Preference-Based Approach for Optimizing Taxi Dispatching in Mobility Systems

Namrata De
namrata.de.1997@gmail.com

Masters Data Science, FU Berlin
Course: ”How can we model and solve future mobility challenges?”
Supervisor: Dr. Martin Aleksandrov
July, 2023

Abstract

Mobility systems face challenges in optimizing taxi dispatching to meet the preferences of drivers and passengers. This work proposes a novel preference-based approach that incorporates multiple preference layers, including maximum profit, preferred location, minimum waiting time, minimum fare, driver and passenger ratings. By leveraging a lexical graph and a matching algorithm between drivers and passengers (e.g., the Gale-Shapley algorithm), the system achieves a personalized and efficient taxi dispatching system. This ongoing work presents the methodology, benefits, and scenarios for implementing the preference-based approach, along with an example showcasing its practical application. Additionally, the document explores the algorithm for each case, planned work, and examines the adaptability of using subsets of preference layers for different scenarios.

Contents

1 Introduction

Mobility systems play a pivotal role in modern transportation, and taxi dispatching is a critical component of these systems. Conventional approaches often prioritize maximum profits and minimum travel distances, disregarding the unique preferences of drivers and passengers. To address this limitation, this work proposes a preference-based approach that considers multiple preference layers to optimize taxi dispatching.

Including passenger ratings and driver ratings in the algorithm contributes to the overall safety and security of the system. Passenger ratings help drivers make informed decisions about accepting ride requests, while driver ratings assist passengers in selecting reliable and reputable drivers.

2 Methodology

2.1 Problem Formulation

The preference-based algorithm is formalized as a one-to-one matching problem between drivers and passengers. We represent the set of available drivers as $\mathcal{D} = \{d_1, d_2, ..., d_n\}$ and the set of passenger ride requests as $\mathcal{P} = \{p_1, p_2, ..., p_m\}$. Each driver d_i and passenger p_j possess distinct preferences, denoted by $\mathcal{P}_{d_i}$ and $\mathcal{P}_{p_j}$, respectively. The primary objective is to find a matching $\mathcal{M} \subseteq \mathcal{D} \times \mathcal{P}$ that maximizes overall satisfaction by considering the expressed preferences of both drivers and passengers.

2.2 Preference Types and Scoring

The preference-based approach revolves around six preference layers: For Drivers, they are: Maximum profit (P), Preferred location (Z), Minimum waiting time for passengers (W), Minimum fare (F), Maximum driver rating (Rd), and Maximum passenger ratings (Rp).

Maximum profit (P): Profit can be calculated according to the taxi-dispatching-company's policies. Here, we only assume that Total Fare of a ride is the upper-bound of how much profit a driver makes from the ride. Profit is generally a key factor for drivers while choosing a passenger.

Preferred location (Z): We assume each driver has a set of preferred locations/ zones in the city/ neighborhood they operate in. While signing up for the service, they submit a list of their preferred zones to the taxi-dispatching-company. We have included this option because drivers might like to operate in zones which are more popular among passengers (allowing them to book more rides in a day), or they can choose to select rides where the destination is close to their base location, so the commute time and distance to base location is lesser.

Minimum waiting time (W): Passengers usually like to minimize the time waiting for a taxi to arrive. For example, for an emergency situation, lessening of waiting time is crucial.

Minimum fare (F): Some passengers might like to choose drivers who would be available for less fare, even if it means the waiting time would be higher. Cases where time is not a constraint, we believe many passengers would like to opt for this choice.

Driver Rating/ Passenger Rating (Rd/ Rp): To generate ratings, after completion of each ride, both driver and passenger have to rate each other anonymously, based on their behaviour during the ride, and the rating of a user (driver/passenger) is the average rating received by them so far, while every user initially has the highest possible rating assigned to them by default. Unlike existing algorithms, this approach adds an additional layer of safety by including ratings. Since extracting personal data like gender, race etc. to filter driver/ passengers could be discouraged by the respective users and create bias in the algorithm, using a technique like rating keeps personal information anonymous, while also providing an overall sense of how safe/ reliable a user is. The algorithm aims to penalize users who have lower ratings compared to other users, we believe, which would motivate users to be cooperative and maintain good behaviour during the ride, increasing the overall safety/ security measures.

While signing up for the taxi dispatching service, drivers are requested to set up their order of preferences, in terms of the following (in no particular order):

- Preference of rides with high reward/profit (Preference type: P)
- Preference of rides with destinations in preferred zones of the city (z1/z2/z3/.../zn) (Preference type: Z)
- Preference of rides with passengers with higher ratings (Preference type: Rp)
- All has equal weightage (Preference type: Ed)

Similarly, passengers are also requested to set up the same (in no particular order):

- Preference of rides with minimum waiting time (Preference type: W)
- Preference of rides with minimum fare (Preference type: F)
- Preference of rides with drivers with higher ratings (Preference type: Rd)
- All has equal weightage (Preference type: Ep)

Each driver and passenger profile comprises key parameters, such as ratings, preferred zones, current location, destination location, preference orders etc. Leveraging the preference orders, a lexical graph is constructed for each user. We then aim to utilize a matching algorithm, (in this case, we have used the Gale-Shapley algorithm) to match drivers with passengers based on their preference layers. How we create a modified algorithm to create matching are further explained in Section 4 ("Proposed Work").

3 Example (with Random Data)

Assume we have a system with 4 available drivers and 4 potential passengers. Before starting the matching algorithm, we assume the Taxi Company has the following information about the Drivers and the Passengers at a given time.

3.1 Driver Profiles

Driver ID	Rating	Preferred Zone	Current Location	Preference Orders
Driver 1	5	Zone A, Zone B	52°06 N 5°07 E	P $\rightarrow$ Z $\rightarrow$ Rp
Driver 2	4.2	Zone B, Zone C	37°49 N 25°45 W	Z $\rightarrow$ P $\rightarrow$ Rp
Driver 3	1	Zone A, Zone B	36°11 N 44°01 E	E
Driver 4	3	Zone A, Zone C	49°54 N 97°08 W	Rp $\rightarrow$ P $\rightarrow$ Z

Table 1: Driver Profiles

3.2 Passenger Profiles

Passenger ID	Rating	Pickup	Destination	Distance	Destination Zone	Base Fare	Preference Orders
Passenger 1	2	35°25 N 136°46 E	60°10 N 24°56 E	10 km	Zone A	10	W → F → Rd
Passenger 2	4	42°28 N 59°36 E	42°00 N 21°26 E	5 km	Zone C	23	Rd → W → F
Passenger 3	5	18°29 S 70°20 W	34°26 S 150°53 E	12 km	Zone A	11.8	Rd → F → W
Passenger 4	3.5	13°31 N 144°50 E	19°55 N 99°50 E	30 km	Zone B	15	E

Table 2: Passenger Profiles

Looking at Drivers' Perspectives,

Driver 1 vs. Passengers

Driver 1 Current Location: 52°06 N 5°07 E (from Driver Profile)

Passenger	Pickup Location	Distance to Pickup	Time to Reach Pickup	Profit	Passenger Rating	Destination Zone Matching
Passenger 1	35°25 N 136°46 E	1.5 km	15 min	$150	2	Yes
Passenger 2	42°28 N 59°36 E	2 km	20 min	$25	4	No
Passenger 3	18°29 S 70°20 W	5 km	30 min	$180	5	Yes
Passenger 4	13°31 N 144°50 E	0.5 km	5 min	$300	3.5	Yes

Table 3: Driver 1 vs. Passengers

Referring to "Preference Orders" from "Driver Profiles," we know Driver 1 prefers P → Z → Rp, i.e. the most important criteria for them is securing maximum profit, then we give preference to specific zones (Driver 1 prefers Zone A and Zone B as destination) and then we care about finding passengers with good ratings.

According to Profit, his preference is: Passenger 4 → Passenger 3 → Passenger 1 → Passenger 2

According to Destination Zones, his preference is: Passenger 1 or Passenger 2 or Passenger 3 (all in Zone A/B) → Passenger 2 (Zone C)

According to Ratings, his preference is: Passenger 3 → Passenger 2 → Passenger 4 → Passenger 1

We can form a lexical graph for Driver 1 as following:

Driver 1's Lexicographic Preference List: (denoted by Di_Lm, where "i" denotes the Driver ID and m denotes the layer number, {m = 1, 2, 3})

- Layer 1 (Profit): Passenger 4 → Passenger 3 → Passenger 1 → Passenger 2 (D1_L1)
- Layer 2 (Location): Passenger 1 or Passenger 3 or Passenger 4 → Passenger 2 (D1_L2)
- Layer 3 (Passenger Ratings): Passenger 3 → Passenger 2 → Passenger 4 → Passenger 1 (D1_L3)

Similarly, we can create 3-fold preference lists for other drivers.

Passenger 1 vs. Drivers

Pickup Location for Passenger 1: 35°25 N 136°46 E (from Passenger Profile)

Driver	Driver Rating	Distance to Pickup Location	Waiting Time For Passenger	Fare
Driver 1	5	1.5 km	15 min	$150
Driver 2	4.2	10 km	40 min	$2
Driver 3	1	5 km	30 min	$155
Driver 4	3	0.4 km	2 min	$200

Table 4: Passenger 1 vs. Drivers

Referring to "Preference Orders" from "Passenger Profiles," we know Passenger 1 prefers W → F → Rd, i.e. the most important criteria for them is having minimum waiting time, then we give preference to choosing minimum fare taxis and then we care about finding drivers with good ratings.

According to Waiting Time, their preference is: Driver 4 → Driver 1 → Driver 3 → Driver 2

According to Taxi Fares, their preference is: Driver 2 → Driver 1 → Driver 3 → Driver 4

According to Ratings, their preference is: Driver 1 → Driver 2 → Driver 4 → Driver 3

We can form a lexical graph for Passenger 1 as following:

Passenger 1's Lexicographic Preference List: (denoted by Pi_Ln, where "i" denotes the Passenger ID and n denotes the layer number, {n = 1, 2, 3}

- Layer 1 (Waiting Time): Driver 4 → Driver 1 → Driver 3 → Driver 2 (P1_L1)
- Layer 2 (Cost): Driver 2 → Driver 1 → Driver 3 → Driver 4 (P1_L2)
- Layer 3 (Driver Ratings): Driver 1 → Driver 2 → Driver 4 → Driver 3 (P1_L3)

Similarly, we can create 3-fold preference lists for other passengers.

Now, we can refer to the following sample multi-layered preference list for our modified matching algorithm.

Driver Preferences

Driver 1

- Layer 1 (Profit): P4 → P3 → P1 → P2 ... (D1_L1)
- Layer 2 (Zone Matching): P1 or P3 or P4 → P2 ... (D1_L2)
- Layer 3 (Passengers' Ratings): P3 → P2 → P4 → P1 ... (D1_L3)

Driver 2

- Layer 1 (Zone Matching): P4 → P1 or P3 → P2 ... (D2_L1)
- Layer 2 (Profit): P3 → P4 → P1 → P2 ... (D2_L2)
- Layer 3 (Passengers' Ratings): P3 → P2 → P4 → P1 ... (D2_L3)

Driver 3

- All Equal Preference (Pick one Layer Randomly):
  - Zone Matching: P1 or P3 → P4 → P2 ... (D3_L1)

- Profit: P3 → P2 → P1 → P4 . . . (D3_L2)
- Passengers' Ratings: P3 → P2 → P4 → P1 . . . (D3_L3)

Driver 4

- Layer 1 (Passengers' Ratings): P3 or P2 → P4 → P1 . . . (D4_L1) (choose if yet unmatched) (otherwise)
- Layer 2 (Profit): P1 → P4 → P3 → P2 . . . (D4_L2)
- Layer 3 (Zone Matching): P1 or P3 → P2 → P4 . . . (D4_L3)

Passenger Preferences

Passenger 1

- Layer 1 (Waiting Time): D4 → D1 → D3 → D2 . . . (P1_L1)
- Layer 2 (Cost): D2 → D1 → D3 → D4 . . . (P1_L2)
- Layer 3 (Drivers' Ratings): D1 → D2 → D4 → D3 . . . (P1_L3)

Passenger 2

- Layer 1 (Drivers' Ratings): D1 → D2 → D4 → D3 . . . (P2_L1)
- Layer 2 (Waiting Time): D4 → D2 → D3 → D1 . . . (P2_L2)
- Layer 3 (Cost): D1 → D2 → D3 → D4 . . . (P2_L3)

Passenger 3

- Layer 1 (Drivers' Ratings): D1 → D2 → D4 → D3 . . . (P3_L1)
- Layer 2 (Cost): D2 → D1 → D4 → D3 . . . (P3_L2)
- Layer 3 (Waiting Time): D1 → D2 → D3 → D4 . . . (P3_L3)

Passenger 4

- All Equal Preference (Pick One Layer Randomly):
  - Drivers' Ratings: D1 → D2 → D4 → D3 . . . (P4_L1)
  - Cost: D3 → D2 → D4 → D1 . . . (P4_L2)
  - Waiting Time: D3 → D4 → D1 → D2 . . . (P4_L3)

4 Proposed Work

4.1 Case 1: Preference Over Layers are Considered (Layer 1 being most preferred and Layer 3 being least preferred)

In this case, we consider the preference over the layers, where Layer 1 is the most preferred, followed by Layer 2, and then Layer 3. We will run the matching algorithm on the following combinations:

- First Iteration of Choices:
  - D1_L1: P4 → P3 → P1 → P2
  - P1_L1: D4 → D1 → D3 → D2
  - D2_L1: P4 → P1 or P3 → P2

Since there is a tie between P1 and D3 in D2_L1, we break the tie using the next most preferred layer (ref. D2_L2) - where P3 is preferred over P1. So, the final order is:

  - D2_L1: P4 → P3 → P1 → P2
  - P2_L1: D1 → D2 → D4 → D3
  - D3_L1: P3 → P2 → P1 → P4
  - P3_L1: D1 → D2 → D4 → D3
  - D4_L1: P3 → P2 → P4 → P1
  - P4_L1: D3 → D4 → D1 → D2

Using a matching algorithm (in this case, the Gale-Shapley Algorithm), we find the following matching pairs:

- D1 - P4
- D2 - P3
- D3 - P1
- D4 - P2

4.2 Case 2: Scoring Over All Layers Using Borda Voting Rule

A voting system (we have used the Borda Scoring System in this case) is implemented over each layer and each preference order and a modified order of preferences is created regarding each driver and passenger.

We then apply the matching algorithm (for example, the Gale-Shapley Algorithm) on the generated set.

4.3 Case 3: Scoring Over All Layers Using Weighted Borda Voting Rule

In this case, while scoring each candidate, we also take in account the original layer preferences of each user. Therefore, we assign weights to each layer of preference, Layer 1 having the highest weight followed by Layer 2 and Layer 3 respectively. The weight for each layer can be chosen by the taxi-dispatching-company. A voting system (we have used the Weighted Borda Scoring System in this case) is then implemented over each layer and each preference order and a modified order of preferences is created regarding each driver and passenger.

We then apply the matching algorithm on the generated set.

4.4 Case 4: Discarding Preference Over Layers

In this case, we discard the preference over the multiple layers and run matching algorithm on combinations selected by the taxi-dispatching-company (generated from the following possible combinations of Drivers' vs. Passengers' Preferences):

- Max Profit vs. Min Waiting Time
- Max Profit vs. Min Cost
- Max Profit vs. Drivers' Rating
- Preferred Location vs. Min Waiting Time
- Preferred Location vs. Min Cost
- Preferred Location vs. Drivers' Rating
- Passengers' Rating vs. Min Waiting Time
- Passengers' Rating vs. Min Cost
- Passengers' Rating vs. Drivers' Rating

This method allows the algorithm to be robust and run on a subset of Case 1. For future work, we aim to make it even more dynamic, where the taxi-dispatching-company has full control over how many layers to choose for each driver and passenger. For example,

- {Max Profit → Passengers' Rating} vs. Min Waiting Time
- Max Profit vs. {Min Waiting Time → Min Fare}

- {Preferred Location → Max Profit} vs. {Drivers' Rating → Min Waiting Time → Min Fare}

are also some of the possible choice of options on how we can run the algorithm.

4.5 Measures for Evaluation

We will evaluate the above algorithms based on the following criteria:

- Number of blocking pairs from preference layers which are not considered (for Case 1 and Case 4).
- Safety score (a function of rating of drivers and passengers, the algorithm will always try to maximize the safety score).
- Total profit of drivers
- Total waiting time for passengers
- Total fare

4.6 Ongoing/Planned Work

Currently, we have created matching algorithms for Case 1, Case 2, Case 3 and parts of Case 4 (one vs one matching). (Please refer to Section 7.2 (Appendix)). We have also created algorithms to generate sample data for Drivers' and Passengers' profiles, where number of driver/ passenger profiles to be created are provided as input parameters. (Please refer to Section 7.1 (Appendix)).

The matching algorithms mentioned above has been applied to the set of synthetic data generated (Please refer to Section 7.3 (Appendix) to see the results returned by our algorithm).

As further work, we plan to test the performance of this algorithms based on the criteria mentioned in Section 4.5. We also aim to check the time complexity of our algorithms when applied on a large dataset of Driver and Passenger profiles. Furthermore, we will also check if we can find an existing real-world dataset containing similar information as we have on our Drivers'/ Passengers' profiles and run our algorithms on it to check how compatible they are for being used with real-world data.

5 Conclusion

This work introduces a preference-based approach for optimizing taxi dispatching in mobility systems. The proposed algorithm accommodates multiple preference layers, ensuring that drivers and passengers are matched based on their expressed preferences. The ongoing work aims to implement and evaluate the algorithm on synthetic and real-world dataset to demonstrate its effectiveness in delivering personalized and efficient taxi dispatching services.

6 References

1. Gale, D., & Shapley, L. S. (1962). College admissions and the stability of marriage. American Mathematical Monthly, 69(1), 9-15.

2. Roth, A. E. (1982). The Economics of Matching: Stability and Incentives. Mathematics of Operations Research, 7(4), 617-628.

3. Gusfield, D., & Irving, R. W. (1989). The Stable Marriage Problem: Structure and Algorithms. MIT Press.

4. Knuth, D. E. (1976). Mariages stables. Les Presses de l'Université de Montréal.

5. Manlove, D. F. (2013). Algorithmics of Matching Under Preferences. World Scientific.

6. Abdulkadiroglu, A., & Sönmez, T. (2003). School Choice: A Mechanism Design Approach. American Economic Review, 93(3), 729-747.

7. Kesten, O. (2010). On two kinds of manipulability of medical resident matching algorithms. Games and Economic Behavior, 69(2), 419-435.

8. Kojima, F., Pathak, P. A., & Roth, A. E. (2013). Matching with Couples: Stability and Incentives in Large Markets. The Quarterly Journal of Economics, 128(4), 1585-1632.

9. Blum, A., & Roth, A. E. (2015). A Simple Algorithm for Finding a Stable Marriage. Algorithmica, 72(2), 397-414.

10. McDermid, M. (2017). The Gale-Shapley algorithm and the national resident matching program: A practical guide. Bulletin of the American College of Surgeons, 102(9), 26-30.

7 Appendix

Appendix

July 31, 2023

[ ]:
```
# 7.1 Create Driver and Passenger Profiles.
### Give the number of Driver and Passengers you want to create as input.
```

[1]:
```
def create_driver_passenger_profiles(num_drivers, num_passengers):

    """
    Generate synthetic driver and passenger profiles for ride-sharing matching.

    Parameters:
        num_drivers (int): Number of drivers to generate profiles for.
        num_passengers (int): Number of passengers to generate profiles for.

    Returns:
        pandas.DataFrame, pandas.DataFrame: Two DataFrames representing driver␣
 ↪profiles and passenger profiles, respectively.
            The driver_profiles_df contains columns: 'Driver_ID',␣
 ↪'Driver_Rating', 'Preferred_Zones', 'Current_Location',␣
 ↪'Preference_Orders_for_Drivers'.
            The passenger_profiles_df contains columns: 'Passenger_ID',␣
 ↪'Passenger_Rating', 'Pickup_Location', 'Destination_Location',␣
 ↪'Distance_km', 'Destination_District', 'Base_Fare',␣
 ↪'Preference_Orders_for_Passengers'.
    """



    import warnings
    warnings.filterwarnings('ignore')
    import pandas as pd
    import numpy as np
    import random
    from geopy.geocoders import Nominatim
    from geopy.distance import geodesic
    # Set the random seed for reproducibility
    random.seed(42)

    # Define the districts of Berlin
```

```
    districts = [
        "Charlottenburg-Wilmersdorf", "Friedrichshain-Kreuzberg",␣
↪"Lichtenberg", "Marzahn-Hellersdorf",
        "Mitte", "Neukölln", "Pankow", "Reinickendorf", "Spandau",␣
↪"Steglitz-Zehlendorf",
        "Tempelhof-Schöneberg", "Treptow-Köpenick"
    ]

    # Define the frequencies of districts (can be adjusted as needed)
    district_frequencies = [
        0.15,   # Charlottenburg-Wilmersdorf
        0.2,    # Friedrichshain-Kreuzberg
        0.1,    # Lichtenberg
        0.1,    # Marzahn-Hellersdorf
        0.1,    # Mitte
        0.05,   # Neukölln
        0.05,   # Pankow
        0.05,   # Reinickendorf
        0.05,   # Spandau
        0.05,   # Steglitz-Zehlendorf
        0.05,   # Tempelhof-Schöneberg
        0.05    # Treptow-Köpenick
    ]

    # Define the preference orders for the driver
    preference_orders_for_drivers = [
        "P → Z → Rp",
        "P → Rp → Z",
        "Rp → P → Z",
        "Rp → Z → P",
        "Z → Rp → P",
        "Z → P → Rp"
    ]

    # Create a geolocator object
    geolocator = Nominatim(user_agent="driver_profiles")
    # Define the preference orders for the Passenger
    preference_orders_for_passengers = [
        "W → F → Rd",
        "W → Rd → F",
        "F → Rd → W",
        "F → W → Rd",
        "Rd → W → F",
        "Rd → F → W"
    ]
    def generate_random_coordinates_d():
        # Generate random latitude and longitude within the bounds of Berlin
```

```
        latitude = np.random.uniform(52.338234, 52.67551)
        longitude = np.random.uniform(13.088304, 13.761117)
        location = geolocator.reverse(f"{latitude:.6f}, {longitude:.6f}",␣
↪exactly_one=True)
        return f"{latitude:.6f}, {longitude:.6f} ({location.address})"

    def generate_random_coordinates_p():
        # Generate random latitude and longitude within the bounds of Berlin
        latitude = np.random.uniform(52.338234, 52.67551)
        longitude = np.random.uniform(13.088304, 13.761117)
        location = geolocator.reverse(f"{latitude:.6f}, {longitude:.6f}",␣
↪exactly_one=True)
        return latitude, longitude, location.address


    def get_destination_district(latitude, longitude):
        # Perform reverse geocoding to get address details
        location = geolocator.reverse((latitude, longitude), exactly_one=True)

        # Extract the district from the address details
        district = location.raw.get("address", {}).get("suburb", None)
        return district


    def generate_driver_profiles(num_drivers):
        # Generate unique Driver_IDs
        driver_ids = random.sample(range(1, 501), num_drivers)

        # Generate Rating using Gaussian distribution (mean=3.5, std=1)
        ratings = np.around(np.random.normal(3.5, 1, num_drivers), decimals=1)

        # Generate three Preferred Zones based on district frequencies for each␣
↪driver
        preferred_zones = [random.sample(districts, 3) for _ in␣
↪range(num_drivers)]

        # Generate Current Location (latitude and longitude) and corresponding␣
↪district based on Berlin's geographic bounds
        current_locations = [generate_random_coordinates_d() for _ in␣
↪range(num_drivers)]

        # Generate Preference Orders
        preference_orders_list_for_drivers = random.
↪choices(preference_orders_for_drivers, k=num_drivers)

        # Create the DataFrame
```

```
        driver_profiles_df = pd.DataFrame({
            "Driver_ID": ["driver" + str(id) for id in driver_ids],
            "Driver_Rating": ratings,
            "Preferred_Zones": [", ".join(zones) for zones in preferred_zones],
            "Current_Location": current_locations,
            "Preference_Orders_for_Drivers": preference_orders_list_for_drivers
        })

        return driver_profiles_df


    def generate_passenger_profiles(num_passengers):
        # Generate unique Passenger_IDs
        passenger_ids = random.sample(range(1, 501), num_passengers)

        # Generate Rating using Gaussian distribution (mean=3.5, std=1)
        ratings = np.around(np.random.normal(3.5, 1, num_passengers), decimals=1)

        # Generate Pickup Location (latitude, longitude, and address) based on Berlin's geographic bounds
        pickup_locations = [generate_random_coordinates_p() for _ in range(num_passengers)]

        # Generate Destination Location (latitude, longitude, and address) based on Berlin's geographic bounds
        destination_locations = [generate_random_coordinates_p() for _ in range(num_passengers)]

        # Calculate the Distance between Pickup Location and Destination Location (in km)
        distances = [geodesic(pickup[:2], dest[:2]).kilometers for pickup, dest in zip(pickup_locations, destination_locations)]
        distances_rounded = [round(distance, 2) for distance in distances]

        # Get the Destination District for each destination location
        destination_districts = [get_destination_district(dest[0], dest[1]) for dest in destination_locations]

        # Calculate the Fare as Distance multiplied by a factor (e.g., 1.5)
        fare_factor = 1.5
        fares = [distance * fare_factor for distance in distances]
        fares_rounded = [round(fare, 2) for fare in fares]

        # Generate Preference Orders
```

```
        preference_orders_list_for_passengers = random.
↪choices(preference_orders_for_passengers, k=num_passengers)

        # Create the DataFrame
        passenger_profiles_df = pd.DataFrame({
            "Passenger_ID": ["passenger" + str(id) for id in passenger_ids],
            "Passenger_Rating": ratings,
            "Pickup_Location": [f"{lat:.6f}, {lon:.6f} ({address})" for lat,␣
↪lon, address in pickup_locations],
            "Destination_Location": [f"{lat:.6f}, {lon:.6f} ({address})" for␣
↪lat, lon, address in destination_locations],
            "Distance_km": distances_rounded,
            "Destination_District": destination_districts,
            "Base_Fare": fares_rounded,
            "Preference_Orders_for_Passengers":␣
↪preference_orders_list_for_passengers
        })

        return passenger_profiles_df


    num_drivers_to_create = num_drivers
    driver_profiles_df = generate_driver_profiles(num_drivers_to_create)
    num_passengers_to_create = num_passengers
    passenger_profiles_df =␣
↪generate_passenger_profiles(num_passengers_to_create)
    return driver_profiles_df, passenger_profiles_df
```

[ ]:
```
# 7.2.1 Create matching for Case 1 (where we consider Layer Preferences for␣
↪each User)
### Give the Driver and Passengers Profiles as input.
```

[2]:
```
def create_matching_case_1 (driver_profiles_df, passenger_profiles_df):

    """
    Perform matching between drivers and passengers based on their profiles and␣
↪preferences, where layer preferences are considered for each driver/␣
↪passenger.

    Parameters:
        driver_profiles_df (pandas.DataFrame): DataFrame containing driver␣
↪profiles with columns 'Driver_ID', 'Driver_Rating', 'Preferred_Zones',␣
↪'Current_Location', and 'Preference_Orders_for_Drivers'.
```

```
        passenger_profiles_df (pandas.DataFrame): DataFrame containing␣
↪passenger profiles with columns 'Passenger_ID', 'Passenger_Rating',␣
↪'Pickup_Location', 'Destination_Location', 'Distance_km',␣
↪'Destination_District', 'Base_Fare', and 'Preference_Orders_for_Passengers'.

    Returns:
        tuple: A tuple containing two dictionaries representing the matching␣
↪results.
            - The first dictionary (drivers_matching) contains the matched␣
↪passenger for each driver.
            - The second dictionary (passengers_matching) contains the matched␣
↪driver for each passenger.
    """

    import warnings
    warnings.filterwarnings('ignore')
    import pandas as pd
    import numpy as np
    import random
    from geopy.geocoders import Nominatim
    from geopy.distance import geodesic
    # Set the random seed for reproducibility
#     random.seed(42)
    traffic_factors = [0.1, 0.5, 1.5]



    def get_distance_time_profit(driver_row, passenger_row):
        # Extract latitude and longitude of driver's current location
        driver_latitude, driver_longitude = [float(coord.strip()) for coord in␣
↪driver_row["Current_Location"].split("(")[0].split(",")]

        # Extract latitude and longitude of passenger's pickup location
        pickup_latitude, pickup_longitude = [float(coord.strip()) for coord in␣
↪passenger_row["Pickup_Location"].split("(")[0].split(",")]

        # Calculate distance between driver's current location and passenger's␣
↪pickup location
        distance_to_pickup = geodesic((driver_latitude, driver_longitude),␣
↪(pickup_latitude, pickup_longitude)).kilometers

        # Choose a random traffic factor from the list
        traffic_factor = random.choice(traffic_factors)

        # Calculate time to reach pickup location (distance * traffic factor)
        time_to_reach_pickup = distance_to_pickup * traffic_factor
```

```python
    # Calculate profit (m * Fare)
    profit_base = 0.8 * passenger_row["Base_Fare"]
    profit_travel_to_passenger = 0.2 * distance_to_pickup
    profit_net = profit_base + profit_travel_to_passenger

    fare_base = passenger_row["Base_Fare"]
    fare_travel= 0.6 * distance_to_pickup
    fare_net = fare_base + fare_travel

    return distance_to_pickup, time_to_reach_pickup, profit_net, fare_net

def evaluate_passengers(driver_profiles_df, passenger_profiles_df):
    # Initialize an empty list to store evaluation results
    evaluation_results = []

    # Iterate over each row (driver) in driver_profiles
    for _, driver_row in driver_profiles_df.iterrows():
        driver_id = driver_row["Driver_ID"]

        # Iterate over each row (passenger) in passenger_profiles
        for _, passenger_row in passenger_profiles_df.iterrows():
            passenger_id = passenger_row["Passenger_ID"]

            # Get the distance, time, and profit for the current driver-passenger pair
            distance, time_to_reach, profit_net, fare_net = get_distance_time_profit(driver_row, passenger_row)
            rating = passenger_row["Passenger_Rating"]
            destination_zone = passenger_row["Destination_District"]
            zone_matching = np.random.choice(["1", "0"])

            # Append the evaluation result to the list
            evaluation_results.append([driver_id, passenger_id, distance, time_to_reach, profit_net, fare_net, rating, destination_zone,zone_matching])

    # Create the evaluate_passenger DataFrame
    evaluate_passenger_df = pd.DataFrame(evaluation_results, columns=["Driver_ID", "Passenger_ID", "Distance to Pickup", "Time to Reach Pickup", "Net Profit", "Net Fare", "Passenger_Rating", "Destination Zone","Zone Matching"])
    evaluate_passenger_df = evaluate_passenger_df.sort_values(by=["Driver_ID","Passenger_ID"])
    evaluate_passenger_df = evaluate_passenger_df.reset_index(drop=True)


    return evaluate_passenger_df
```

```
def evaluate_drivers(driver_profiles_df, evaluate_passenger_df):

    # Merge the two dataframes based on the common column "Driver_ID"
    evaluate_drivers_df = pd.merge(evaluate_passenger_df,␣
↪driver_profiles_df, on="Driver_ID")

    # Select only the desired columns
    evaluate_drivers_df = evaluate_drivers_df[["Passenger_ID", "Driver_ID",␣
↪"Driver_Rating", "Net Fare", "Time to Reach Pickup"]]

    # Sort the dataframe by Passenger_ID and then Driver_ID
    evaluate_drivers_df = evaluate_drivers_df.
↪sort_values(by=["Passenger_ID", "Driver_ID"])

    # Reset the index if needed
    evaluate_drivers_df = evaluate_drivers_df.reset_index(drop=True)
    return evaluate_drivers_df


def create_preference_list_for_drivers(driver_profiles_df,␣
↪passenger_profiles_df, evaluate_passenger_df):
    # Initialize an empty dictionary to store preference lists for each␣
↪driver


    # Iterate over each row (driver) in driver_profiles
    for _, driver_row in driver_profiles_df.iterrows():
        driver_id = driver_row["Driver_ID"]
        preference_order = driver_row["Preference_Orders_for_Drivers"].
↪split(" → ")
#         print(driver_id)
#         print(preference_order)

#         Filter passengers who were evaluated by the current driver
        driver_evaluations =␣
↪evaluate_passenger_df[evaluate_passenger_df["Driver_ID"] == driver_id]
#         print(driver_evaluations)
#         print("----------")

        if   preference_order[0] == "Rp" and preference_order[1] == "P" and␣
↪preference_order[2] == "Z"   :
            driver_evaluations.sort_values(by=["Passenger_Rating","Net␣
↪Profit","Zone Matching"], ascending=[False, False,False], inplace=True)
        elif preference_order[0] == "Rp" and preference_order[1] == "Z" and␣
↪preference_order[2] == "P"   :
```

```
                driver_evaluations.sort_values(by=["Passenger_Rating","Zone␣
↪Matching","Net Profit"], ascending=[False, False,False], inplace=True)
            elif preference_order[0] == "P" and preference_order[1] == "Z" and␣
↪preference_order[2] == "Rp"  :
                driver_evaluations.sort_values(by=["Net Profit","Zone␣
↪Matching","Passenger_Rating"], ascending=[False, False,False], inplace=True)
            elif preference_order[0] == "P" and preference_order[1] == "Rp" and␣
↪preference_order[2] == "Z"  :
                driver_evaluations.sort_values(by=["Net␣
↪Profit","Passenger_Rating","Zone Matching"], ascending=[False, False,False],␣
↪inplace=True)
            elif preference_order[0] == "Z" and preference_order[1] == "Rp" and␣
↪preference_order[2] == "P"  :
                driver_evaluations.sort_values(by=["Zone␣
↪Matching","Passenger_Rating","Net Profit"], ascending=[False, False,False],␣
↪inplace=True)
            elif preference_order[0] == "Z" and preference_order[1] == "P" and␣
↪preference_order[2] == "Rp"  :
                driver_evaluations.sort_values(by=["Zone Matching""Net␣
↪Profit","Passenger_Rating"], ascending=[False, False,False], inplace=True)


    #         print(driver_evaluations)
    #         print("----------")
    #         print("@@@@@@@@@@@@@")


            # Add the sorted passenger IDs to the preference list
            preference_lists_for_drivers[driver_id] =␣
↪driver_evaluations["Passenger_ID"].tolist()

        return preference_lists_for_drivers

    def create_preference_list_for_passengers(passenger_profiles_df,␣
↪evaluate_drivers_df):

        # Iterate over each row (passenger) in passenger_profiles
        for _, passenger_row in passenger_profiles_df.iterrows():
            passenger_id = passenger_row["Passenger_ID"]
            preference_order =␣
↪passenger_row["Preference_Orders_for_Passengers"].split(" → ")

            # Filter drivers who were evaluated by the current passenger
            passenger_evaluations =␣
↪evaluate_drivers_df[evaluate_drivers_df["Passenger_ID"] == passenger_id]
```

21

```python
        # Sort drivers based on passenger's preference order
        if preference_order[0] == "Rd" and preference_order[1] == "W" and preference_order[2] == "F":
            passenger_evaluations.sort_values(by=["Driver_Rating", "Time to Reach Pickup", "Net Fare"], ascending=[False, True, True], inplace=True)
        elif preference_order[0] == "W" and preference_order[1] == "F" and preference_order[2] == "Rd":
            passenger_evaluations.sort_values(by=["Time to Reach Pickup", "Net Fare", "Driver_Rating"], ascending=[True, True, False], inplace=True)
        elif preference_order[0] == "W" and preference_order[1] == "Rd" and preference_order[2] == "F":
            passenger_evaluations.sort_values(by=["Time to Reach Pickup", "Driver_Rating", "Net Fare"], ascending=[True, False, True], inplace=True)
        elif preference_order[0] == "F" and preference_order[1] == "Rd" and preference_order[2] == "W":
            passenger_evaluations.sort_values(by=["Net Fare", "Driver_Rating", "Time to Reach Pickup"], ascending=[True, False, True], inplace=True)
        elif preference_order[0] == "F" and preference_order[1] == "W" and preference_order[2] == "Rd":
            passenger_evaluations.sort_values(by=["Net Fare", "Time to Reach Pickup", "Driver_Rating"], ascending=[True, True, False], inplace=True)
        elif preference_order[0] == "Rd" and preference_order[1] == "F" and preference_order[2] == "W":
            passenger_evaluations.sort_values(by=["Driver_Rating", "Net Fare", "Time to Reach Pickup"], ascending=[False, True, True], inplace=True)

        # Add the sorted driver IDs to the preference list
        preference_lists_for_passengers[passenger_id] = passenger_evaluations["Driver_ID"].tolist()

    return preference_lists_for_passengers

def match_driver_passenger(drivers_preferences, passengers_preferences):
    # Number of drivers/passengers
    n = len(drivers_preferences)

    # Initialize an empty matching for drivers and passengers
    drivers_matching = {}
    passengers_matching = {}

    # Initialize each driver to be free
    drivers_free = {driver: True for driver in drivers_preferences.keys()}

    while True:
        # Find an unmatched driver
```

```
            driver = next((driver for driver, is_free in drivers_free.items()␣
↪if is_free), None)

            if driver is None:
                # All drivers are matched
                break

            # Iterate over the preferences of the driver
            for passenger in drivers_preferences[driver]:
                # Check if the passenger is free
                if passenger not in passengers_matching:
                    # The passenger is free, so match the driver and passenger
                    drivers_matching[driver] = passenger
                    passengers_matching[passenger] = driver
                    drivers_free[driver] = False
                    break
                else:
                    # The passenger is currently matched
                    matched_driver = passengers_matching[passenger]

                    # Check if the passenger prefers the new driver over the␣
↪current match
                    if passengers_preferences[passenger].index(driver) <␣
↪passengers_preferences[passenger].index(matched_driver):
                        # The passenger prefers the new driver, so update the␣
↪matching
                        drivers_matching[driver] = passenger
                        drivers_matching[matched_driver] = None
                        passengers_matching[passenger] = driver
                        drivers_free[driver] = False
                        drivers_free[matched_driver] = True
                        break

        return drivers_matching, passengers_matching


    evaluate_passenger_df= evaluate_passengers(driver_profiles_df,␣
↪passenger_profiles_df)
    evaluate_drivers_df= evaluate_drivers(driver_profiles_df,␣
↪evaluate_passenger_df)

    # Initialize an empty dictionary to store preference lists for each driver
    preference_lists_for_drivers = {}
    preference_lists_for_drivers =␣
↪create_preference_list_for_drivers(driver_profiles_df,␣
↪passenger_profiles_df, evaluate_passenger_df)
```

```
    # Initialize an empty dictionary to store preference lists for each␣
↪passenger
    preference_lists_for_passengers = {}
    preference_lists_for_passengers =␣
↪create_preference_list_for_passengers(passenger_profiles_df,␣
↪evaluate_drivers_df)

    drivers_matching, passengers_matching =␣
↪match_driver_passenger(preference_lists_for_drivers,␣
↪preference_lists_for_passengers)
    return drivers_matching
```

[ ]:
```
# 7.2.2 Create matching for Case 2 (where we score Users with Borda Voting Rule␣
↪)
## [Each Layer of Preference has Equal priority while calculating Borda Scores]
### Give the Driver and Passengers Profiles as input.
```

[3]:
```
def create_matching_case_2 (driver_profiles_df, passenger_profiles_df):

    """
    Perform matching between drivers and passengers based on their profiles and␣
↪preferences using the Borda voting rule.

    Parameters:
        driver_profiles_df (pandas.DataFrame): DataFrame containing driver␣
↪profiles with columns 'Driver_ID', 'Driver_Rating', 'Preferred_Zones',␣
↪'Current_Location', and 'Preference_Orders_for_Drivers'.
        passenger_profiles_df (pandas.DataFrame): DataFrame containing␣
↪passenger profiles with columns 'Passenger_ID', 'Passenger_Rating',␣
↪'Pickup_Location', 'Destination_Location', 'Distance_km',␣
↪'Destination_District', 'Base_Fare', and 'Preference_Orders_for_Passengers'.

    Returns:
        tuple: A tuple containing two dictionaries representing the matching␣
↪results.
            - The first dictionary (drivers_matching) contains the matched␣
↪passenger for each driver.
            - The second dictionary (passengers_matching) contains the matched␣
↪driver for each passenger.
    """

    import warnings
    warnings.filterwarnings('ignore')
    import pandas as pd
    import numpy as np
```

```
    import random
    from geopy.geocoders import Nominatim
    from geopy.distance import geodesic
    # Set the random seed for reproducibility
#     random.seed(42)
    traffic_factors = [0.1, 0.5, 1.5]



    def get_distance_time_profit(driver_row, passenger_row):
        # Extract latitude and longitude of driver's current location
        driver_latitude, driver_longitude = [float(coord.strip()) for coord in␣
 ↪driver_row["Current_Location"].split("(")[0].split(",")]

        # Extract latitude and longitude of passenger's pickup location
        pickup_latitude, pickup_longitude = [float(coord.strip()) for coord in␣
 ↪passenger_row["Pickup_Location"].split("(")[0].split(",")]

        # Calculate distance between driver's current location and passenger's␣
 ↪pickup location
        distance_to_pickup = geodesic((driver_latitude, driver_longitude),␣
 ↪(pickup_latitude, pickup_longitude)).kilometers

        # Choose a random traffic factor from the list
        traffic_factor = random.choice(traffic_factors)

        # Calculate time to reach pickup location (distance * traffic factor)
        time_to_reach_pickup = distance_to_pickup * traffic_factor

        # Calculate profit (m * Fare)
        profit_base = 0.8 * passenger_row["Base_Fare"]
        profit_travel_to_passenger = 0.2 * distance_to_pickup
        profit_net = profit_base + profit_travel_to_passenger

        fare_base = passenger_row["Base_Fare"]
        fare_travel= 0.6 * distance_to_pickup
        fare_net = fare_base + fare_travel

        return distance_to_pickup, time_to_reach_pickup, profit_net, fare_net

    def evaluate_passengers(driver_profiles_df, passenger_profiles_df):
        # Initialize an empty list to store evaluation results
        evaluation_results = []

        # Iterate over each row (driver) in driver_profiles
        for _, driver_row in driver_profiles_df.iterrows():
            driver_id = driver_row["Driver_ID"]
```

```
        # Iterate over each row (passenger) in passenger_profiles
        for _, passenger_row in passenger_profiles_df.iterrows():
            passenger_id = passenger_row["Passenger_ID"]

            # Get the distance, time, and profit for the current␣
↪driver-passenger pair
            distance, time_to_reach, profit_net, fare_net =␣
↪get_distance_time_profit(driver_row, passenger_row)
            rating = passenger_row["Passenger_Rating"]
            destination_zone = passenger_row["Destination_District"]
            zone_matching = np.random.choice(["1", "0"])

            # Append the evaluation result to the list
            evaluation_results.append([driver_id, passenger_id, distance,␣
↪time_to_reach, profit_net, fare_net, rating, destination_zone,zone_matching])

    # Create the evaluate_passenger DataFrame
    evaluate_passenger_df = pd.DataFrame(evaluation_results,␣
↪columns=["Driver_ID", "Passenger_ID", "Distance to Pickup", "Time to Reach␣
↪Pickup", "Net Profit", "Net Fare", "Passenger_Rating", "Destination␣
↪Zone","Zone Matching"])
    evaluate_passenger_df = evaluate_passenger_df.
↪sort_values(by=["Driver_ID","Passenger_ID"])
    evaluate_passenger_df = evaluate_passenger_df.reset_index(drop=True)


    return evaluate_passenger_df

def evaluate_drivers(driver_profiles_df, evaluate_passenger_df):

    # Merge the two dataframes based on the common column "Driver_ID"
    evaluate_drivers_df = pd.merge(evaluate_passenger_df,␣
↪driver_profiles_df, on="Driver_ID")

    # Select only the desired columns
    evaluate_drivers_df = evaluate_drivers_df[["Passenger_ID", "Driver_ID",␣
↪"Driver_Rating", "Net Fare", "Time to Reach Pickup"]]

    # Sort the dataframe by Passenger_ID and then Driver_ID
    evaluate_drivers_df = evaluate_drivers_df.
↪sort_values(by=["Passenger_ID", "Driver_ID"])

    # Reset the index if needed
    evaluate_drivers_df = evaluate_drivers_df.reset_index(drop=True)
    return evaluate_drivers_df
```

```
def get_passenger_preferences_for_drivers(driver_profiles,␣
↪evaluate_passenger_df):
    # Get a list of all unique passenger IDs
    all_passengers = evaluate_passenger_df["Passenger_ID"].unique()

    # Initialize lists to store the driver IDs and preferences for each␣
↪driver
    driver_ids = []
    preferences = []

    # Iterate over each row (driver) in driver_profiles
    for _, driver_row in driver_profiles.iterrows():
        driver_id = driver_row["Driver_ID"]
        driver_ids.append(driver_id)

        # Filter passengers who were evaluated by the current driver
        driver_evaluations =␣
↪evaluate_passenger_df[evaluate_passenger_df["Driver_ID"] == driver_id]

        # Create three different preference lists for passengers based on␣
↪profit, rating, and zone matching
        passengers_profit = driver_evaluations.sort_values(by=["Net␣
↪Profit"], ascending=False)["Passenger_ID"].tolist()
        passengers_rating = driver_evaluations.
↪sort_values(by=["Passenger_Rating"], ascending=False)["Passenger_ID"].
↪tolist()
        passengers_zone_matching = driver_evaluations.sort_values(by=["Zone␣
↪Matching"], ascending=False)["Passenger_ID"].tolist()

        # Add the three preference lists for the current driver to the␣
↪preferences list
        preferences.append([passengers_profit, passengers_rating,␣
↪passengers_zone_matching])


    return driver_ids, all_passengers, preferences


def get_driver_preferences_for_passengers(evaluate_drivers_df):
    # Get a list of all unique passenger IDs
    passenger_ids = evaluate_drivers_df["Passenger_ID"].unique()

    # Initialize lists to store the passenger IDs and preferences for each␣
↪passenger
    passengers_ids_list = []
```

```python
        candidates = []
        preferences = []

        # Iterate over each passenger ID
        for passenger_id in passenger_ids:
            passengers_ids_list.append(passenger_id)

            # Filter drivers who were evaluated by the current passenger
            passenger_evaluations = evaluate_drivers_df[evaluate_drivers_df["Passenger_ID"] == passenger_id]

            # Create three different preference lists for drivers based on Net Fare, Driver Rating, and Time to Reach Pickup
            drivers_net_fare = passenger_evaluations.sort_values(by=["Net Fare"], ascending=True)["Driver_ID"].tolist()
            drivers_rating = passenger_evaluations.sort_values(by=["Driver_Rating"], ascending=False)["Driver_ID"].tolist()
            drivers_time_to_reach_pickup = passenger_evaluations.sort_values(by=["Time to Reach Pickup"], ascending=True)["Driver_ID"].tolist()

            # Add the three preference lists for the current passenger to the preferences list
            preferences.append([drivers_net_fare, drivers_rating, drivers_time_to_reach_pickup])

            # Add the drivers to the candidates list
            candidates.extend(drivers_net_fare)
            candidates.extend(drivers_rating)
            candidates.extend(drivers_time_to_reach_pickup)

        # Remove duplicates from the candidates list
        candidates = list(set(candidates))

        return passengers_ids_list, candidates, preferences


    def borda_voting_for_drivers(driver_ids, candidates, preferences):
        # Initialize a dictionary to store the points for each candidate
        candidate_points = {driver_id: {candidate: 0 for candidate in candidates} for driver_id in driver_ids}

        # Calculate the total number of voters
        num_voters = len(candidates)
```

```
        # Assign points to candidates based on their ranks in each voter's␣
↪preference order
        for i, driver_preferences in enumerate(preferences):
            for j, candidate_list in enumerate(driver_preferences):
                for k, candidate in enumerate(candidate_list):
                    points = num_voters - k - 1
                    candidate_points[driver_ids[i]][candidate] += points

        # Get the preference order for each driver based on the points
        preference_orders = {}
        for driver_id in driver_ids:
            sorted_candidates = sorted(candidate_points[driver_id].items(),␣
↪key=lambda x: x[1], reverse=True)
            preference_order = [candidate for candidate, points in␣
↪sorted_candidates]
            preference_orders[driver_id] = preference_order

#         # Print the total score received by each candidate
#         for driver_id, candidate_points_dict in candidate_points.items():
#             for candidate, points in candidate_points_dict.items():
#                 print(f"Driver {driver_id} - {candidate} received a total of␣
↪{points} points.")

        return preference_orders
    def borda_voting_for_passengers(driver_ids, candidates, preferences):
        # Initialize a dictionary to store the points for each candidate
        candidate_points = {passenger_id: {driver_id: 0 for driver_id in␣
↪candidates} for passenger_id in driver_ids}

        # Calculate the total number of voters
        num_voters = len(candidates)

        # Assign points to candidates based on their ranks in each voter's␣
↪preference order
        for i, passenger_preferences in enumerate(preferences):
            passenger_id = driver_ids[i]
#             print(f"Scores for Passenger {passenger_id}:")
            for j, driver_preference_list in enumerate(passenger_preferences):
                for k, driver in enumerate(driver_preference_list):
                    points = num_voters - k - 1
#                     print(f"    Passenger {passenger_id} ranks Driver␣
↪{driver} in position {k + 1}, assigning {points} points.")
                    candidate_points[passenger_id][driver] += points

        # Get the preference order for each driver based on the points
        passenger_preference_orders = {}
```

```
        for passenger_id in driver_ids:
            sorted_candidates = sorted(candidate_points[passenger_id].items(),␣
↪key=lambda x: x[1], reverse=True)
            preference_order = [candidate for candidate, points in␣
↪sorted_candidates]
            passenger_preference_orders[passenger_id] = preference_order

#          # Print the total score received by each candidate
#          for passenger_id, candidate_points_dict in candidate_points.items():
#              for driver_id, points in candidate_points_dict.items():
#                  print(f"Passenger {passenger_id} - Driver {driver_id}␣
↪received a total of {points} points.")

        return passenger_preference_orders


    def match_driver_passenger(drivers_preferences, passengers_preferences):
        # Number of drivers/passengers
        n = len(drivers_preferences)

        # Initialize an empty matching for drivers and passengers
        drivers_matching = {}
        passengers_matching = {}

        # Initialize each driver to be free
        drivers_free = {driver: True for driver in drivers_preferences.keys()}

        while True:
            # Find an unmatched driver
            driver = next((driver for driver, is_free in drivers_free.items()␣
↪if is_free), None)

            if driver is None:
                # All drivers are matched
                break

            # Iterate over the preferences of the driver
            for passenger in drivers_preferences[driver]:
                # Check if the passenger is free
                if passenger not in passengers_matching:
                    # The passenger is free, so match the driver and passenger
                    drivers_matching[driver] = passenger
                    passengers_matching[passenger] = driver
                    drivers_free[driver] = False
                    break
                else:
                    # The passenger is currently matched
```

```
                matched_driver = passengers_matching[passenger]

                # Check if the passenger prefers the new driver over the current match
                if passengers_preferences[passenger].index(driver) < passengers_preferences[passenger].index(matched_driver):
                    # The passenger prefers the new driver, so update the matching
                    drivers_matching[driver] = passenger
                    drivers_matching[matched_driver] = None
                    passengers_matching[passenger] = driver
                    drivers_free[driver] = False
                    drivers_free[matched_driver] = True
                    break

    return drivers_matching, passengers_matching


evaluate_passenger_df= evaluate_passengers(driver_profiles_df, passenger_profiles_df)
evaluate_drivers_df= evaluate_drivers(driver_profiles_df, evaluate_passenger_df)

# Store preference lists for each driver according to Borda Voting Rule
driver_ids = []
preferences_of_drivers = []
driver_ids, all_passengers, preferences_of_drivers = get_passenger_preferences_for_drivers(driver_profiles_df, evaluate_passenger_df)
drivers_preference_orders_borda = borda_voting_for_drivers(driver_ids, all_passengers, preferences_of_drivers)


# Store preference lists for each passenger according to Borda Voting Rule
passenger_ids, all_driver_candidates, passenger_preferences = get_driver_preferences_for_passengers(evaluate_drivers_df)
passenger_preference_orders_borda = borda_voting_for_passengers(passenger_ids, all_driver_candidates, passenger_preferences)


drivers_matching, passengers_matching = match_driver_passenger(drivers_preference_orders_borda, passenger_preference_orders_borda)
return drivers_matching
```

```
[ ]: # 7.2.3 Create matching for Case 3 (where we score Users with Weighted Borda␣
     ↪Voting Rule)

     ## [Each Layer of Preference has Weights Assigned to them while calculating␣
     ↪Borda Scores]
     ### Give the Driver and Passengers Profiles, as well as the Weights for each␣
     ↪Preference Layers as input.
```

```
[4]: def create_matching_case_3(driver_profiles_df, passenger_profiles_df,␣
     ↪weight_layer1, weight_layer2, weight_layer3):


         """
         Perform matching between drivers and passengers based on their profiles,␣
     ↪preferences, and additional weights for different preference layers using␣
     ↪the Weighted Borda voting rule.

         Parameters:
             driver_profiles_df (pandas.DataFrame): DataFrame containing driver␣
     ↪profiles with columns 'Driver_ID', 'Driver_Rating', 'Preferred_Zones',␣
     ↪'Current_Location', and 'Preference_Orders_for_Drivers'.
             passenger_profiles_df (pandas.DataFrame): DataFrame containing␣
     ↪passenger profiles with columns 'Passenger_ID', 'Passenger_Rating',␣
     ↪'Pickup_Location', 'Destination_Location', 'Distance_km',␣
     ↪'Destination_District', 'Base_Fare', and 'Preference_Orders_for_Passengers'.
             weight_layer1 (float): Weight for the first preference layer.
             weight_layer2 (float): Weight for the second preference layer.
             weight_layer3 (float): Weight for the third preference layer.

         Returns:
             tuple: A tuple containing two dictionaries representing the matching␣
     ↪results.
                 - The first dictionary (drivers_matching) contains the matched␣
     ↪passenger for each driver.
                 - The second dictionary (passengers_matching) contains the matched␣
     ↪driver for each passenger.
         """


         import warnings
         warnings.filterwarnings('ignore')
         import pandas as pd
         import numpy as np
         import random
         from geopy.geocoders import Nominatim
         from geopy.distance import geodesic
```

```
    # Set the random seed for reproducibility
#     random.seed(42)
    traffic_factors = [0.1, 0.5, 1.5]




    def get_distance_time_profit(driver_row, passenger_row):
        # Extract latitude and longitude of driver's current location
        driver_latitude, driver_longitude = [float(coord.strip()) for coord in␣
 ↪driver_row["Current_Location"].split("(")[0].split(",")]

        # Extract latitude and longitude of passenger's pickup location
        pickup_latitude, pickup_longitude = [float(coord.strip()) for coord in␣
 ↪passenger_row["Pickup_Location"].split("(")[0].split(",")]

        # Calculate distance between driver's current location and passenger's␣
 ↪pickup location
        distance_to_pickup = geodesic((driver_latitude, driver_longitude),␣
 ↪(pickup_latitude, pickup_longitude)).kilometers

        # Choose a random traffic factor from the list
        traffic_factor = random.choice(traffic_factors)

        # Calculate time to reach pickup location (distance * traffic factor)
        time_to_reach_pickup = distance_to_pickup * traffic_factor

        # Calculate profit (m * Fare)
        profit_base = 0.8 * passenger_row["Base_Fare"]
        profit_travel_to_passenger = 0.2 * distance_to_pickup
        profit_net = profit_base + profit_travel_to_passenger

        fare_base = passenger_row["Base_Fare"]
        fare_travel= 0.6 * distance_to_pickup
        fare_net = fare_base + fare_travel

        return distance_to_pickup, time_to_reach_pickup, profit_net, fare_net

    def evaluate_passengers(driver_profiles_df, passenger_profiles_df):
        # Initialize an empty list to store evaluation results
        evaluation_results = []

        # Iterate over each row (driver) in driver_profiles
        for _, driver_row in driver_profiles_df.iterrows():
            driver_id = driver_row["Driver_ID"]

            # Iterate over each row (passenger) in passenger_profiles
```

```
            for _, passenger_row in passenger_profiles_df.iterrows():
                passenger_id = passenger_row["Passenger_ID"]

                # Get the distance, time, and profit for the current␣
↪driver-passenger pair
                distance, time_to_reach, profit_net, fare_net =␣
↪get_distance_time_profit(driver_row, passenger_row)
                rating = passenger_row["Passenger_Rating"]
                destination_zone = passenger_row["Destination_District"]
                zone_matching = np.random.choice(["1", "0"])

                # Append the evaluation result to the list
                evaluation_results.append([driver_id, passenger_id, distance,␣
↪time_to_reach, profit_net, fare_net, rating, destination_zone,zone_matching])

        # Create the evaluate_passenger DataFrame
        evaluate_passenger_df = pd.DataFrame(evaluation_results,␣
↪columns=["Driver_ID", "Passenger_ID", "Distance to Pickup", "Time to Reach␣
↪Pickup", "Net Profit", "Net Fare", "Passenger_Rating", "Destination␣
↪Zone","Zone Matching"])
        evaluate_passenger_df = evaluate_passenger_df.
↪sort_values(by=["Driver_ID","Passenger_ID"])
        evaluate_passenger_df = evaluate_passenger_df.reset_index(drop=True)


        return evaluate_passenger_df

    def evaluate_drivers(driver_profiles_df, evaluate_passenger_df):

        # Merge the two dataframes based on the common column "Driver_ID"
        evaluate_drivers_df = pd.merge(evaluate_passenger_df,␣
↪driver_profiles_df, on="Driver_ID")

        # Select only the desired columns
        evaluate_drivers_df = evaluate_drivers_df[["Passenger_ID", "Driver_ID",␣
↪"Driver_Rating", "Net Fare", "Time to Reach Pickup"]]

        # Sort the dataframe by Passenger_ID and then Driver_ID
        evaluate_drivers_df = evaluate_drivers_df.
↪sort_values(by=["Passenger_ID", "Driver_ID"])

        # Reset the index if needed
        evaluate_drivers_df = evaluate_drivers_df.reset_index(drop=True)
        return evaluate_drivers_df
```

```python
def get_passenger_preferences_for_drivers(driver_profiles, evaluate_passenger_df):
    # Get a list of all unique passenger IDs
    all_passengers = evaluate_passenger_df["Passenger_ID"].unique()

    # Initialize lists to store the driver IDs and preferences for each driver
    driver_ids = []
    preferences = []

    # Iterate over each row (driver) in driver_profiles
    for _, driver_row in driver_profiles.iterrows():
        driver_id = driver_row["Driver_ID"]
        driver_ids.append(driver_id)

        # Filter passengers who were evaluated by the current driver
        driver_evaluations = evaluate_passenger_df[evaluate_passenger_df["Driver_ID"] == driver_id]

        # Create three different preference lists for passengers based on profit, rating, and zone matching
        passengers_profit = driver_evaluations.sort_values(by=["Net Profit"], ascending=False)["Passenger_ID"].tolist()
        passengers_rating = driver_evaluations.sort_values(by=["Passenger_Rating"], ascending=False)["Passenger_ID"].tolist()
        passengers_zone_matching = driver_evaluations.sort_values(by=["Zone Matching"], ascending=False)["Passenger_ID"].tolist()

        # Add the three preference lists for the current driver to the preferences list
        preferences.append([passengers_profit, passengers_rating, passengers_zone_matching])


    return driver_ids, all_passengers, preferences


def get_driver_preferences_for_passengers(evaluate_drivers_df):
    # Get a list of all unique passenger IDs
    passenger_ids = evaluate_drivers_df["Passenger_ID"].unique()

    # Initialize lists to store the passenger IDs and preferences for each passenger
    passengers_ids_list = []
    candidates = []
```

```
        preferences = []

        # Iterate over each passenger ID
        for passenger_id in passenger_ids:
            passengers_ids_list.append(passenger_id)

            # Filter drivers who were evaluated by the current passenger
            passenger_evaluations = evaluate_drivers_df[evaluate_drivers_df["Passenger_ID"] == passenger_id]

            # Create three different preference lists for drivers based on Net Fare, Driver Rating, and Time to Reach Pickup
            drivers_net_fare = passenger_evaluations.sort_values(by=["Net Fare"], ascending=True)["Driver_ID"].tolist()
            drivers_rating = passenger_evaluations.sort_values(by=["Driver_Rating"], ascending=False)["Driver_ID"].tolist()
            drivers_time_to_reach_pickup = passenger_evaluations.sort_values(by=["Time to Reach Pickup"], ascending=True)["Driver_ID"].tolist()

            # Add the three preference lists for the current passenger to the preferences list
            preferences.append([drivers_net_fare, drivers_rating, drivers_time_to_reach_pickup])

            # Add the drivers to the candidates list
            candidates.extend(drivers_net_fare)
            candidates.extend(drivers_rating)
            candidates.extend(drivers_time_to_reach_pickup)

        # Remove duplicates from the candidates list
        candidates = list(set(candidates))

        return passengers_ids_list, candidates, preferences

    def borda_voting_for_drivers(driver_ids, candidates, preferences, weights):
        # Initialize a dictionary to store the points for each candidate
        candidate_points = {driver_id: {candidate: 0 for candidate in candidates} for driver_id in driver_ids}

        # Calculate the total number of voters
        num_voters = len(candidates)

        # Assign points to candidates based on their ranks in each voter's preference order
        for i, driver_preferences in enumerate(preferences):
```

```
            for j, candidate_list in enumerate(driver_preferences):
                for k, candidate in enumerate(candidate_list):
                    points = (num_voters - k - 1) * weights[j]  # Apply the␣
↪weights to the points
                    candidate_points[driver_ids[i]][candidate] += points

        # Get the preference order for each driver based on the points
        preference_orders = {}
        for driver_id in driver_ids:
            sorted_candidates = sorted(candidate_points[driver_id].items(),␣
↪key=lambda x: x[1], reverse=True)
            preference_order = [candidate for candidate, points in␣
↪sorted_candidates]
            preference_orders[driver_id] = preference_order

#          # Print the total score received by each candidate
#          for driver_id, candidate_points_dict in candidate_points.items():
#              for candidate, points in candidate_points_dict.items():
#                  print(f"Driver {driver_id} - {candidate} received a total of␣
↪{points} points.")


        return preference_orders

    def borda_voting_for_passengers(driver_ids, candidates, preferences,␣
↪weights):
        # Initialize a dictionary to store the points for each candidate
        candidate_points = {passenger_id: {driver_id: 0 for driver_id in␣
↪candidates} for passenger_id in driver_ids}

        # Calculate the total number of voters
        num_voters = len(candidates)

        # Assign points to candidates based on their ranks in each voter's␣
↪preference order
        for i, passenger_preferences in enumerate(preferences):
            passenger_id = driver_ids[i]
            for j, driver_preference_list in enumerate(passenger_preferences):
                for k, driver in enumerate(driver_preference_list):
                    points = (num_voters - k - 1) * weights[j]  # Apply the␣
↪weights to the points
#                      print(f"    Passenger {passenger_id} ranks Driver␣
↪{driver} in position {k + 1}, assigning {points} points.")

                    candidate_points[passenger_id][driver] += points
```

```
        # Get the preference order for each driver based on the points
        passenger_preference_orders = {}
        for passenger_id in driver_ids:
            sorted_candidates = sorted(candidate_points[passenger_id].items(),␣
↪key=lambda x: x[1], reverse=True)
            preference_order = [candidate for candidate, points in␣
↪sorted_candidates]
            passenger_preference_orders[passenger_id] = preference_order

#         # Print the total score received by each candidate
#         for passenger_id, candidate_points_dict in candidate_points.items():
#             for driver_id, points in candidate_points_dict.items():
#                 print(f"Passenger {passenger_id} - Driver {driver_id}␣
↪received a total of {points} points.")


        return passenger_preference_orders

    def match_driver_passenger(drivers_preferences, passengers_preferences):
        # Number of drivers/passengers
        n = len(drivers_preferences)

        # Initialize an empty matching for drivers and passengers
        drivers_matching = {}
        passengers_matching = {}

        # Initialize each driver to be free
        drivers_free = {driver: True for driver in drivers_preferences.keys()}

        while True:
            # Find an unmatched driver
            driver = next((driver for driver, is_free in drivers_free.items()␣
↪if is_free), None)

            if driver is None:
                # All drivers are matched
                break

            # Iterate over the preferences of the driver
            for passenger in drivers_preferences[driver]:
                # Check if the passenger is free
                if passenger not in passengers_matching:
                    # The passenger is free, so match the driver and passenger
                    drivers_matching[driver] = passenger
                    passengers_matching[passenger] = driver
                    drivers_free[driver] = False
                    break
```

```
            else:
                # The passenger is currently matched
                matched_driver = passengers_matching[passenger]

                # Check if the passenger prefers the new driver over the␣
↪current match
                if passengers_preferences[passenger].index(driver) <␣
↪passengers_preferences[passenger].index(matched_driver):
                    # The passenger prefers the new driver, so update the␣
↪matching
                    drivers_matching[driver] = passenger
                    drivers_matching[matched_driver] = None
                    passengers_matching[passenger] = driver
                    drivers_free[driver] = False
                    drivers_free[matched_driver] = True
                    break

    return drivers_matching, passengers_matching

evaluate_passenger_df= evaluate_passengers(driver_profiles_df,␣
↪passenger_profiles_df)
evaluate_drivers_df= evaluate_drivers(driver_profiles_df,␣
↪evaluate_passenger_df)

# Store preference lists for each driver according to Borda Voting Rule
driver_ids = []
preferences_of_drivers = []
driver_ids, all_passengers, preferences_of_drivers =␣
↪get_passenger_preferences_for_drivers(driver_profiles_df,␣
↪evaluate_passenger_df)
drivers_preference_orders_borda = borda_voting_for_drivers(driver_ids,␣
↪all_passengers, preferences_of_drivers, [weight_layer1, weight_layer2,␣
↪weight_layer3])


# Store preference lists for each passenger according to Borda Voting Rule
passenger_ids, all_driver_candidates, passenger_preferences =␣
↪get_driver_preferences_for_passengers(evaluate_drivers_df)
passenger_preference_orders_borda =␣
↪borda_voting_for_passengers(passenger_ids, all_driver_candidates,␣
↪passenger_preferences, [weight_layer1, weight_layer2, weight_layer3])


drivers_matching, passengers_matching =␣
↪match_driver_passenger(drivers_preference_orders_borda,␣
↪passenger_preference_orders_borda)
```

```
    return drivers_matching
```

[ ]:
```
# 7.2.4 Create matching for Case 4 (where we do not consider Layer Preferences
 ↪for each User, instead create matching for a pair of prefrences between
 ↪Drivers and Passengers)
### Give the Driver and Passengers Profiles as input, along with the two
 ↪preferences you want to select.
### Preference Option for Drivers : Max Profit (P), Passenger Rating (Rp), Zone
 ↪Matching (Z).
### Preference Option for Passengers: Min Waiting Time (W), Min Fare (F),
 ↪Driver Rating (Rd).
```

[5]:
```
def create_matching_case_4 (driver_profiles_df, passenger_profiles_df,
 ↪driver_preference_type, passenger_preference_type):


    """
    Perform matching between drivers and passengers based on their profiles and
 ↪preference types.

    Parameters:
        driver_profiles_df (pandas.DataFrame): DataFrame containing driver
 ↪profiles with columns 'Driver_ID', 'Driver_Rating', 'Preferred_Zones',
 ↪'Current_Location', and 'Preference_Orders_for_Drivers'.
        passenger_profiles_df (pandas.DataFrame): DataFrame containing
 ↪passenger profiles with columns 'Passenger_ID', 'Passenger_Rating',
 ↪'Pickup_Location', 'Destination_Location', 'Distance_km',
 ↪'Destination_District', 'Base_Fare', and 'Preference_Orders_for_Passengers'.
        driver_preference_type (str): Driver's preference type for sorting
 ↪passengers. Available options are 'Rp' (for passenger rating),'P' (for
 ↪profit), and 'Z' (for Zone matching).
        passenger_preference_type (str): Passenger's preference type for
 ↪sorting drivers. Available options are 'Rd' (for driver rating),'F' (for
 ↪fare), and 'W' (for waiting time).

    Returns:
        tuple: A tuple containing two dictionaries representing the matching
 ↪results.
            - The first dictionary (drivers_matching) contains the matched
 ↪passenger for each driver.
            - The second dictionary (passengers_matching) contains the matched
 ↪driver for each passenger.
    """

    import warnings
    warnings.filterwarnings('ignore')
```

```
import pandas as pd
import numpy as np
import random
from geopy.geocoders import Nominatim
from geopy.distance import geodesic
# Set the random seed for reproducibility
#     random.seed(42)
traffic_factors = [0.1, 0.5, 1.5]



def get_distance_time_profit(driver_row, passenger_row):
    # Extract latitude and longitude of driver's current location
    driver_latitude, driver_longitude = [float(coord.strip()) for coord in␣
↪driver_row["Current_Location"].split("(")[0].split(",")]

    # Extract latitude and longitude of passenger's pickup location
    pickup_latitude, pickup_longitude = [float(coord.strip()) for coord in␣
↪passenger_row["Pickup_Location"].split("(")[0].split(",")]

    # Calculate distance between driver's current location and passenger's␣
↪pickup location
    distance_to_pickup = geodesic((driver_latitude, driver_longitude),␣
↪(pickup_latitude, pickup_longitude)).kilometers

    # Choose a random traffic factor from the list
    traffic_factor = random.choice(traffic_factors)

    # Calculate time to reach pickup location (distance * traffic factor)
    time_to_reach_pickup = distance_to_pickup * traffic_factor

    # Calculate profit (m * Fare)
    profit_base = 0.8 * passenger_row["Base_Fare"]
    profit_travel_to_passenger = 0.2 * distance_to_pickup
    profit_net = profit_base + profit_travel_to_passenger

    fare_base = passenger_row["Base_Fare"]
    fare_travel= 0.6 * distance_to_pickup
    fare_net = fare_base + fare_travel

    return distance_to_pickup, time_to_reach_pickup, profit_net, fare_net

def evaluate_passengers(driver_profiles_df, passenger_profiles_df):
    # Initialize an empty list to store evaluation results
    evaluation_results = []

    # Iterate over each row (driver) in driver_profiles
```

```
        for _, driver_row in driver_profiles_df.iterrows():
            driver_id = driver_row["Driver_ID"]

            # Iterate over each row (passenger) in passenger_profiles
            for _, passenger_row in passenger_profiles_df.iterrows():
                passenger_id = passenger_row["Passenger_ID"]

                # Get the distance, time, and profit for the current␣
↪driver-passenger pair
                distance, time_to_reach, profit_net, fare_net =␣
↪get_distance_time_profit(driver_row, passenger_row)
                rating = passenger_row["Passenger_Rating"]
                destination_zone = passenger_row["Destination_District"]
                zone_matching = np.random.choice(["1", "0"])

                # Append the evaluation result to the list
                evaluation_results.append([driver_id, passenger_id, distance,␣
↪time_to_reach, profit_net, fare_net, rating, destination_zone,zone_matching])

        # Create the evaluate_passenger DataFrame
        evaluate_passenger_df = pd.DataFrame(evaluation_results,␣
↪columns=["Driver_ID", "Passenger_ID", "Distance to Pickup", "Time to Reach␣
↪Pickup", "Net Profit", "Net Fare", "Passenger_Rating", "Destination␣
↪Zone","Zone Matching"])
        evaluate_passenger_df = evaluate_passenger_df.
↪sort_values(by=["Driver_ID","Passenger_ID"])
        evaluate_passenger_df = evaluate_passenger_df.reset_index(drop=True)


        return evaluate_passenger_df

    def evaluate_drivers(driver_profiles_df, evaluate_passenger_df):

        # Merge the two dataframes based on the common column "Driver_ID"
        evaluate_drivers_df = pd.merge(evaluate_passenger_df,␣
↪driver_profiles_df, on="Driver_ID")

        # Select only the desired columns
        evaluate_drivers_df = evaluate_drivers_df[["Passenger_ID", "Driver_ID",␣
↪"Driver_Rating", "Net Fare", "Time to Reach Pickup"]]

        # Sort the dataframe by Passenger_ID and then Driver_ID
        evaluate_drivers_df = evaluate_drivers_df.
↪sort_values(by=["Passenger_ID", "Driver_ID"])

        # Reset the index if needed
```

```
        evaluate_drivers_df = evaluate_drivers_df.reset_index(drop=True)
        return evaluate_drivers_df


    def create_preference_list_for_drivers(driver_profiles_df,␣
↪passenger_profiles_df, evaluate_passenger_df, driver_preference_type):
        # Initialize an empty dictionary to store preference lists for each␣
↪driver


        # Iterate over each row (driver) in driver_profiles
        for _, driver_row in driver_profiles_df.iterrows():
            driver_id = driver_row["Driver_ID"]
            preference_order = driver_preference_type
    #         print(driver_id)
    #         print(preference_order)

    #         Filter passengers who were evaluated by the current driver
            driver_evaluations =␣
↪evaluate_passenger_df[evaluate_passenger_df["Driver_ID"] == driver_id]
    #         print(driver_evaluations)
    #         print("----------")
#             print("Drivers Preference type:", preference_order )

            if   preference_order == "Rp" :
                driver_evaluations.sort_values(by=["Passenger_Rating"],␣
↪ascending=[False], inplace=True)
            elif preference_order == "P" :
                driver_evaluations.sort_values(by=["Net Profit"],␣
↪ascending=[False], inplace=True)
            elif preference_order == "Z" :
                driver_evaluations.sort_values(by=["Zone Matching"],␣
↪ascending=[False], inplace=True)
            else:
                print("Please Provide Driver preference in correct format.␣
↪Available options are 'Rp' (for passenger rating),'P' (for profit) and 'Z'␣
↪(for Zone matching) ")
                break;

    #         print(driver_evaluations)
    #         print("----------")
    #         print("@@@@@@@@@@@@@")


            # Add the sorted passenger IDs to the preference list
```

```
            preference_lists_for_drivers[driver_id] =
 ↪driver_evaluations["Passenger_ID"].tolist()
#             print(preference_lists_for_drivers)

        return preference_lists_for_drivers

    def create_preference_list_for_passengers(passenger_profiles_df,
 ↪evaluate_drivers_df, passenger_preference_type):

        # Iterate over each row (passenger) in passenger_profiles
        for _, passenger_row in passenger_profiles_df.iterrows():
            passenger_id = passenger_row["Passenger_ID"]
            preference_order = passenger_preference_type

            # Filter drivers who were evaluated by the current passenger
            passenger_evaluations =
 ↪evaluate_drivers_df[evaluate_drivers_df["Passenger_ID"] == passenger_id]


#             print("Passenger Preference type:", preference_order )

            # Sort drivers based on passenger's preference order

            if preference_order == "Rd" :
                passenger_evaluations.sort_values(by=["Driver_Rating"],
 ↪ascending=[False], inplace=True)
            elif preference_order == "W" :
                passenger_evaluations.sort_values(by=["Time to Reach Pickup"],
 ↪ascending=[True], inplace=True)
            elif preference_order == "F" :
                passenger_evaluations.sort_values(by=["Net Fare"],
 ↪ascending=[True], inplace=True)
            else:
                print("Please Provide Passenger preference in correct format.
 ↪Available options are 'Rd' (for driver rating),'F' (for fare) and 'W' (for
 ↪waiting time) ")
                break;

            # Add the sorted driver IDs to the preference list
            preference_lists_for_passengers[passenger_id] =
 ↪passenger_evaluations["Driver_ID"].tolist()
#             print(preference_lists_for_passengers)

        return preference_lists_for_passengers

    def match_driver_passenger(drivers_preferences, passengers_preferences):
```

```python
    # Number of drivers/passengers
    n = len(drivers_preferences)

    # Initialize an empty matching for drivers and passengers
    drivers_matching = {}
    passengers_matching = {}

    # Initialize each driver to be free
    drivers_free = {driver: True for driver in drivers_preferences.keys()}

    while True:
        # Find an unmatched driver
        driver = next((driver for driver, is_free in drivers_free.items() if is_free), None)

        if driver is None:
            # All drivers are matched
            break

        # Iterate over the preferences of the driver
        for passenger in drivers_preferences[driver]:
            # Check if the passenger is free
            if passenger not in passengers_matching:
                # The passenger is free, so match the driver and passenger
                drivers_matching[driver] = passenger
                passengers_matching[passenger] = driver
                drivers_free[driver] = False
                break
            else:
                # The passenger is currently matched
                matched_driver = passengers_matching[passenger]

                # Check if the passenger prefers the new driver over the current match
                if passengers_preferences[passenger].index(driver) < passengers_preferences[passenger].index(matched_driver):
                    # The passenger prefers the new driver, so update the matching
                    drivers_matching[driver] = passenger
                    drivers_matching[matched_driver] = None
                    passengers_matching[passenger] = driver
                    drivers_free[driver] = False
                    drivers_free[matched_driver] = True
                    break

    return drivers_matching, passengers_matching
```

```
    evaluate_passenger_df= evaluate_passengers(driver_profiles_df,␣
↪passenger_profiles_df)
    evaluate_drivers_df= evaluate_drivers(driver_profiles_df,␣
↪evaluate_passenger_df)

    # Initialize an empty dictionary to store preference lists for each driver
    preference_lists_for_drivers = {}
    preference_lists_for_drivers =␣
↪create_preference_list_for_drivers(driver_profiles_df,␣
↪passenger_profiles_df, evaluate_passenger_df, driver_preference_type)

    # Initialize an empty dictionary to store preference lists for each␣
↪passenger
    preference_lists_for_passengers = {}
    preference_lists_for_passengers =␣
↪create_preference_list_for_passengers(passenger_profiles_df,␣
↪evaluate_drivers_df, passenger_preference_type)

    drivers_matching, passengers_matching =␣
↪match_driver_passenger(preference_lists_for_drivers,␣
↪preference_lists_for_passengers)
    return drivers_matching
```

[ ]:
```
# 7.3 Sample Results:
```

Creating sample data for Drivers and Passengers:

[6]:
```
d,p=create_driver_passenger_profiles(num_drivers=5,num_passengers=5)
```

[7]:
```
d.head()
```

[7]:
```
   Driver_ID  Driver_Rating  \
0  driver328            2.6
1   driver58            3.0
2   driver13            4.6
3  driver380            4.1
4  driver141            5.4

                                     Preferred_Zones  \
0  Marzahn-Hellersdorf, Treptow-Köpenick, Lichten…
1  Treptow-Köpenick, Friedrichshain-Kreuzberg, Sp…
2  Friedrichshain-Kreuzberg, Steglitz-Zehlendorf,…
3  Charlottenburg-Wilmersdorf, Treptow-Köpenick, …
4     Marzahn-Hellersdorf, Treptow-Köpenick, Spandau

                                    Current_Location  \
```

```
0  52.630033, 13.512219 (HELIOS Klinikum Berlin B…
1  52.546058, 13.756170 (44, Goethestraße, Freder…
2  52.628498, 13.588984 (Löhmer Weg, Birkholz, Be…
3  52.444755, 13.512833 (15, Segelfliegerdamm, Jo…
4  52.655531, 13.310904 (Referenzfläche, Franzisk…

  Preference_Orders_for_Drivers
0                   Rp → Z → P
1                   Rp → Z → P
2                   Z → Rp → P
3                   Z → Rp → P
4                   Rp → P → Z
```

[8]:
```
p.head()
```

[8]:
```
   Passenger_ID  Passenger_Rating  \
0  passenger230               3.0
1  passenger302               2.1
2  passenger143               3.2
3  passenger415               3.4
4  passenger446               3.5

                                     Pickup_Location  \
0  52.396094, 13.442219 (163, Karl-Marx-Straße, S…
1  52.367454, 13.534339 (P108, Käthe-Paulus-Allee…
2  52.629855, 13.746269 (4, Ligusterweg, Rudolfsh…
3  52.601870, 13.486266 (39A, Straße 67, Karow, P…
4  52.479086, 13.618478 (11, Schopenhauerstraße, …

                                Destination_Location  Distance_km  \
0  52.572154, 13.517811 (Neubrandenburger Straße …        20.25
1  52.348703, 13.490756 (Zaunstraße 1, Schönefeld…         3.63
2  52.519304, 13.193274 (46, Weverstraße, Wilhelm…        39.46
3  52.561076, 13.362882 (7, Walderseestraße, Rein…         9.52
4  52.512052, 13.215793 (Brandensteinweg, Wilhelm…        27.59

   Destination_District  Base_Fare Preference_Orders_for_Passengers
0  Neu-Hohenschönhausen      30.38                       W → F → Rd
1             Schönefeld      5.44                       Rd → W → F
2           Wilhelmstadt     59.19                       Rd → W → F
3          Reinickendorf     14.27                       F → Rd → W
4           Wilhelmstadt     41.39                       W → F → Rd
```

Testing each type of algorithm on this sample data generated:

[9]:
```
result_case1 = create_matching_case_1 (driver_profiles_df = d,␣
↪passenger_profiles_df = p)
```

```
result_case1
```

[9]:
```
{'driver328': 'passenger415',
 'driver58': 'passenger446',
 'driver13': 'passenger302',
 'driver380': 'passenger230',
 'driver141': 'passenger143'}
```

[10]:
```
result_case2 = create_matching_case_2 (driver_profiles_df = d,␣
 ↪passenger_profiles_df = p)
result_case2
```

[10]:
```
{'driver328': 'passenger415',
 'driver58': 'passenger302',
 'driver13': 'passenger143',
 'driver380': 'passenger230',
 'driver141': 'passenger446'}
```

[11]:
```
result_case3 = create_matching_case_3 (driver_profiles_df = d,␣
 ↪passenger_profiles_df = p,
                                       weight_layer1 = 10,
                                       weight_layer2 = 5,
                                       weight_layer3 = 1)



result_case3
```

[11]:
```
{'driver328': 'passenger230',
 'driver58': 'passenger302',
 'driver13': 'passenger143',
 'driver380': 'passenger446',
 'driver141': 'passenger415'}
```

[12]:
```
result_case3 = create_matching_case_3 (driver_profiles_df = d,␣
 ↪passenger_profiles_df = p,
                                       weight_layer1 = 15,
                                       weight_layer2 = 10,
                                       weight_layer3 = 5)



result_case3
```

[12]:
```
{'driver328': 'passenger302',
 'driver58': 'passenger230',
 'driver13': 'passenger143',
 'driver380': 'passenger446',
 'driver141': 'passenger415'}
```

[13]:
```
result_case4 = create_matching_case_4 (driver_profiles_df = d,␣
 ↪passenger_profiles_df = p,
                                       driver_preference_type = 'P',␣
 ↪passenger_preference_type = 'W')


result_case4
```

[13]:
```
{'driver328': 'passenger415',
 'driver58': 'passenger143',
 'driver13': 'passenger302',
 'driver380': 'passenger230',
 'driver141': 'passenger446'}
```

[14]:
```
result_case4 = create_matching_case_4 (driver_profiles_df = d,␣
 ↪passenger_profiles_df = p,
                                       driver_preference_type = 'Z',␣
 ↪passenger_preference_type = 'F')


result_case4
```

[14]:
```
{'driver328': 'passenger302',
 'driver58': 'passenger143',
 'driver13': 'passenger446',
 'driver380': 'passenger415',
 'driver141': 'passenger230'}
```

[15]:
```
result_case4 = create_matching_case_4 (driver_profiles_df = d,␣
 ↪passenger_profiles_df = p,
                                       driver_preference_type = 'Rp',␣
 ↪passenger_preference_type = 'Rd')


result_case4
```

[15]:
```
{'driver328': 'passenger302',
 'driver58': 'passenger230',
 'driver13': 'passenger415',
 'driver380': 'passenger143',
 'driver141': 'passenger446'}
```

[ ]:
```

```