. They would like to have some easy and comprehensive way to know about connectivity in different areas in Singapore. Our app offers them the ease and convenience to find the information that they are interested in right at their fingertips.

Comparing with looking through tons of data online, we believe the more intuitive way for users to understand the connectivity in each area is through our interactive and comprehensive visualization. Our backend team used the most updated and reliable sources for transportation from the OneMap API and location data to provide users with the most accurate information. To assess connectivity, we utilized two metrics: accessibility and availability.

Application Overview

In our design, simplicity and ease of use are key priorities. Our frontend interface was primarily built using the Shiny app in R.



	Planning Area	Number of Bus Stops	Number of Train Stations	Number of Bus Services	Average Bus Frequency	Pop Age 0-20	Pop Age 21-64	Pop Age 65 and Above	Total_Population
1	BEDOK	286	6	93	14	50240	169290	60700	280230
2	SENGKANG	155	18	60	13	64010	163940	30410	258360
3	TAMPINES	257	11	69	15	54060	173220	47490	274770
4	DOWNTOWN CORE	79	15	95	15	310	3170	330	3810

Above is an overall view of the interface. To prevent users from feeling overwhelmed by the abundance of information, we've organized the interface to display only the most essential details. As you can see, there are three main sections in the interface: (1) the input slider section, (2) the map section, and (3) the table view section for both the (a) accessibility and (b) availability tabs. We will explain each of them in detail in the next section.

Package Used and Their Functions

Package Name	Function
shiny	Build interactive web applications
leaflet	Create interactive maps
sf	Manipulate spatial data
dplyr	Data manipulation and transformation
DT	Create interactive tables

Import Data

The data we used are the data being engineered and cleansed by the back-end team and are ready for front-end to use. Below is the overview of the datasets we used and their functions. All these files are under the src directory of our repo.

Dataset Name	Function
access_score.csv	Generate accessibility score
avail_with_score.csv	Generate availability score
planning_area.geojson	Plotting the spatial map with 55 planning regions being labelled
pop_by_agegroup.csv	Value-added feature of demographic distribution in different areas

Functions and Relevant Features Involved in the Front-end

We used two dimensions to interpret the concept of connectivity, namely availability and connectivity.

The **availability** of a region refers to the fundamental public transportation infrastructure available to residents in that area. For this aspect, we mainly consider four different factors:

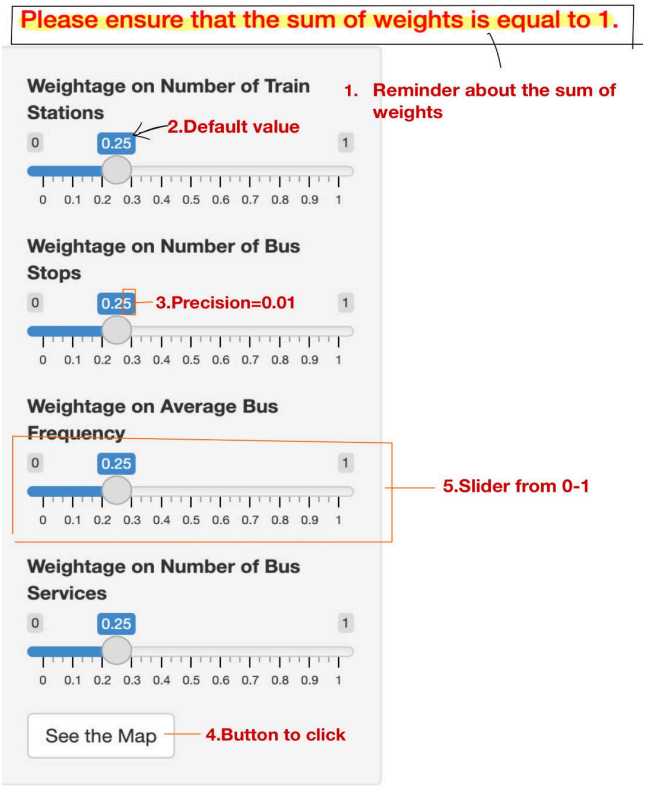
1. Number of train stations in the region
2. Number of bus stops in the region
3. Number of bus routes in the region
4. Average frequency of buses (in minutes) in each region (Frequency of Dispatch for Bus Services in the region)

The **connectivity** of a region refers to how accessible the destinations are using public transports from the area that people are living in. We quantify this metric by assessing the differences in travel time (in minutes) between utilizing public and private transport to three representative places: CBD (Business) ION Orchard (Leisure) and SGH (Healthcare). They are selected based on the online resources stating that they are the most popular locations under each category.

Our UI and server logic are quite similar and consistent for availability and accessibility. Here, we would like to use availability to illustrate how each part works.

1. Slider Section

We understand that users may have different priorities for various metrics. Thus, we incorporate a customization function via the slider section. Now, we will explain main features of the slider section and explain how to realize them using code.Below is a screenshot with label of the slider section in our interface.



Slider Input Widget with Features being Labelled

Users can adjust the weights assigned to different factors from 0 to 1(5) with a precision of 0.01(3) based on their needs. For instance, a user who often takes the MRT to their workplace can give a larger weight to the number of train stations in the area. The default value of the weights is 0.25

across all sliders (2), which will be convenient for the users if they want to treat all the factors equally.

We achieved this using under the sidebarPanel of UI. We created three *sliderInput()* functions for accessibility (three factors to compute the score) and four sliderInput functions for availability (four factors to compute the score).Inside each sliderInput function, we set the relevant variable, label, min and max value and default value as well as the precision using step argument. Below is an example code to do this:

```
sidebarPanel(sliderInput('trainWeight', "Weightage on Number of Train Stations ", min = 0, max = 1, value = 0.25,
step = 0.01)
```

Once the user key in all the weights, they can click on the see the map button which will generate the map and table for them(4). We employed actionButton function under UI to do it.

Below is the code to calculating the score for availability. This code calculates the score based on weighted factors in the *"avail_score_with_pop"* dataset using values from sliders in a Shiny application. Each factor is multiplied by a corresponding weight to get the weighted sum.

```
avail_score_with_pop$score <- with(avail_score_with_pop, input$trainWeight * num_trainstations_score + input$buss
topWeight * num_busstops_score + input$busfreqWeight * avg_bus_freq_score + input$busserviceWeight * num_busservi
ces_score)
```

One essential condition is that the sum of the weights must be equal to 1 for consistency of comparison(1). To inform our users of this, we add a red line reminder above the slider stating, *"Please ensure that the sum of weights is equal to 1."* We realize this by adding column function under each *tabpanel* under the UI session. We set the first argument of the column function to be 12 to specify the width of the column. The number 12 indicates that the content should span the entire width of the tab panel. Then, we set the second argument to be h4 to create level 4 heading with the text we want to display. In order to make the reminder more obvious, we set the color of the text to be red by doing *"style ="color: red;"*.

If the total weight is incorrect(not 1), a pop-up will alert the users to remind them. We employed *showNotification* function here to achieve this.The duration of the notification is set to 5 seconds, and the type of notification is set to "error" to visually distinguish it as an error message.

```
if (total_weight != 1) {
      showNotification(paste("Total weightage must sum up to 1. Current total is ", total_weight),duration = 5,
        type = "error")
```

To reduce the amount of math that users need to do, we design our slider such that the last slider will adjust its value based on the previous values inputted by our users. This was done via observe function under server section as shown below. The *updateSliderInput()* function to update the value of the slider input. The new value is set to the difference between 1 (the target sum of weights) and the current *"total_weight".*

```
observe({
    total_weight <- input$trainWeight + input$busstopWeight + input$busfreqWeight
    updateSliderInput(session, "busserviceWeight", value = 1 - total_weight)
  })
```

2. Interactive Heat Map

After being assigned weights of each factor by users, we will be able to generate the availability score. However, comparing a list of scores for different regions is not easy and engaging. We decided to build a heat map which can vividly reflects the availability and accessibility in different areas.

```
 output$availMap <- renderLeaflet({
        leaflet() %>%
          addProviderTiles("CartoDB.Positron") %>%
          addPolygons(data = merged_data_ava,
                      fillColor = ~colorNumeric("RdYlGn", domain = merged_data_ava$score)(score),
                      fillOpacity = 0.7,
                      color = "black",
                      weight = 1,
                      label = ~paste(planning_area, ": ", round(score, 2),
                                     "Total Population:", total_pop),
                      labelOptions = labelOptions(direction = "auto", permanent = FALSE)) %>%
          addLegend(pal = colorNumeric("RdYlGn", domain = merged_data_ava$score),
                    values = merged_data_ava$score,
                    title = "Availability Score",
                    position = "bottomright")
      })
```

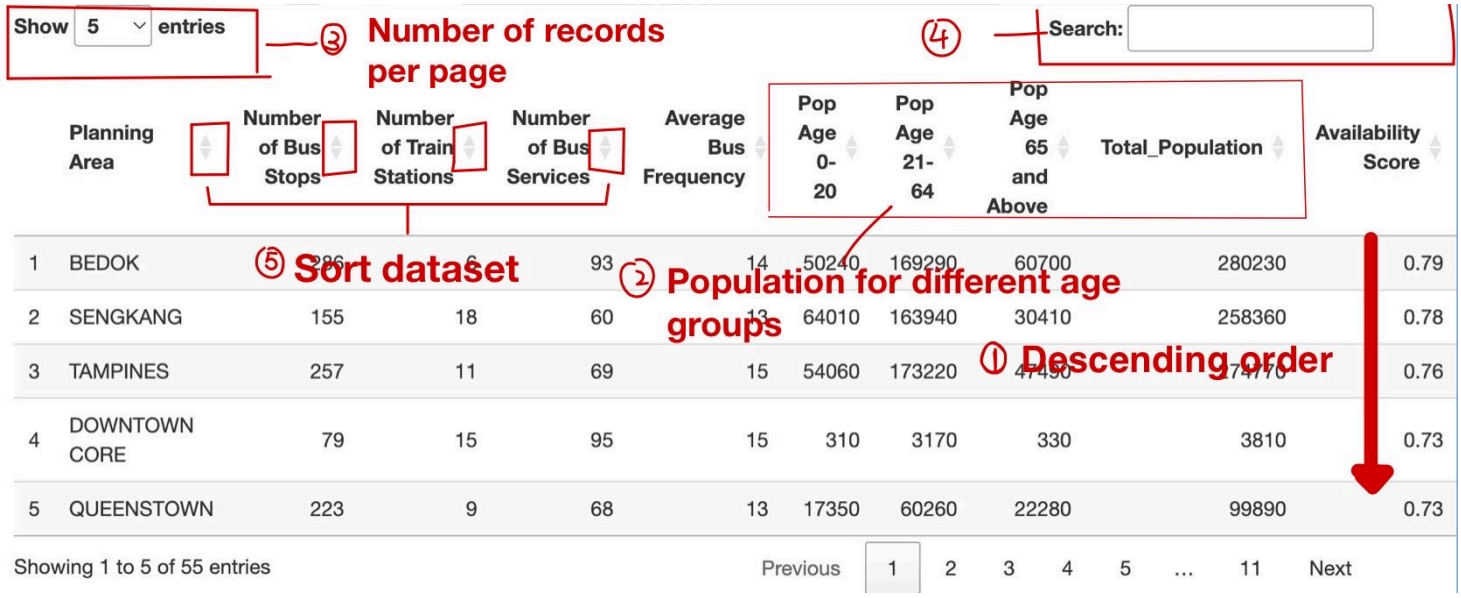
We utilized the *leaflet()* function in R along with *GeoJSON data* (from the src folder) to create interactive Leaflet maps. The *"CartoDB.Positron"* tile layer is added as the base map using *addProviderTiles()*.Polygons representing geographical areas are added to the map using *addPolygons()*, with fillColor determined by a color scale based on the "score" variable from *"merged_data_ava"*.

We chose a color gradient from green to red for the heat maps, where green signifies high scores and red signifies lower scores. Labels are added to each polygon showing the planning area name, score, and total population. When the user hovers the mouse over any specific region, they will instantly see the region's name, availability score, and the number of residents living in this area.

```
avail_score_with_pop <- merge(avail_score, pop_by_agegroup, by.x = "planning_area", by.y = "PA")
```

We incorporated the population of each region by merging the *"avail_with_score.csv"* with the *"pop_by_agegroup.csv"* (from the src folder) by matching values in the *"planning_area"* column of *"avail_score"* with values in the *"PA"* column of *"pop_by_agegroup"*. Population in each region is an important consideration when determining the availability of public transport in different areas. It is possible for a region to have a small number of transportation infrastructure while also having a small population. In such cases, even if the availability score is low, it may not pose a significant issue, as fewer people will use public transport.

3. Table View



Interactive Table with features Being Labelled

Beyond the heat maps, we employed the *renderDT* function udner the DT package to create interactive data table with detailed numerical data for each region. This feature empowers our users to explore specifics and compare different areas easily. Regions are neatly listed in the table, sorted by score in descending order(1). We used *arrange(desc(score))* to realize this.

While providing a score for different regions is an easy way to access its connectivity, some users might want the exact data for each factor to get the score. That's why our tables offer exact data for each factor contributing to the score. We selected the most relevant factors as shown below by using *select()* function under dplyr package. We manipulated the data before it is displayed under the server section.For instance, we mutated the score column to only keep 2 decimal places. We also used *colnames()* function to change the column name to make them more intuitive for our users.

```
colnames(sorted_avail_score) <- c("Planning Area", "Number of Bus Stops ", "Number of Train Stations", "Number of Bus Services","Average Bus Frequency" ,"Pop Age 0-20" ,"Pop Age 21-64", "Pop Age 65 and Above","Total_Population","Availability Score")
```

Besides the factors mentioned above, we also include the population at different age groups in the availability table(2).We understand that demographic distribution is also one crucial aspects that people always take into account when they rent or buy a house. Incorporating this useful information will definitely be helpful for them.

```
datatable(sorted_avail_score, options = list(pageLength = 5)
```

The code above set the default rows being displayed per page in the table to be 5. Our users can also adjust the number of rows being displayed per page based on their own preference(3).

Moreover, the search function at the top right hand corner of the table will allow our users to search and get the information of the community they are interested(4). This helps to avoid the hassle of manually finding the information throughout the entire table. Our users can also sort the data by a specific column by simply clicking on the column header which is one excellent function of DT table(5).

All these features make the data table interactive and convenient for our users. Last but not least, we use *shinyApp(ui = ui, server = server)* to run the code.

Backend Documentation

Our backend codes can be found can be found in `../src/backend.ipynb`.

The goal of the backend analysis is to construct a pipeline that calculates the **connectivity score** for each of the 55 planning areas in Singapore. Connectivity is broken down into:

1. **Availability**

- Number of train stations in the region
- Number of bus stops in the region
- Number of bus routes in the region
- Average frequency of buses (min) in each region (Frequency of Dispatch for Bus Services in the region)

2. **Accessibility**

- Differences in Travel Time (min) between taking Public and Private Transport (car) to:
  - CBD (Business)
  - ION Orchard (Leisure)
  - SGH (Healthcare)
- These three destinations are chosen because they represent three essential aspects of an ordinary resident's life.

How to use the scores

To use the constructed scores, simply load the files as shown below.

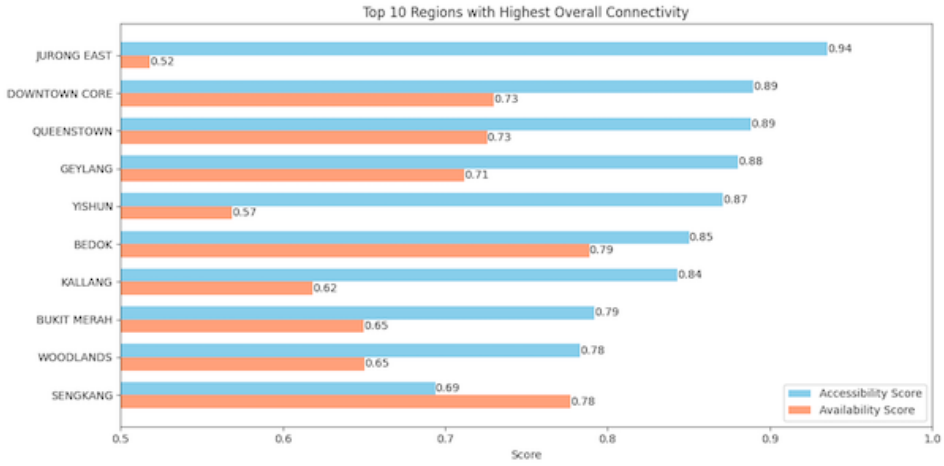
In [ ]:
```
import pandas as pd

avail_with_score = pd.read_csv('../data/avail_with_score.csv')
access_with_score = pd.read_csv('../data/access_score.csv')
```

Example Use Case: Top 10 Regions with Highest Overall Connectivity

We can combine both scores to analyse the overall connectivity of regions by taking a simple average of all metric scores.

In [ ]:
```
# merge both availability and accessibility scores
total_score = pd.merge(avail_with_score, access_with_score, on='planning_area', how='left', \
                       suffixes=('_avail', '_access'))
total_score['total_score'] = (total_score['total_score_avail'] + total_score['total_score_access']) / 2
total_score = total_score.sort_values(by='total_score', ascending=False).head(10)
```

We plot the Top 10 Regions with Highest Overall Connectivity below.



Future Enhancements

More metrics can be easily added to the current score formulations. Potential metrics include difference in travel time to other important destinations and number of bus interchanges in the region. We can also allow user to customise the departure time for travel time data collection, as departure time is a self-defined parameter in our data collection function.

Connectivity scores can also be constructed and analysed on a subzone level instead of the current planning area level by extracting geospatial data on subzones. However, this data is currently not available from OneMap API, hence we were unable to map the bus stops and train stations to their respective subzones.

Data Collection

Transport Services

We made API calls to Singapore Land Transport Authority (LTA) DataMall for data on:

- Bus Services: `../data/busservices.csv`
- Bus Routes: `../data/busroutes.csv`
- Bus Stops: `../data/busstops.csv`
- Passenger Volume by Bus Stops: `../data/pv.csv`

We downloaded the shapefile from LTA DataMall for data on:

- Train Stations: `../data/TrainStation_Feb2023/RapidTransitSystemStation.shp`

Note: `LTA_KEY` refers to your personal API Key from LTA DataMall.

Bus Services, Bus Routes, Bus Stops

We create a function for API calls to Bus Services, Bus Routes and Bus Stops where

- `service_type` : refers to a string denoting Bus Services, Bus Routes or Bus Stops, to be included in the request URL
- `skip_values` : refers to a list containing number of records to skip for each request and depends on how large the dataset is as there is a limit of 500 records per request.

In [ ]:
```
def get_pt_data(service_type, skip_values):
    base_url = "http://datamall2.mytransport.sg/ltaodataservice"

    service = []

    for skip in skip_values:
        endpoint_url = f"/{service_type}?$skip={skip}"
        resource_url = base_url + endpoint_url
        res = requests.get(resource_url, headers={"AccountKey": "LTA_KEY", "accept": "application/json"})
        res_list = res.json()
        df = pd.DataFrame(res_list['value'])
        service.append(df)

    service_df = pd.concat(service, ignore_index=True)
    return service_df
```

Passenger Volume by Bus Stops

This dataset had to be collected separately as additional parameters were required and the request returns a link to download the csv file instead.

In [ ]:
```
import requests

# Passenger volume: produces link to download csv
base_url = "http://datamall2.mytransport.sg/ltaodataservice"
endpoint = "/PV/Bus"
resource_url = base_url + endpoint
last_3_months = ['202402', '202401', '202312']
for month in last_3_months:
    query_params = {'Date': month}
    # Request data from the server
    res = requests.get(resource_url, headers={"AccountKey": "LTA_KEY", "accept": "application/json"}, \
                       params=query_params)
    res_list = res.json()
    print(res_list['value'])
```

Planning Areas

We made API calls to OneMap API to map each bus stop and train station to their respective planning areas using the location coordinates.
We also downloaded the MasterPlan 2019 geojson file directly from Data.gov.sg for the plotting of the boundaries on a heatmap.

From Data.gov.sg

We extract the Planning Area names from `Description` using regular expressions as it is rendered in HTML format.

The final file to be used: `../data/planning_area.geojson`

In [ ]:
```
import geopandas as gpd
import re

planning_area = gpd.read_file('../data/MasterPlan2019PlanningAreaBoundaryNoSea.geojson')

pattern = r'<th>PLN_AREA_N</th>\s*<td>(.*?)</td>'

names = []
# Iterate through each row and apply regex pattern
for index, row in planning_area.iterrows():
    html_data = row['Description']
    result = re.search(pattern, html_data)
    if result:
        #print(result.group(1))
        names.append(result.group(1))
    else:
        print("Pattern not found.")

planning_area['planning_area'] = names
planning_area = planning_area[['planning_area', 'geometry']]
# save locally
planning_area.to_file('../data/planning_area.geojson', driver='GeoJSON')
```

Extract Planning Area from OneMap API to join with Bus Stops and Train Stations

The data can be found in:

- Bus Stops: `../data/busstops_with_planningarea.csv`
- Train Stations: `../data/trainstations_with_planningarea.csv`

We create a function to loop through the data for Bus Stops and Train Stations with their coordinates to map to their planning areas where

- `pt_data`: refers to the dataframe for each public transport type, e.g. bus stops or train stations.

Notes:

- The `token` specified is your token from OneMap API.
- The exact `query_string` required may change, refer to the OneMap API website for more information.

In [ ]:
```
def get_planning_area(pt_data):
    pt_data['planning_area'] = ''

    domain = 'https://www.onemap.gov.sg/api/public/popapi/getPlanningarea?'
    token = 'xxx'
    headers = {"Authorization": token}
    incl_lat = 'latitude='
    incl_long = '&longitude='

    for index,row in pt_data.iterrows():

        if index % 100 == 0:
            print(index)

        lat = str(row['Latitude'])
        long = str(row['Longitude'])

        query_string = domain+incl_lat+lat+incl_long+long

        try:
            response = requests.request("GET", query_string, headers=headers, timeout=15)
            resp_list = response.json()

            pt_data.loc[index, "planning_area"] = resp_list[0]['pln_area_n']

        except:
            pt_data.loc[index, "planning_area"] = 'invalid'

    return pt_data
```

For train stations, we first convert the centroid coordinates in the shapefile to common latitude and longitude coordinates before running the loop to make API calls for the planning area of each train station.
Note that you may need to swap the order in the `transform()` line in the future due to changes in the PyProj library.

- Example: `lon, lat = transform(wgs84, svy21, svy21_x, svy21_y)`

In [ ]:
```
import geopandas as gpd
from pyproj import Proj, transform

# Read the shapefile
shape = gpd.read_file("../data/TrainStation_Feb2023/RapidTransitSystemStation.shp")

# Calculate centroid coordinates
shape['centroid_y'] = shape.geometry.centroid.y
shape['centroid_x'] = shape.geometry.centroid.x

# Define the SVY21 projection (EPSG:3414)
svy21 = Proj(init='EPSG:3414')

# Define the WGS84 projection (EPSG:4326)
wgs84 = Proj(init='EPSG:4326')

# Define the SVY21 coordinates (example values)
svy21_x = shape['centroid_x']
svy21_y = shape['centroid_y']

# Perform the coordinate transformation
lon, lat = transform(svy21, wgs84, svy21_x, svy21_y)

# Append the latitude and longitude coordinates to the dataframe
location = pd.DataFrame({'latitude': lat, 'longitude': lon})
trainstations = pd.concat([shape, location], axis=1)
```

Travel Time

We made API calls to Google Maps Platform for data on travel time by `transit` (public mode) or `driving` (private mode) from each of the 55 residential areas in Singapore to the three destination of interests: CBD area, ION Orchard and Singapore General Hospital, with departure time set to the Monday 5pm.

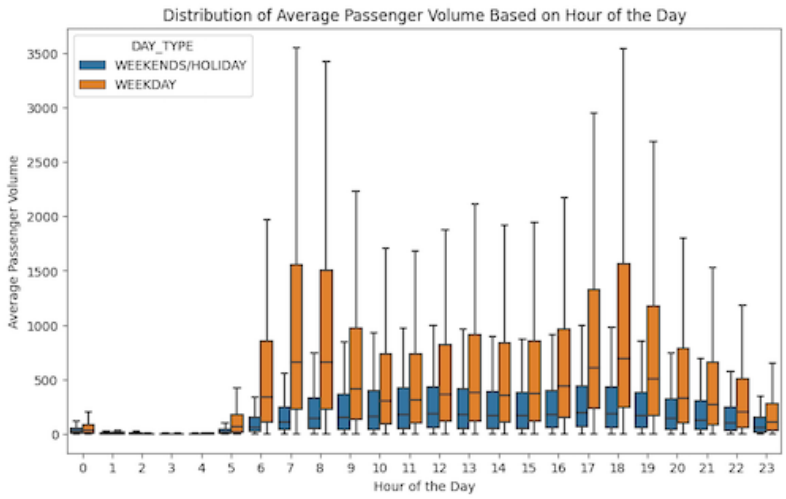
Note:

- `GMAP_KEY` refers to your personal API Key from Google Maps Platform.
- `unix_timestamp` refers to the timestamp for the nearest Monday 5pm, singapore time, from the time of data collection.

The data are stored in the following files:

- Public Transport Travel Times: `../data/travel_data_public.csv`
- Private Transport Travel Times: `../data/travel_data_private.csv`

From our exploratory data analysis using Passenger Volume by Bus Stops, we find that the Highest Average Volume is during a weekday at 7-8am and 5-6pm likely due to people commuting to and fro work. Hence, we chose a Monday (weekday) 5pm to better represent public transport commuters' demand.



In [ ]:
```python
import datetime
import pytz

def get_next_weekday(start_date, weekday, time, timezone_str):
    """
    Get the next specific weekday and time in the specified timezone.

    :param start_date: datetime.date, the date from which to find the next weekday
    :param weekday: int, desired weekday (0=Monday, 6=Sunday)
    :param time: datetime.time, desired time on the given weekday
    :param timezone_str: str, the string representing the timezone
    :return: datetime.datetime, the next weekday occurrence at the given time
    """
    timezone = pytz.timezone(timezone_str)
    current_datetime = datetime.datetime.combine(start_date, time)
    current_datetime = timezone.localize(current_datetime)

    # Increment date until we hit the desired weekday
    while current_datetime.weekday() != weekday:
        current_datetime += datetime.timedelta(days=1)

    return current_datetime

# Define parameters
desired_weekday = 0  # Monday
desired_time = datetime.time(17, 0)  # 5 PM
local_timezone = 'Asia/Singapore'  # Change as per your location

# Get today's date
today = datetime.date.today()
next_monday_at_5pm = get_next_weekday(today, desired_weekday, desired_time, local_timezone)

# Convert to Unix timestamp
unix_timestamp = int(next_monday_at_5pm.timestamp())
print("The Unix timestamp for the next Monday at 5 PM is:", unix_timestamp)
```

In [ ]:
```python
def get_travel_time(origin, destination, api_key, mode):
    endpoint_url = "https://maps.googleapis.com/maps/api/directions/json"

    # Parameters
    params = {
        'origin': origin,
        'destination': destination,
        'mode': mode,
        'key': api_key,
        'departure_time': unix_timestamp
    }

    # Make a GET request to the Google Maps API
    response = requests.get(endpoint_url, params=params)
    data = response.json()

    # Check if the request was successful
    if data['status'] == 'OK':
        # Extract travel time
        route = data['routes'][0]
        leg = route['legs'][0]
        duration = leg['duration']
        return round(duration['value']/60) # travel time in minutes
    else:
        return "Error: " + data['status']
```

In [ ]:
```python
# Get all 55 residential areas
planning_area_data = gpd.read_file('../data/planning_area.geojson')
planning_area_names = planning_area_data['planning_area']

# Define three target destinations of interest
destinations = {'ION': 'ION ORCHARD, SINGAPORE',
                'SGH': 'SINGAPORE GENERAL HOSPITAL, SINGAPORE',
                'CBD': 'CITY HALL MRT STATION, SINGAPORE'}
```

In [ ]:
```python
def save_travel_time():
    # dataframe to store travel times
    df1 = [] # public
    df2 = [] # private
    modes = ['transit', 'driving']

    for mode in modes:
        for origin in planning_area_names:
```

```
            for dest in destinations:
                ori = origin + ', SINGAPORE'
                travel_time = get_travel_time(ori, destinations[dest], GMAP_KEY, mode)
                if mode == 'transit':
                    df1.append([origin, dest, travel_time])
                else:
                    df2.append([origin, dest, travel_time])
    travel_data_public = pd.DataFrame(df1, columns=['planning_area', 'destination', 'time_public'])
    travel_data_private = pd.DataFrame(df2, columns=['planning_area', 'destination', 'time_private'])

    # save locally
    travel_data_public.to_csv("../data/travel_data_public.csv", index=False)
    travel_data_private.to_csv("../data/travel_data_private.csv", index=False)
```

Resident Population in 2023

We retrieved resident population data from SingStat since population density in an area can be a key factor for commuters in deciding whether to move to an area.

To make the data more interpretable for users in the dashboard, we cleaned the data using the below steps:

1. Categorise the population into 3 age groups: 0-20 years, 21-64 years and 65 years and above.
2. Summarise population based on age groups and region using `GROUPBY`.

The cleaned data can be found in: `../data/pop_by_agegroup.csv`

Construction of Connectivity Scores

Here we consider two aspects of connectivity of a residential area, availability (of public transport services) and accessibility (by taking public services). Below are formulaic represeations of the two scores.

Availability Score

$$\text{availability score} = w_1 \times \text{num busstops score} + w_2 \times \text{num trainstations score} + w_3 \times \text{num busservices score} + w_4 \times \text{avg bus freq score}$$

$$\text{where } w_i \text{ are weights to be assigned by the user in the interface and } \sum_{i=1}^{4} w_i = 1$$

Accessibility Score

$$\text{accessibility score} = w_1 \times \text{cbd diff score} + w_2 \times \text{ion diff score} + w_3 \times \text{sgh diff score}$$

$$\text{where } w_i \text{ are weights to be assigned by the user in the interface and } \sum_{i=1}^{3} w_i = 1$$

We use a weighted average of all the metrics in each category, where the coefficients can be customised by the commuters so long as they add up to 1. In this way, the final scores are dynamic and more representative of the user's personal preferences.

For example, a user who only cares about going to the CBD may assign a weight of 1 to `cbd diff score` and 0 for the rest when looking at accessibility.

Availability Score

The final scores can be found here: `../data/avail_with_score.csv`.

For all the metrics defined in our Availability Score, we summarised the data from LTA DataMall for each region using a `GROUPBY` clause.
Below is an example using Bus Stops data (`busstops`) for **Number of Bus Stops** in the region:

In [ ]:
```
num_busstops = busstops.groupby('planning_area').size().reset_index(name='num_busstops')
```

Number of Bus Routes in each Region

Using Bus Routes data (`busroutes`), we perform an additional join with the `busstops` data to map the planning areas to each bus service (route) and extract the unique records before summarising the data for each region.

In [ ]:
```
busroutes_planning_area = busroutes.merge(busstops, left_on='BusStopCode', right_on='BusStopCode', how='left')
busroutes_planningarea = busroutes_planning_area[['ServiceNo', 'planning_area']].drop_duplicates()
```

Average Frequency of Bus Services in each Region

Using Bus Services data, we perform another join with the `busroutes_planning_area` data to map the bus frequencies of each service to their respective planning areas.
Next, we further cleaned the data of the frequency columns before calculating the average of the frequencies.

Accessibility Score

The final scores can be found here: `../data/access_score.csv` .

Difference in Travel Time (min) between taking Public and Private Transport

The difference in travel time to each destination is defined by Public Transport Travel Time (`time_public`) - Private Transport Travel Time (`time_private`) from the Travel Time datasets.

Note that we do not simply take the public transport time as it could just result in downtown areas having shorter time and higher accessibility. Thus, the difference between public and private would be more indicative of how accessible a region is by only taking public transport, especially for commuters who are exploring heartland regions when looking to purchase/rent a new home.

Data Normalisation

To construct the final availability and accessibility scores, we performed Min-Max Scaling to scale all metrics into [0,1] to make them comparable. Noting that some metrics like **Average Bus Frequency** are negatively correlated with the connectivity score (e.g. a lower Average Bus Frequency means higher availability), we reversed the direction of these metrics before scaling to ensure that all metrics are positively correlated with the final score.

Below is an example using `availability` to construct **Availability Score**:

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler
min_max_scaler = MinMaxScaler()

# Apply Min-Max scaling to normalize values to positive numbers only
avail_values = availability.drop(columns='planning_area')

# Reverse direction of Freq columns
avail_values['avg_bus_freq'] = - avail_values['avg_bus_freq']

# Shift values by their minimum value to make them positive
avail_values = avail_values - avail_values.min()

# Apply Min-Max scaling to scale values to the range [0, 1]
avail_values = pd.DataFrame(min_max_scaler.fit_transform(avail_values), columns=avail_values.columns)

# Add the avail_values columns back to the DataFrame
avail_values = avail_values.add_suffix('_score')
avail_with_score = pd.concat([availability, avail_values], axis=1)
```