Drivers' Rout(in)e

Agnese Cervino
agnese.cervino@studenti.unitn.it
Università di Trento
Trento, Italy

Enrico Guerriero
enrico.guerriero@studenti.unitn.it
Università di Trento
Trento, Italy

Andrea Leoni
andrea.leoni-1@studenti.unitn.it
Università di Trento
Trento, Italy

Damiano Zaccaron
damiano.zaccaron@studenti.unitn.it
Università di Trento
Trento, Italy

KEYWORDS

Datasets, Data mining, Clustering, Recommendation system, Frequent itemsets

1 INTRODUCTION

In today's data-driven world, data mining techniques offer powerful solutions to a wide range of challenges. This report explores a data mining problem within the logistics and merchandise transport sector, focusing on a potential issue faced by companies: drivers who do not always adhere to the provided instructions. In this scenario, companies provide drivers with standard routes composed of individual trips, each defined by a starting city, a destination city, and a list of transported goods along with their respective quantities. To address this challenge, the authors propose three algorithmic solutions with distinct outcomes:

- The first solution takes as input all standard routes required by the company and the actual routes traveled by drivers. It then generates an alternative set of standard routes that drivers are more likely to follow.
- Using the same input as above, the second solution identifies, for each driver, the five standard routes (among those already present in the company's database) they are most likely to adhere to.
- The third solution, relying solely on the routes traveled by each driver, constructs an ideal standard route tailored specifically to that driver's behavior.

These outcomes represent three distinct approaches to enhancing efficiency in work organization. They are independent and can be implemented individually, providing flexibility depending on the specific requirements of the organization.
In the data generation process, routes are assigned uniformly to the drivers. This assumption influences the output of the proposed algorithms. However, the implementation ensures that the output remains optimal even if real-world data deviates from this assumption.

To address the first problem, clustering techniques were applied to the actual routes traveled by drivers. The number of clusters generated was set equal to the number of initial standard routes. The centroid of each cluster was then identified and used as the recommended route for that group.

For the second problem, all actual routes traveled by each driver were aggregated into an object referred to as *preferences*. Each standard route was compared to this object, and a similarity score was computed. The five standard routes with the highest similarity scores were selected as the driver's preferred routes.

The solution to the third problem builds on the *preferences* object created for the second solution. The ideal standard route is constructed by identifying favorite trips, cities, and merchandise, each weighted according to their significance in the driver's history.

Experimental Evaluation. The success of the first solution was assessed using three evaluation metrics. The dataset was divided into a training set (80%) and a test set (20%). After generating the recommended standard routes from the training set, two sets of distances were calculated:

1. The distances between the recommended routes and all actual routes in the test set.
2. The distances between the actual routes in the test set and the nearest standard routes.

These two sets of distances were then compared to evaluate performance.

Two methods were used to evaluate the second solution. First, the dataset was divided into a training set and a test set. The robustness of the solution was tested by projecting the results from the training set onto the test set. Second, the significance of the five recommended standard routes for each driver was compared to all other standard routes, using the average distance from the driver's *preferences* as the metric.

The third solution was evaluated similarly to the second. A training set was used to generate *preferences* and construct the ideal route. The *preferences* were then updated using the test set, and the similarity between the ideal route and the updated *preferences* was computed.

Data Mining Exam, January 2024, Trento, Italy
2025.

2 RELATED WORKS

In this section, an overview of the key technologies, theories, and techniques that underpin the methodologies applied in this work is provided. The aim is to equip the reader with the necessary background to better understand the solutions proposed in subsequent sections. Readers with a strong familiarity with these topics may choose to skip this section.

K-Means Clustering

K-Means clustering is an iterative algorithm used to partition data into distinct, non-overlapping clusters, where each data point belongs to exactly one cluster. The algorithm proceeds as follows:

(1) Initially, K random points are selected as centroids, where K is a predetermined number of clusters.
(2) Each data point in the set is assigned to the nearest centroid, forming K clusters.
(3) For each cluster, a new centroid is calculated as the mean of all points within that cluster.
(4) The process repeats, with data points reassigned to the nearest updated centroids.
(5) The algorithm terminates when no data points are reassigned to a different cluster after updating the centroids.

2.1 Suffix Tree

A suffix tree is a data structure that compactly represents all the suffixes of a given text. For a text of length n, the suffix tree contains exactly n leaves, each labeled from 1 to n. The tree satisfies the following properties:

- Every node, except for the root, has at least two children.
- No two edges originating from the same node begin with the same character.

To reconstruct all the suffixes of the text, one can traverse all the branches of the tree. A special character, \$ (typically referred to as a sentinel), is appended to the end of each suffix to mark its termination. This character is lexicographically smaller than all other characters in the alphabet, ensuring proper ordering and uniqueness in the tree structure [3].

2.2 FP-Growth Algorithm

The FP-Growth (Frequent Pattern Growth) algorithm, available in the Python library PySpark, is a widely used method for identifying frequent itemsets in transactional data. The algorithm operates in the following steps:

(1) It first counts the frequency of each item in the dataset and identifies those that meet a user-defined frequency threshold, classifying them as frequent items.
(2) It then encodes the transactions using a compact suffix-tree structure, known as an FP-Tree, which avoids the explicit generation of candidate itemsets.
(3) Finally, it extracts the frequent itemsets directly from the FP-Tree structure.

In addition to the standard FP-Growth algorithm, PySpark's `spark.mllib` library provides a parallelized version called PFP (Parallel FP-Growth), which is optimized for distributed computing environments. For a more in-depth discussion of the algorithm and its parallel implementation, the authors recommend the work by Li et al. [5].

2.3 Davies-Bouldin Index

The Davies-Bouldin Index (DBI) is a metric used to evaluate the validity of a clustering process. Specifically, it measures the trade-off between cohesion within clusters and separation between clusters. A lower DBI value indicates better clustering, with compact and well-separated clusters, while a higher DBI suggests potential overlap between clusters. The calculation of the DBI involves the following steps:

- **Calculation of centroids:** Determine the centroid of each cluster.
- **Calculation of intra-cluster dispersion:** For each cluster, compute the average distance between every point and its centroid.
- **Calculation of inter-cluster dispersion:** For each pair of clusters, compute the distance between their centroids.

The Davies-Bouldin Index is then calculated using the formula:

$$DBI = \frac{1}{k}\sum_{i=1}^{k}\max_{i\neq j}\left(\frac{wcd_i + wcd_j}{dbc_{i,j}}\right),$$

where:

- k is the total number of clusters,
- wcd_i is the within-cluster dispersion for cluster i,
- $dbc_{i,j}$ is the distance between the centroids of clusters i and j.

For further details, the authors recommend consulting [2].

2.4 Calinski-Harabasz Score

The Calinski-Harabasz Score, also known as the Variance Ratio Criterion (VRC), is a metric used to evaluate the performance of clustering algorithms. It is defined as the ratio of the sum of between-cluster dispersion to the sum of within-cluster dispersion. A higher score indicates better-defined and well-separated clusters. Mathematically, the score is calculated as follows:

$$CH = \frac{\text{Between-Cluster Dispersion}/(k-1)}{\text{Within-Cluster Dispersion}/(n-k)},$$

where:

- k is the number of clusters,
- n is the total number of data points.

The between-cluster dispersion measures the separation of clusters, while the within-cluster dispersion quantifies the compactness of individual clusters. A higher Calinski-Harabasz Score reflects a clustering solution with dense and well-separated clusters. For further details, refer to [4].

2.5 Silhouette Coefficient

The Silhouette Coefficient (SC) is a metric used to evaluate the quality of clustering. It measures how similar a sample is to its own cluster compared to other clusters[1]. For a given sample, the Silhouette Coefficient is calculated as:

$$SC = \frac{b - a}{\max(a, b)},$$

where:

- a is the mean distance between the sample and all other points in the same cluster,
- b is the mean distance between the sample and all other points in the next nearest cluster.

The Silhouette Coefficient for a set of samples is the average of the SC values for all individual samples. The score ranges between -1 and 1, where:

- A value close to 1 indicates that the sample is well-matched to its own cluster and poorly matched to neighboring clusters,
- A value close to 0 suggests overlapping clusters,
- A value close to -1 implies that the sample may have been assigned to the wrong cluster.

2.6 PCY Algorithm

The PCY (Park-Chen-Yu) algorithm is an efficient method for identifying frequent itemsets, designed as an optimized variant of the Apriori algorithm. It operates in two main passes:

(1) **First Pass:**
   - Count the frequency of each individual item in the dataset.
   - Simultaneously, maintain a hash table of fixed size to store the counts of item pairs. Each pair of items is hashed into a bucket, and the count for that bucket is incremented.
   - Given a support threshold s, any bucket with a count below s is discarded, as it cannot contain frequent item pairs.

(2) **Second Pass:**
   - Examine the remaining buckets that may contain frequent item pairs.
   - For each candidate pair in these buckets, verify its frequency in the dataset.
   - If the frequency of a pair exceeds the support threshold s, it is classified as frequent.

The PCY algorithm reduces memory usage during the second pass by leveraging the hash table to eliminate infrequent pairs early, making it more efficient than the Apriori algorithm. This optimization is particularly beneficial for large datasets with high dimensionality. [6]

3 IMPLEMENTATION

3.1 Data

For privacy reasons, no real data was provided to the authors. Therefore, a careful analysis of the type of data being processed was conducted to construct an optimal data generation algorithm that could reflect that of a real-world company.
The data pertains to a logistics company, specifically focusing on trucks tasked with transporting goods.
Below is a brief analysis of the data structure, which serves as the foundation for constructing the datasets:

- Two datasets are processed, both in `.json` file format.
- The first dataset, `standard.json`, contains the **standard routes**, which are predefined paths computed by software tailored to the logistics company's needs.
- The second dataset, `actual.json`, contains the **actual routes**, representing the paths drivers have effectively taken during their work. These routes often deviate from the standard routes.
- The `standard.json` dataset has two fields:
  - *id*: A unique identifier for the standard route, formatted as "s" followed by a number (e.g., "s1", "s2").
  - *route*: This field contains all relevant information about the path, structured as an ordered sequence of trips.
- The `actual.json` dataset has four fields:
  - *id*: A unique identifier for the actual route, formatted as "a" followed by a number (e.g., "a1", "a2").
  - *driver*: A unique identifier code for the driver who operated the actual route.
  - *sroute*: The identifier code of the standard route assigned to the driver when they completed the actual route.
  - *route*: This field is identical in structure to the *route* field in `standard.json`.
- Routes are ordered compositions of trips, where each trip (except the first) begins in the city where the previous trip ended. A single trip comprises three pieces of information:
  - *from*: The departure city.
  - *to*: The destination city.
  - *merchandise*: The goods transported during the trip, including a list of items and their respective quantities.
- Two strong assumptions underpin the construction of the datasets and the design of the algorithms:
  - Physical distances between cities are neglected; geographical considerations are not factored into the analysis.
  - In each city, trucks completely unload their merchandise and load new items for the next destination. Any type and quantity of merchandise can be loaded at each city.

With this information, it is possible to proceed with the creation of the two datasets.

Construction of `standard.json`

The construction of the `standard.json` dataset began with the use of a file named `provinces.csv`, which contains the names of all 110 Italian provinces. This file provided a pool of cities to serve as departure and destination points for the trips. The code is designed

to be generalizable, allowing any file containing destination strings to be used instead.
Additionally, a pool of merchandise was required for dataset generation. To achieve this, random strings consisting of 3 to 9 random letters were generated.

The dataset construction process was modularized into small functions, each serving a specific purpose. Below is a description of the key functions used:

- *province_reader*: Reads the `.csv` file and returns a list of provinces.
- *province_cutter*: Takes the list of provinces and an integer (less than or equal to the number of provinces) as input, returning a randomly selected subset of provinces.
- *merchandise_generator*: Generates a list of random strings, with the number of strings specified by the input parameter `tot_merch`.
- *randomizer*: A utility function used to randomize data at various stages of the process.
- *trip_generator*: Constructs a single trip as a dictionary. It takes as input a set of provinces (either the full set or a subset generated by *province_cutter*), a merchandise set, the starting city, and the number of items per trip. The function assigns:
  - The key `"from"` to the departure city.
  - The key `"to"` to a randomly selected destination city from the province set.
  - The key `"merchandise"` to a dictionary of randomly selected merchandise items and their quantities.
- *single_sr_generator*: Generates a standard route. It takes as input the province set, the merchandise set, the average number of items per trip, and the number of trips. The function:
  - Randomly selects a starting city.
  - Iterates the *trip_generator* function to create a sequence of trips, where each trip's starting city is the destination of the previous trip.
  - Returns a list of trips representing the standard route.
- *standard_routes_generator*: Constructs the final `standard.json` dataset. It takes as input the number of standard routes to generate, the reduced province set, the average number of items per trip, the average number of trips per route, and the merchandise pool. The function:
  - Creates a list of dictionaries, each representing a standard route.
  - Assigns the key `"id"` to a unique identifier of the form "s" + integer.
  - Assigns the key `"route"` to a standard route generated by *single_sr_generator*.
  - Returns the list of dictionaries as the final dataset.

Construction of `actual.json`

The creation of the `actual.json` file, while structurally similar to `standard.json`, followed a distinct process. Specifically, the actual routes were generated as randomized variations of the standard routes. Below is an explanation of the process, described through the key functions used:

- *drivers_generator*: This function takes an integer as input and returns a list of random strings of that length, where each string represents a driver's identifier. This function can be replaced with an actual list of driver names if available.
- *n_merchandise_randomizer*: This function takes a merchandise dictionary as input, adds an integer white noise, and randomizes the number of items while maintaining the same expected value as the original number of items.
- *t_merchandise_randomizer*: This function randomizes the items in the merchandise dictionary. It can add, remove, or modify items based on predefined probabilities.
- *single_ar_generator*: This function generates an actual route by altering a given standard route. It takes as input a standard route, the set of provinces, and the merchandise pool. With predefined probabilities, it modifies cities in each trip, adds or removes trips, and randomizes the merchandise using the aforementioned functions.
- *actual_routes_generator*: This function constructs the final `actual.json` dataset. It takes as input the set of standard routes, drivers, merchandise, provinces, and the number of routes each driver should perform. It assigns each driver a random number of standard routes (with an expected value equal to the input parameter), alters them using *single_ar_generator*, and organizes them into the required format. The function returns the set of actual routes.

Both the actual routes and the standard routes are then written to a `.json` file using a utility function called *json_writer*.

Route Evolution: Shaping Actual Routes from Standard Routes

The code that generates the data, as described earlier, has various input parameters. In particular, there are parameters that can be used to adjust the probabilities of altering the actual routes. The following is a brief analysis of how different probabilities impact the data, and what is the probability that a standard route is executed perfectly according to our code.

It is emphasized that the data should not be excessively realistic but rather generalizable. The goal of this data generation is to provide an extremely generic dataset devoid of patterns, allowing the construction of an algorithm that can work regardless of intrinsic trends in the data.

The list of all possible alterations with their respective probabilities is as follows:

- A single item of the merchandise is either removed (P_{Mrem}) or added (P_{Madd}).
- A single item of the merchandise is substituted with another one (P_{Msub}).
- The quantity of a single item is modified (P_{Mmod}). The quantity can vary positively or negatively by multiple values, but the overall probability of undergoing any quantity change is considered.
- The probability of changing the starting city of the route or an arrival city for each trip. The probability is the same and represents the likelihood of replacing a city (P_{Csub}).

- The probability of adding or removing a trip. The probabilities are equivalent to those of adding (P_{Cadd}) or removing (P_{Crem}) a city within the route.

Before moving on to formulating the probability of a standard route remaining unchanged, it is necessary to assign names to other parameters involved in the calculation. It is important to note that many parameters are randomly generated around a mean value. To calculate compound probabilities, leveraging a property of these probabilities, it is sufficient to use the expected value:

- n_{item}: The expected value of the number of items for each standard route.
- n_{quant}: The expected value of the quantity of each item for each standard route.
- n_{city}: The expected value of the number of city for each standard route. It is emphasized that the number of cities is equal to the number of trips + 1.

Following the logic of the code, various relevant probabilities are calculated. The first elements randomly extracted are the driver and the standard route they must execute. Since both are randomly generated and assumed to be equal in expected value, this random component is not explicitly considered in the probability calculations.
The first random modification involves changing a city within the route. The probability that no city is modified is given by:

$$P_1 = (1 - P_{Csub})^{n_{city}},$$

where P_{Csub} is the probability of substituting a city, and n_{city} is the number of cities in the route.
Next, the probability of adding or removing a trip (and thus a city) is considered. The probability of adding a trip depends on the number of possible insertion points, which includes the spaces between existing trips and the beginning or end of the route. The probability that no trip is added is:

$$P_2 = (1 - P_{Cadd})^{n_{city}+1},$$

where P_{Cadd} is the probability of adding a trip.
The probability of removing a trip depends on the number of trips, which has already been altered by the addition of new trips. Specifically, the expected number of trips added is:

$$E(X_{\text{add}}) = (n_{city} + 1) \times P_{Cadd},$$

where X_{add} is the random variable representing the number of trips added. The probability that no trip is removed is then:

$$P_3 = (1 - P_{Crem})^{n_{city}-1+(n_{city}+1)\times P_{Cadd}},$$

where P_{Crem} is the probability of removing a trip.
The expected number of trips removed is:

$$E(X_{\text{rem}}) = \left[n_{city} - 1 + (n_{city} + 1) \times P_{Cadd}\right] \times P_{Crem},$$

where X_{rem} is the random variable representing the number of trips removed. Therefore, the total expected number of cities after these modifications is:

$$n_{city\_new} = n_{city} + (n_{city}+1) \times P_{Cadd} - \left[n_{city} - 1 + (n_{city} + 1) \times P_{Cadd}\right] \times P_{Crem}.$$

The analysis now shifts to the probabilities of changes to the merchandise, assuming the number of trips is equal to the newly calculated expected number of cities minus one.

The first modification that the merchandise basket can undergo is the replacement of an item with another not already present. For each trip and for each item in each trip, this probability must be considered. The probability that no item is modified is:

$$P_4 = (1 - P_{Msub})^{n_{city\_new} \times n_{item}},$$

where P_{Msub} is the probability of substituting an item, n_{city_new} is the expected number of cities after modifications, and n_{item} is the number of items per trip.

Next, the probability of adding items is considered. The reasoning is similar to that of adding trips:

$$P_5 = (1 - P_{Madd})^{n_{city\_new} \times (n_{item}+1)},$$

where P_{Madd} is the probability of adding an item. The expected number of items added is:

$$E(X) = P_{Madd} \times \left[n_{city\_new} \times (n_{item} + 1)\right].$$

Similarly, the probability of reducing the number of items is:

$$P_6 = (1 - P_{Mrem})^{n_{city\_new} \times n_{item} + P_{Madd} \times \left[n_{city\_new} \times (n_{item}+1)\right]},$$

where P_{Mrem} is the probability of removing an item. The expected number of items removed is:

$$E(X) = \left\{n_{city\_new} \times n_{item} + P_{Madd} \times \left[n_{city\_new} \times (n_{item} + 1)\right]\right\} \times P_{Mrem}.$$

The total expected number of items in a route after these modifications is:

$$\begin{aligned} n_{tot\_item\_new} = {} & n_{item} \times n_{city\_new} \\ & + P_{Madd} \times \left[n_{city\_new} \times (n_{item} + 1)\right] \\ & - \left\{n_{city\_new} \times n_{item} + P_{Madd} \times \left[n_{city\_new} \times (n_{item} + 1)\right]\right\} \times P_{Mr} \end{aligned}$$

Finally, the probability of not modifying the quantities of each individual item is:

$$P_7 = (1 - P_{Mmod})^{n_{tot\_item\_new}},$$

where P_{Mmod} is the probability of modifying the quantity of an item.

The seven calculated probabilities represent the likelihood of leaving the standard route unchanged during each of the seven randomization phases. However, these probabilities are not independent, so the probability of obtaining an actual route identical to a standard route cannot be calculated as a simple product of the individual probabilities. Instead, the probability P of not changing a standard route is computed as:

$$\begin{aligned} P = {} & P_1 \times (P_2|P_1) \times (P_3|P_2 \land P_1) \times (P_4|P_3 \land P_2 \land P_1) \\ & \times (P_5|P_4 \land P_3 \land P_2 \land P_1) \times (P_6|P_5 \land P_4 \land P_3 \land P_2 \land P_1) \\ & \times (P_7|P_6 \land P_5 \land P_4 \land P_3 \land P_2 \land P_1). \end{aligned}$$

This formula assumes that n_{city} and n_{item} remain unchanged. Thus, the final formula simplifies to:

$$\begin{aligned} P = {} & (1 - P_{Csub})^{n_{city}} \times (1 - P_{Cadd})^{n_{city}+1} \times (1 - P_{Crem})^{n_{city}-1} \\ & \times (1 - P_{Msub})^{n_{city} \times n_{item}} \times (1 - P_{Madd})^{n_{city} \times (n_{item}+1)} \\ & \times (1 - P_{Mrem})^{n_{city} \times n_{item}} \times (1 - P_{Mmod})^{n_{item} \times n_{city}}. \end{aligned}$$

Given this probability, it is possible to adjust the various error probabilities. Even if these probabilities are very small, most standard routes will still undergo modifications when processed as input to become actual routes.

Parameters tracking

With the purpose of simplifying the generation of the data for every different run, an approach for parameters tracking was implemented. In the project folder, a file *.env* was added. The file contains the following attributes:

- the run id (*RUN_ID*)
- the total number of standard routes (*STANDARD_ROUTES_COUNT*)
- number of drivers (*DRIVERS_COUNT*)
- mean number of trips in every route (*TRIPS_PER_ROUTE*)
- numbers of province in the subset (*PROVINCES_TO_PICK*)
- number of items in the starting set (*TOTAL_NUMBER_OF_ITEMS*)
- mean of number of type of merchandise in each trip (*NUMBER_OF_ITEMS_PER_TRIP*)
- mean number of routes assigned to every driver (*ROUTES_PER_DRIVER*)

Using the library *dotenv* it is possible to obtain the environment variables (declared in a *.env* file) during a run. Once the run is started and all the parameters are extracted, the method *save_run_parameters* builds a JSON object, with the environment variables names as keys and each respective value as their value. Then this object is saved in a file named *run_params.json* in the project folder's *src/data/*. This parameters are useful only for the generation of the data (apart from the *run_id*, which is used to pick standard< *run_id* >.json and actual< *run_id* >.json files and generate the relative outputs), while the output generation steps do not need this values.
This approach is important in order to be able to replicate results and keep track of different runs.

Routes in Object-Oriented Programming

Given that Python is the programming language used, object-oriented programming (OOP) is fully leveraged for processing routes. While lists and dictionaries were employed during the data creation phase, specific classes are preferred for handling data imported from JSON files. Each object initially represented as a dictionary during the creation process is now encapsulated within its own class.

Specifically, the following classes have been implemented:

- `Merchandise`: Represents the merchandise transported during a trip, with attributes corresponding to the items and their quantities.
- `Trip`: Encapsulates the details of a single trip, including the departure city, destination city, and the associated merchandise.
- `StandardRoute`: Represents a standard route, consisting of an ordered sequence of trips and a unique identifier.
- `ActualRoute`: A subclass of `StandardRoute`, representing an actual route taken by a driver. It includes additional attributes such as the driver's identifier and the assigned standard route.

Each key in the original dictionary has been transformed into an attribute of the corresponding class, with the dictionary's content assigned as its value. This approach enhances code modularity, readability, and reusability, while also enabling the implementation of methods specific to each class's functionality.

3.2 Recommended Standard Routes

This section focuses on the creation of the first output: a set of recommended standard routes to replace the existing ones.
Unlike the current set, which was generated by software tailored to the company's needs, the new set is designed to reflect the drivers' implicit agendas and preferences. It is assumed that deviations from the standard routes or alterations in transported goods by drivers are indicative of their personal preferences or requirements.

Since drivers' preferences can only be inferred from the routes they have actually traveled, and it is impossible to determine whether a route was taken due to error or personal necessity, the dataset used for generating this output consists of actual routes.
The algorithm is based on the hypothesis that the most significant indicator of a driver's preference lies not in individual deviations from standard routes but in the repetition of specific routes, city stops, and transported goods over time. Therefore, the actual routes dataset is the sole decisive input for generating the recommended standard routes.

Theoretical Concept

The core concept of the algorithm is to aggregate actual routes into clusters within a suitable space and derive the recommended standard routes as the centroids of these clusters. This approach ensures that each actual route is "represented" within a cluster, while the centroid serves as a balancing point that mediates among all routes in the cluster. Furthermore, the number of clusters can be constrained to match the number of standard routes, ensuring coherence in the recommended set. However, this does not imply that each recommended standard route must represent an equal number of actual routes.
To implement this technique, the first step is to construct a suitable space for the analysis.

Space Definition

The process begins with the creation of a vector space. Let N be the number of distinct cities appearing in the actual routes, M the number of distinct types of merchandise, and T the number of unique trip combinations found in the actual routes (e.g., a trip from "Trento" to "Verona" is recorded as "Trento:Verona").
The space will have $N + M + T$ dimensions, one for each city, type of merchandise, and trip. Each actual route is represented as a point in this space.

The coordinates of these points are determined as follows:

- For cities: A dimension corresponding to a city takes the value 5 if the city appears in the actual route, and 0 otherwise.
- For trips: A dimension corresponding to a trip takes the value 10 if the trip is present in the route, and 0 otherwise.
- For merchandise: The value assigned to a merchandise dimension is significantly lower to ensure it influences the distance calculation without being overly decisive. Specifically, the value is the quantity of that merchandise in the

route divided by the total quantity of all merchandise in the route.

Through this process, each actual route is represented as an array of float values, where each value corresponds to a coordinate in the space. The array of arrays representing all actual routes is saved in a file named `coordinates<run_id>.csv`. The first row of the file contains the list of dimensions.

Clustering with PySpark

The k-means algorithm from PySpark is employed for clustering. The input is a table of actual routes, where each row represents an actual route and each column corresponds to a dimension in the space. Each cell in the table contains the value of the actual route for that dimension. This data is extracted from the `coordinates<run_id>.csv` file created earlier.

The file is read using PySpark's `read` method, which converts the file content directly into a DataFrame, the data type used for clustering. By providing the additional options `header` and `inferSchema`, PySpark reads the first row of the file as the table header and automatically infers the data types of the columns.

Let K be the number of standard routes. This value is used as the second parameter for the k-means algorithm, ensuring that the number of resulting clusters matches the number of standard routes. The program then produces K recommended routes. After clustering, the program outputs the values of the centroids, which represent the recommended standard routes.

Building Routes from Cluster Centroids

The next step involves reconstructing the recommended standard routes from the cluster centroids. For each centroid, dimensions with a value of 0 are disregarded, as they do not contribute to the recommended route. This applies to cities, merchandise, and trips. To extract the relevant cities from the clustering, the following steps are performed for each cluster center:

- The values of the city dimensions in the cluster center are sorted in descending order, excluding values equal to zero.
- The number of cities to include in the recommended standard route is determined by taking the average number of trips per route and adding one (since the first trip's "from" city is not preceded by another trip). The threshold index is set as the minimum between this value and the length of the sorted array. The value at this index (-1) serves as the threshold.
- Every city dimension with a value greater than or equal to the threshold is included in the recommended standard route.

A similar process is applied to identify the relevant trips for the cluster center, though the average number of trips per route is not incremented in this case. For merchandise, all dimensions with a value greater than 0 are considered for the entire route.

To determine the merchandise carried in each trip, a user-defined function is employed. This function takes as input a set of actual routes and their corresponding points in the space. It returns, for each destination city in every trip, the most frequently carried merchandise and its average quantity. By inputting all actual routes of a single cluster along with their space representation, the function provides the necessary merchandise information for each destination city.

The decision to organize the output by destination city rather than by trip is based on two considerations:

- From a practical perspective, transport companies can load any type of resource in any city, making the delivery city the primary focus.
- Within a single cluster, the likelihood of having two trips in the recommended standard route with the same destination city is very low. Thus, computing merchandise by trips instead of cities would unnecessarily complicate the program for a rare scenario.

In summary, the actual routes are synthesized into clusters, with centroids translated into recommended standard routes.
The expected outcomes vary with input data: if drivers closely follow the standard routes, the recommended routes will differ minimally, reflecting limited influence from drivers' agendas or preferences. Conversely, significant deviations in actual routes will result in noticeable changes to the recommended standard routes.

3.3 Five Preferred routes for each driver

With the aim of finding the five preferred standard routes for each diver, as the five routes that they are more likely to follow, the option of comparing every actual routes with every standard route, and define their similarity, has been considered.
However, since it is possible that the customer company possesses vast amount of data, this process can quickly become computationally expensive. In fact, supposing the company's data consists of 100 drivers with an average of 300 travelled actual routes and 200 total standard routes, 100 matrix with dimensions $200x300$ would be needed to compare the data and produce the five preferred routes. The complexity raises even more if the amount of data increases. In this scenario, a need to reduce the dimension of the data to process emerges.
To overcome this problem, the authors provide a different approach to the solution, aiming to reduce the amount of comparisons to be made with large set of data without losing big chunks of important information.
To summarize the actual routes travelled by each driver, a new object *Preferences* is designed and computed for each driver. The idea is to analyze the actual routes travelled by each driver and only extract the important information. The object will then be compared with the standard routes to compute the output. This reduces the number of comparisons as instead of comparing n actual routes, only the object Preference will be compared.

This structure of the object is defined in the *preferences.py* class in the *entities* package. A list of *ActualRoutes* object is passed as a parameter to the function *get_actual_routes_per_driver*, which returns a dictionary with keys the drivers names', and values the actual routes travelled by the corresponding driver (saved as a list of *ActualRoute* objects).

Extracting preferences for each driver

In the file presented to the professor, a *for* loop is used to iterate through the drivers and pass the list of Actual Routes as argument to the preferences constructor together with a threshold float value and an integer number of buckets (used in the PCY algorithm).
In the file *second_main.py*, which contains the solution presented to an hypothetical company, a further option is included, where the user is asked to choose between computing the preferences for every driver in the database, or just for a single driver. If the latter option is selected, a further question is prompted where the user is asked to insert the name of the driver he/she wants to compute the preferences for.
This approach was selected because figuring out preferences can be time-consuming. Sometimes, a company may only need preferences for a single driver, and it would not make sense to calculate preferences for everyone when just one is needed.

In this implementation, the class Counter of the library collections was used. When calling *collections.Counter* on a list, it returns an object *Counter*, a dictionary-like object with keys the distinct instances of the list and values the number of times the instance appears in the list.
The resulting *Preferences* object summarizes all the actual routes travelling by a driver with the following attributes:

(1) Most frequently crossed cities ($freq\_city$)
The Counter of all the starting cities (the *from* field of the first trip) and the Counter of every city in the *to* field are summed (the Counter object allows addition between instances, returning the sum over each key). It measures the total number of visits a driver has made in a city.
Only n_trip * 2 most common cities are stored into the *Preferences* object output.

(2) Most frequent starting cities of a route ($freq\_start$)
the Counter of all the starting cities. It measures how many times a city is at the start of a route.
Only n_trip + 1 most common start are stored into the *Preferences* object output.

(3) Most frequently ending cities of a route ($freq\_finish$)
The Counter of all the ending cities (the *to* of the last trip). It measures how many times a city is at the end of a route.
Only n_trip + 1 most common end are stored into the *Preferences* object output.

(4) Most frequent trips ($freq\_trip$)
The Counter of all the trips combinations (the combinations are stored as a list of tuples). This is used to measure how many times each trip was done by a driver.
Only n_trip most common trips are stored into the *Preferences* object output.

(5) Mean number of trips for every route (n_trip)
The empirical mean of the number of trips per route. It measures how many trips a driver does in the routes assigned.

(6) Mean number of type of merchandise for every trip ($type\_merch\_avg$)
The empirical mean of different types of merchandise in all the trips. It measures how many different item of merchandise in average a driver brings.

(7) Mean quantity of every merchandise in trip ($n\_merch\_per\_route$)
It measures the average of the quantities that a driver brings in a route.

(8) Frequent itemset of cities ($freq\_itemset\_city$)
For every actual route in the list, a list of cities is extracted to compute a frequent itemset with the PCY algorithm. This is done to understand the most common cities that repeat in a route.
Only n_trip most common frequent itemset are stored into the *Preferences* object output.

(9) Frequent itemset of trips ($freq\_itemset\_trip$)
For every actual route in the list, a list of trips (tuple containing *from* and *to*) is extracted to compute a frequent itemset with the PCY algorithm. This is done to understand the most common couple of trips done in the same route.
Only n_trip most common frequent itemset are stored into the *Preferences* object output.

(10) Most frequently type of merchandise brought (n_merch)
The Counter of all the different types of merchandise present in the list of actual routes. It measures how many times a driver brings a merchandise in all the trips.
Only ($type\_merch\_avg$) * 2 most common merchandise are stored in the *Preferences* object output

(11) Most frequent type of merchandise per city $n\_merch\_per\_city$
For every city the most frequent type of merchandise is counted. A set of cities is formed taking the most frequent destination cities (*to* field), the most frequent cities present in the frequent itemset of trips and the most frequent cities in the frequent trips. For every city of this set, every type of merchandise brought in the actual routes is counted.
This variable will be used to build the results of the next point.

The choice to only store in memory the most common values for each variable was made because the object *Preferences* was constructed with the intent of lightening the comparison tasks. It was therefore considered that instances with a low count were not representative of the actual preference of a driver, and were therefore striked from the output. This approach may lead to loss of precision, as some information are inevitably lost, but it is considered to be a fair trade-off in exchange of performance.
The particular choices of size to save in the output were made considering that the *Preferences* would still be a good description of the drivers' behavior.

PCY algorithm

As mentioned above, the PCY algorithm was used to compute the frequent itemset for (8), (9) and (10). The search stopped at pairs of frequent items, without going on to search for triplets or more. This decision was made to keep the computational cost low. In addition, searching for pairs proved to be sufficient to obtain relevant results.

The threshold is set at 0.2.
The authors have found that this support allows an output in most cases. Otherwise, the algorithm is able to halve the threshold until a non-zero result is obtained.

Using preferences to calculate similarities

The next step is to compute the similarities between every standard route and the preferences object of a driver. In order to do that, for every driver preferences the method $preferoute\_similarity$ is called for every standard route, returning a similarity score. This score is stored in a dictionary containing all the standard routes scores for a single driver.
The *preferoute_similarity* function takes as input:

- a standard route
- the driver preferences
- an array representing the weights (if not set, all the weights are equal to 1)

With this inputs, it builds an array of scores with the following criteria (every position of the array has value 0 at the beginning):

(1) for every $city$ in the standard route, if it is present in the keys of $freq\_city$, sum at the first position of the array $freq\_city[city]$ divided by the sum over every key of $freq\_city$
(2) if the starting city of a route is in the keys of $freq\_start$, insert 1 in the second position of the array
(3) if the ending city of a route is in the keys of $freq\_finish$, insert 1 in the third position of the array
(4) for every $trip$ in the standard route, if it is present in the keys of $freq\_trip$, sum at the fourth position of the array $freq\_trip[trip]$ divided by the sum over every key of $freq\_trip$
(5) for every $city\_combo$ (combination of two cities) in the standard route, if it is present in the keys of $freq\_itemset\_city$, sum at the fifth position of the array $freq\_itemset\_city$ $[city\_combo]$ divided by the sum over every key of $freq\_itemset\_city$
(6) for every $trip\_combo$ (combination of two trips) in the standard route, if it is present in the keys of $freq\_itemset\_trip$, sum at the sixth position of the array $freq\_itemset\_trip$ $[trip\_combo]$ divided by the sum over every key of $freq\_itemset\_trip$
(7) in the seventh position, insert 1 minus two times the absolute value of the difference between the length of a route (number of trips) and n_trip, divided by the sum of them. If the result is negative, insert 0
(8) in the eighth position, insert 1 minus two times the absolute value of the difference between the average number of merchandise over the trips in a route and $type\_merch\_avg$, divided by the sum of them. If the result is negative, insert 0
(9) for every $merch$ in the standard route, if it is present in the keys of n_merch, sum at the ninth position of the array $n\_merch[merch]$ divided by all the values in n_merch
(10) in the tenth position, insert 1 minus the absolute value of the difference between the average of the total quantities of merchandise for every trip and $n\_merch\_per\_route$, divided by the sum of them. If the value is negative, insert 0

After this processing, the array of weights has to be standardized. Each element is divided by the sum of all the weights and assigned to a standardized array.
Finally, every element of the array of scores is multiplied by the respective standardized weight and the results are added to the return variable of the function, which is the actual similarity found between the preferences and the standard route. The value of this variable is always between 0 and 1.
Once all the similarities of a driver are calculated, the similarities are sorted in descending order and the five with the highest score are selected for the output.

3.4 An optimal standard route for each driver

In this subsection, the authors suggest a new standard route for each driver, tailored on the preferences extrapolated from the driver's actual routes data.

Specifically, an object *Preference* is computed in the same way as described above. The file *main.py* only computes the preferences once and uses it for both the second and third point, but since the files presented to the hypothetical customer company need to be able to run on their own, the object is computed in both the files separately.
In computing the ideal route, several assumptions have been made:

- A driver prioritizes trips over simple destinations.
- The ideal route is designed starting from the path, with merchandise assigned based on the destination.
- The needs of the company are considered secondary to the preferences of the driver.
- A route can pass through a city only once.
- For clarity, the term "trip" refers to a path between a starting and ending city, ignoring the merchandise.

Having computed the object *Preferences*, the authors proceeded to create the algorithm for the ideal route.

The length of the optimal standard route for each driver is set as the rounded mean number of trips computed from the corresponding actual routes, as it is assumed that it is his/her preferred length. The following steps are taken in order, until the optimal length of the route is reached.
Firstly, a series of checks is made on the length of the route: if the average length is 0 or less, an exception is raised. If the length is 1 the most frequent trip is returned as a result. If the length is greater than one a route is computed with the following criteria:

- *Frequent itemset of trips*
  If there are trips inside the frequent itemset that are chainable (e.g. *(CityA, CityB), (CityB, CityC)* or *(CityB, CityA), (CityC, CityB)*), we consider those as priority.
  If none are found, a chaining trip is inserted if the desired length of the output is greater than 2 (e.g. if the frequent itemset is *(CityA, CityD), (CityB, CityC)*, the trip *(CityD, CityB)* is inserted in the middle.
  If the desidered length is equal to 2 and no chainable trip is found, the output will be the most frequent trip concatenated with a compatible frequent trip.
  If no compatible frequent trip is found, the most frequent

start or finish (if the frequent start coincides with the start of the most frequent trip) is added to complete the route. Then, the longest chainable sequence obtainable (as long as the length of the output is lesser than the desired length of the ideal route) from the frequent itemset of trips is returned as output.
The priority is always given to the frequent itemset that is most frequent. If enough data is present to form a set of trips that covers the desired length, the path is returned as an ideal route, and the merchandise is assigned as described below.

- *Frequent trips*
  If the frequent itemset of trips was not enough to compute an ideal route, the first tiebreaker are the most frequent trips.
  If a frequent trip that is not already inside the route can be chained to the chain of trips that was inherited from the previous point, it is added. If the set of frequent trips is not enough to reach the required length, the next tiebreaker is used.
- *Most frequently starting cities of a route*
  If the first city is already in the route, but is not the start of the first trip, it is ignored and the next most frequent starting city is analysed.
  If the city coincides with the start of the chain of trips, it is flagged as the start of the route, and trips can from now on only be chained to the end.
  If the frequent start is not in the chain of trips it is added as the first city and a placeholder city is assigned as the end of the first trip.
  The process is repeated until a result is found. It will surely be found because the length of the frequent start list is greater than the output length.
- *Most frequently ending cities of a route*
  If a city that was already part of the chain was assigned as start, no city from the chain can be assigned as finish, as it would break the chain.
  If it was not, we look for either a new city or the last city of the chain.
  If a new city is added at the end of the list of trips, a placeholder is set as a start of the last trip. If the last city of the chain was assigned, it surely means that the placeholder city is after the starting city.
  If at the end there is a placeholder both at the start and at the end, and the required length is not reached, it is chosen to replace the placeholder at the end of the first trip with the start of the second trip. This can be done because no other information on city positioning inside the route is available from now on.
  If the required length is reached at this point, the placeholder is substituted with the last city of the previous trip (if the placeholder is at the end) or the first city of the next one (if at the start).
- *Frequent itemset of cities*
  If in the frequent itemset of cities there are couples of cities that contain only one city that is already inside the chain of trips, that city is inserted in place of the placeholder, and the placeholder shifts towards the center of the chain. The process is repeated until the frequent itemset of cities is read.
  If the required length is still not reached, the final tiebreaker is used.
- *Frequent cities*
  The most frequent cities that are not already inside the route are added in lieu of the placeholder, which shifts towards the center until the required length is reached.
  This will surely produce an output because there are enough cities in frequent cities (more than n + 1) to complete the path.
  In the end, the placeholder is substituted with the last city of the previous trip or the first city of the next trip according to what is needed to complete the route.
- *Merchandise items*
  After assigning the trips, the merchandise is loaded accordingly to the driver's preferences as described in the first solution (Building Recommended Standard Route from Centroids).
  If the destination city is inside *Preferences* variable *n_merch_per_city*, the items described in that variable are assigned to the trip. The variable contains in fact the items that the driver delivered most frequently to the specific city.
  Due to the restricted size of the variable there is a small possibility that not every destination city is described by it. Should that happen, a sample of the driver's most favourite items to deliver in general is assigned to the trip.
  The number of items consists of the average number of items the driver has brought in his trips, since we consider this to be his ideal number.
- *Merchandise quantity*
  The total quantity per trip is divided by the number of items to obtain a mean, around which a random function assigns a quantity.

The resulting output is the ideal route computed from the preferences of a driver that were extracted from his actual route.

4 EXPERIMENTAL EVALUATION

4.1 First Output: Recommended Standard Route

To assess the initial output, two distinct evaluations are employed. Given that a clustering was generated initially, and subsequently, centroids were transformed into routes, it is imperative to provide a concise evaluation of the effectiveness and quality of the clustering on the data. Following that, an assessment of the actual value of the recommended routes in the output is required. For the evaluation, the initial step involved partitioning the dataset into two distinct sets: the training set and the test set. The underlying concept for the evaluation is to construct clusters using only the data from the training set. Subsequently, the obtained results are evaluated in comparison to the data from the test set.

Clustering Evaluation

To ascertain the success of the clusters, specific descriptive indices were referenced: the silhouette coefficient, the Davies-Bouldin index, and the Calinski-Harabasz score. The concept is to, once the clusters have been constructed in the vector space using the routes from the training set as points, translate the actual routes from the test set into the same space and assign each to a respective cluster. Therefore, the following indices, as previously described in a preceding section, were computed:

- **Silhouette Coefficient**: 0.1938
- **Davies-Bouldin Index**: 0.0
- **Calinski-Harasbaz Score**: 2.4738

The indices provide a descriptive assessment of the clusters that does not yield extremely positive results but, at the same time, suggests a successful clustering.

Recommended Standard Route Evaluation

To evaluate the recommended standard routes, we rely on the vector space generated initially. The recommended standard routes are the projections into space of the centroids obtained from clustering, but subject to approximations. Therefore, to assess the recommended standard routes generated with the training set, and not the centroids, it is necessary to convert the recommended standard routes back into points in space.

Once the recommended standard routes have been described in the space, it is possible to calculate the distance between these and the actual routes in the test set that reside within their respective clusters. The other distance that is calculated is between the actual routes of the test set and the nearest original standard route. In this way, we obtain two distances that are comparable and can provide an index of improvement from the recommended standard routes.

It is noteworthy that the distance utilized is the Euclidean distance in the vector space generated earlier, and it is assumed that even the standard routes, once transposed into space, serve as the centers of the clusters. However, it is not enforced that the clusters remain the same, nor is it necessary for the actual routes to fall necessarily into the cluster of the standard route they are traversing. In this manner, consideration is given to the scenario where a driver has deviated from their standard route to the extent that they are traveling on an actual route more similar to another standard route.

To obtain a comprehensive index, therefore, ratios are computed between the distances of an individual actual route to the original standard route and to the recommended standard route. All these ratios pass through a filter that removes from the list those cases where an actual route perfectly aligns with either a recommended standard route or a standard route. In the end, the information enabling the evaluation of the recommended standard routes comprises three aspects:

- Geometric mean of ratios (without route with no deviation): 1.215
- Number of actual routes equal to a standard route: 0
- Number of actual routes equal to a recommended standard route: 0

The obtained indices signal an improvement compared to the original standard routes. Furthermore, the noticeable dispersion of the data is observed, with not a single standard route being traversed perfectly. This has led to a more distinct improvement in computing new standard routes. Naturally, with more realistic data, one would expect standard routes to undergo less pronounced corrections.

4.2 Second Output: Five Preferred routes for each driver

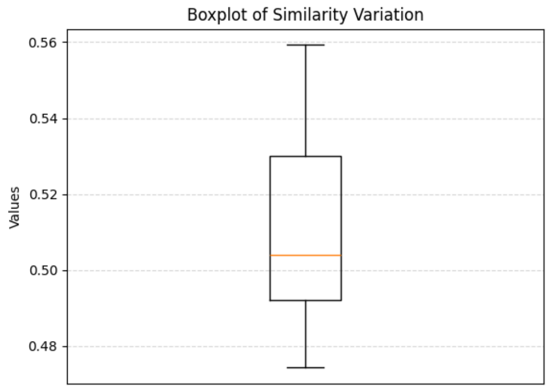
To test the second output, two different approaches were considered: the first one involves using the traditional method of the train-set/test-set, where the dataset was randomly split into two parts. The results obtained using only the train set were then projected onto the test set to assess its robustness. The second approach aims to test the dataset directly in its entirety and evaluate how closely the 5 recommended standard routes align with preferences compared to the other standard routes.

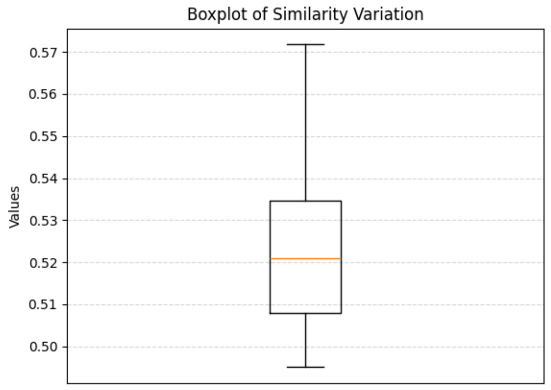
Training - Test

For the first test, a set of actual routes with a size of 80% of the total set was selected. Preferences of the drivers were constructed based on this set, from which the 5 standard routes with the highest similarity were then calculated. The similarities of the 5 routes per driver with respect to their preferences are then saved. At this point, preferences for the entire dataset of actual routes are generated, with the assumption that these preferences have been "updated" over time. The aim is to verify whether the new preferences continue to be compatible with the standard routes generated using the previous preferences. At this stage, the similarity between the updated preferences and the standard routes is calculated, and then compared with the previous similarity.

- Mean of distance between preferences from all actual routes and standard routes selected by train set: 0.5138
- Mean of distance between preferences from all actual routes and standard routes selected by all actual routes: 0.5260

As observed, the results are very promising due to their high similarity. In fact, the difference between the two averages can be considered almost negligible.





Relevance of Best 5 Standard Routes

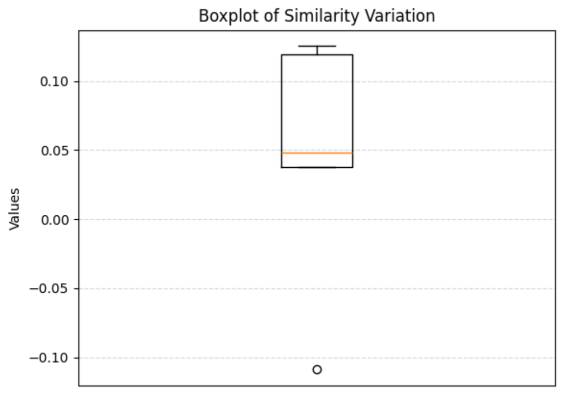
Regarding the second test, the significance of the 5 recommended standard routes compared to all other standard routes was tested. To do this, the average distance of the top 5 standard routes from the preferences is compared with the average distance from all the remaining standard routes and the preferences. Two different indices are obtained: the first one is a ratio, indicating how much the similarity has increased proportionally (or in percentage terms) by considering the top 5 standard routes; the second one is a difference, indicating the absolute increase in value.

- Ratio between mean of 5 best standard routes and the others: 7.9372
- Difference between mean of 5 best standard routes and the others: 0.1948

These results are extremely promising. The first indicates that by selecting the standard routes judged as the best by the algorithm rather than any other standard routes, there is a 720% increase in similarity between standard routes and preferences. The second shows the absolute increase, which is very high for values that range within 0-1.

4.3 Third Output: An optimal standard route for each driver

This point is also tested through the training/test process: the dataset is split with the usual proportion, and the results are compared as done previously. In particular, an ideal route is constructed for each driver using only the data of the actual routes that belong to the train set. Once generated, the driver preferences are updated with the routes from the test set, resulting in preferences built on the entire set of actual routes. At this point, the new preferences are compared with the ideal routes to see if there is a significant change in the distance between the ideal route and the preferences.



The average of the mean variations is 0.0572. It is a very positive result as the variation is low, indicating that the preference update does not have an excessively significant impact on determining the ideal route.

5 CONCLUSION

In conclusion, this in-depth examination of the three proposed solutions has highlighted the diverse approaches available while providing a clear understanding of their respective strengths and considerations.

The testing conducted in the fourth section of our study demonstrates the ability of the first solution to generate meticulously ordered and non-overlapping clusters. These clusters serve as the foundation for efficiently deriving centroids, which in turn contribute to the creation of a refined set of recommended standard routes.
The analysis reveals that these recommended routes outperform the original ones, indicating a tangible and quantifiable improvement. This underscores not only the effectiveness of the first solution but also its potential to enhance route optimization. The ability to derive superior routes from the identified clusters reinforces the viability of this approach as a valuable tool for optimizing route planning processes, promising significant efficiency gains.

The evaluations carried out for the second method demonstrate its effectiveness in successfully identifying five standard routes for each driver. These selected routes exhibit a significant increase in similarity when compared to the actual routes driven by the drivers. This result highlights the method's ability to optimize route selection, aligning it more closely with real-world driving patterns. The findings suggest that the second method holds promise as a reliable and effective tool for improving route planning and navigation based on individual driver preferences and historical travel behavior.

The results of the tests conducted for the third method underscore its effectiveness in creating a standard route tailored to each driver. This method has proven its ability to identify and recommend routes that exhibit greater similarity to those actually traveled. The increased similarity index not only validates the method's effectiveness but also highlights its potential to make a significant contribution to route optimization. By consistently producing standard routes that closely align with real-world travel patterns, this method provides drivers with routes that better reflect their preferences and historical behavior.

It is important to note that while the code is designed to function even with minimal data, the quality of the output improves with the amount of data provided. This is an inherent characteristic of the problem: as more data is gathered, the preferences of the drivers and the underlying patterns in the data become clearer, leading to more well-defined and accurate outputs.

REFERENCES

[1] Imad Dabbura. 2018. K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks. https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a [Accessed: (10/01/2024)].

[2] David L. Davies and Donald W. Bouldin. 1979. A Cluster Separation Measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-1, 2 (1979), 224–227. https://doi.org/10.1109/TPAMI.1979.4766909

[3] Sushant Gaurav. 2021. What is a Suffix Tree? https://sushantgaurav57.medium.com/what-is-a-suffix-tree-91c6b4951f3b [Accessed: (10/01/2024)].

[4] T. Caliński & J Harabasz. 1974. A dendrite method for cluster analysis. P3 (1974), 1–27. https://doi.org/10.1080/03610927408827101

[5] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. 2008. Pfp: Parallel Fp-Growth for Query Recommendation. In *Proceedings of the 2008 ACM Conference on Recommender Systems* (Lausanne, Switzerland) *(RecSys '08)*. Association for Computing Machinery, New York, NY, USA, 107–114. https://doi.org/10.1145/1454008.1454027

[6] Yannis Velegrakis. 2023. Data Mining Lecture. https://didatticaonline.unitn.it/dol/course/view.php?id=37275 [Accessed: (10/01/2024)].