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

ACM Reference Format:
Agnese Cervino, Enrico Guerriero, Andrea Leoni, and Damiano Zaccaron. 2024. Drivers' Rout(in)e. In *Exams (Data Mining Exam).* Trento, Italy, 13 pages.

1 INTRODUCTION

In a data-driven world, there are many challenges that data mining techniques are capable of successfully address. In this report, the authors aim to provide a possible solution to a data mining problem, related to the world of logistics and merchandise transport. In particular, the following methods could be useful to companies that need to deal with the problem of drivers that are not completely accurate in following the provided instructions. Concerning this specific scenario, the instructions provided by the companies consist of standard routes made up of single trips, each described by a starting city, an ending city and a list of transported merchandise with the relative quantities. In this work, the authors suggest some algorithms that have three different outcomes:

- Given all the standard routes that the company needs to travel and the list of the routes actually traveled by the drivers, the first solution suggests another set of possible standard routes that the drivers are more likely to follow.
- Using the same input as above, the second solution suggests, for every driver, the five standard routes that he or she is most likely to follow.
- Given only the set of routes traveled by each driver, the third solution suggests an ideal standard route that is specific for that driver.

These outcomes are three different ways of improving efficiency in the process of organising work. They are independent, thus they can be implemented individually.
An important assumption that has been made in the generation of data is that routes are assigned uniformly to the drivers. This reflects on the output produced by our algorithms. However, even if real data does not follow this assumption, the code is written so that the output produced will still be optimal. For the first problem, the author used clustering techniques over the actual routes, and generated a number of clusters equal to the number of initial standard routes. Then, the centroids of every cluster was taken as the recommended route for that group.
For the second problem, all the actual routes of each driver were combined into an object called *preferences*. All standard routes were compared to this object, resulting in a similarity score. The five standard routes with the highest scores were selected as each driver's preferences.
The solution for the third problem also starts with the object *preferences* built for the second solution. The ideal standard route is built using favourite trips, cities and merch, each weighted according to their importance.
In the last chapter, some experimental evaluation of the solutions are reported. For the first point, the success of the clustering processing is assessed using three indexes. hen, the dataset is divided into a training set (80%) and a test set (20%). After obtaining the recommended standard routes from the training set, two sets of distances are calculated: the ones between the point that represent the recommended and all the actual routes of the test set, and the one between these last actual routes and the nearest standard routes. The two distances are then compared.
Two different methods are used to evaluate the second output. First, the data set is again divided into a training set and a test set. To test robustness, the result of the training set is projected onto the test set. Secondly, the significance of the 5 recommended standard routes for each driver is compared to all other standard routes. The metric used is the average distance from the preferences. The third output is tested in a similar way to the second one; a train set is selected, analyzed to generate preferences and then used to construct the ideal route; then preferences are updated with the test set and the similarity between ideal route and preferences is computed.
Even if this problem, and its solution as a consequence, seems to be very specific, the operational repercussions are extensive. First of all, according to the European Court of Auditors [3], the transport sector accounts for 5.2% of all jobs. Secondly, the reported methodologies can also be generalized to other problems. It is crucial to take into account the human factor when companies logistically organize the work. That is why these methodologies apply to every system that works with orders and compares them with real performances. The only changes needed will be those related to the format of the input data.

Data Mining Exam, January 2024, Trento, Italy

2 RELATED WORK

In this section, the authors present a brief summary of the main technologies, theories and techniques that they will later apply. The purpose is to help the reader better understand the methodologies used for later solutions. It is possible to skip this section if the knowledge of these topics is already deep.

2.1 K-means algorithm

It is an iterative algorithm used to cluster data. Clusters must not overlap, and each point belongs to only one cluster. It starts initially with K random point, where K is a predetermined number of clusters that are taken as centroids. From there, every point of the set is assigned to the closest centroids, and K clusters are created. For each cluster, the new centroid is derived, and the whole process restarts. The algorithm ends when, after calculating the new centroids, no point is assigned to a different cluster. [1]

2.2 Suffix Tree

It is a structure that represents all the suffixes of a text. For an n-character string, the tree will contain n leaves, numbered from 1 to n. All the nodes, except for the roots, have at least 2 children. There are not children edges of the same node that begin with the same character. To reconstruct all the suffixes, it is enough to go through all the branches of the tree. The special character $ is used at the end of every suffix, to mark its completion. Alphabetically, the character $ comes before every other. [4]

2.3 FP-Growth algorithm

In the Python library PySpark, the FP-Growth algorithm is used for the frequent itemset operation. Given a set on transaction, it firstly counts the frequency of the items and identifies those that are frequent, according to a threshold set by the user. Secondly, it encodes transactions using a suffix-tree structure, to avoid generating the candidate sets explicitly. Then, the frequent itemsets can be extracted. Using spark.mllib, a parallel version of this algorithm is available, called PFP. For further information, the authors recommend Li and colleagues [6].

2.4 Davies Bouldin Index

This is a metric used to assess the validity of a process of clustering. In particular, it measures the cohesion and separation between clusters. The lower the DBI is, the better the cluster. A low DBI indicates compact and well separated clusters, while a high DBI suggests that clusters may overlap. Calculation of DBI involves several steps:

- calculation of centroids
- calculation of intra-cluster dispersion: for every cluster, is the average distance between every point and the centroids
- calculation of inter-cluster dispersion: for each pair of clusters, is the distance between their centroids.

The formula for calculating DBI is:

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

Where:

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

[2]

2.5 Calinski Harabasz Score

It is also known as the Variance Ratio Criterion (VRC). This score is used to measure clustering performance. It is calculated as the ratio of the sum of between-cluster dispersion and of within-cluster dispersion. When the index is high, the clusters are well separated. [5]

2.6 Silhouette Coefficient

It is also a measure of clustering performance. It is calculated as the mean distance between a sample and all other points in the same cluster, minus the mean distance between a sample and all other points in the next nearest cluster, all divided by the maximum of these two values. The silhouettes coefficient of a set of samples is the mean of the SC of every sample. The score is bounded between -1 and 1. The higher the value, the denser are the clusters. [1]

2.7 PCY algorithm

This is an algorithm used for finding frequent itemsets, build as a slightly modified version of the Apriori algorithm. In the first pass, along with the count of every single item, a hash table as big as possible in added, with the total count of item pairs. Given a s (support) threshold, all the buckets below this value will not be considered in the following step. The second pass consist in the exploration of the remaining buckets that might contain frequent pairs. After assessing the frequency of these pairs, if it is above s, the pairs are categorised as frequent. This algorithm allows to use less memory in the second pass, and thus be more efficient than the Apriori. [7]

3 SOLUTIONS

3.1 Data

For privacy reasons, no data has been provided. Therefore, a careful analysis of the type of data being processed is necessary to construct an optimal generation algorithm. The data pertains to a logistics company, specifically focusing on trucks that are tasked with transporting goods. Following is a brief analysis of the type of data, from which we can begin constructing the datasets:

- The datasets to be processed are two, and both are in .json file format.
- The first dataset, "standard.json," contains the so-called **standard routes**, which are a series of paths computed by a software exclusively tailored to the logistics company's needs.
- The second dataset, "actual.json," contains the **actual routes**, meaning the paths that drivers have effectively taken during their work. These actual routes often differ from the standard routes.
- Dataset "standard.json" has two fields:
  - *id*: the "id" field serves as a unique identifier for the standard route, taking the form of "s" followed by a number.
  - *route*: the "route" field encompasses all the actual information about the path. The specific structure of this field will be analyzed subsequently.
- Dataset "actual.json" has instead four fields:
  - *id*: the "id" field in this context is identical to that of the standard routes but takes the form of "a" followed by a number.
  - *driver*: the "driver" field contains the unique identifier code of the driver who operated the actual route.
  - *sroute*: the "sroute" field holds the identifier code of the standard route that was assigned to the driver when they completed the actual route.
  - *route*: the "route" field in this context is identical in form to that of the standard route.
- The routes are ordered compositions of trips, with each trip of a route starting in the same city where the previous trip ended, except for the first trip.
- A single trip comprises three pieces of information:
  - departure city ("*from*")
  - destination city ("*to*")
  - merchandise that was transported ("*merchandise*")
- The merchandise consists of two pieces of information: the list of all items that were transported and the quantity of each item.
- There are two strong assumptions that are crucial in constructing the datasets and, consequently, in the algorithms of the solutions:
  - The physical distances between the cities traversed by the trucks are neglected; therefore, geography will not be taken into consideration.
  - In each city, trucks completely unload their merchandise and load the items intended for the next destination. It is possible to load any type of merchandise and in any quantity in each city.

Certainly, given all this information, it is possible to proceed with the creation of the two datasets.

The programming language used for the project is Python, including the dataset generation. From the perspective of data structures, the datasets are generated as lists of dictionaries; however, once imported, they will have their own classes to fully leverage the object-oriented programming offered by Python.

standard.json construction

As a first step, a file named "provinces.csv" was taken, containing the names of all Italian provinces (110). This was done to have a pool of cities for use as both departure and destination points for various trips. Naturally, the code is generalizable for any file containing strings indicating destinations. Another type of data necessary for generating the dataset is a pool of merchandise. For this purpose, a choice was made to generate strings formed by a random number (ranging from 3 to 9) of random letters.

The dataset construction was carried out through the use of small functions. Therefore, for describing the method, The adopted function will be followed:

- *province_reader*: a function that read the .csv file and returns a list of provinces.
- *province_cutter*: a function that take as input the list of provinces and an integer ($\leq len(\text{provinces})$).
- *merchandise_generator*: a function that takes as input an integer tot_merch and generates a list of random string of length tot_merch.
- *randomizer*: a function used to randomize the data multiple times throughout the code.
- *trip_generator*: a function that generates a trip: it takes as input a province set (that could be the whole province set or, as it was done, a subset of the province set, randomly taken thanks to province_cutter), a merchandise set, the start province of the trip and the number of items that every trip has. Regarding the creation of the trip, the function simply constructs a dictionary in the following way: with the key "from," it assigns the value of the departure city (taken as input), with the key "to," it assigns as the value a city randomly selected from the input set of provinces, and with the key "merchandise," it assigns as the value a dictionary composed of n items, as n is the value taken in input. The keys are the names of the merchandise (randomly chosen from the merchandise pool), and the quantities are random numbers.
- *single_sr_generator*: a function that generate a standard route; it takes as input the province set, the merchandise set, the number of object a trip should have in mean and the number of trips. The function randomly selects a starting city from the city pool. Subsequently, it generates n trips, where n is the input number of trips, and each trip has as its starting city the destination city of the previous trip, except for the first one. Specifically, the trip_generator function is iterated n times, taking as input the average number of items each

trip has, summed with an integer white noise. It then returns a list of trips.
- *standard_routes_generator*: The function that ultimately creates the standard routes: it takes as input the number of standard routes to generate, the reduced pool of provinces, the average number of items per trip, the average number of trips per route, and the merchandise pool. The function simply creates a list of dictionaries, where in each dictionary, the key "id" corresponds to an increasing identifier code of the form "s" + integer, and the key "route" corresponds to a standard route generated using the single_sr_generator function. The function then returns the list of dictionaries.

actual.json construction

The creation of the actual.json file, although very similar in structure to standard.json, followed a different process. Indeed, the actual routes were generated as random alterations of the standard routes. As done previously, the creation process will now be explained by describing the brief functions that constitute it (assuming the existence of functions explained earlier):

- *drivers_generator*: This function takes an integer as input and returns a list of strings of that length, where each string is random and corresponds to the name of a driver. Certainly, it is possible to replace this function with an actual list of names.
- *n_merchandise_randomizer*: The function takes a merchandise dictionary as input, adds an integer white noise, randomizes the number of items while maintaining the same expected value as the original number of items.
- *t_merchandise_randomizer*: The function randomizes the items of the merchandise: it can add, remove, or modify them.
- *single_ar_generator*: The function takes as input a standard route and alters it, generating an actual route. It also takes as input the set of provinces and merchandise. With a predefined probability, it can modify each city in every trip, add or remove trips, and change the merchandise using the previously mentioned functions.
- *actual_routes_generator*: The function 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, and alters them using the previously mentioned function. It also organizes them into the required format and returns the set of actual routes.

Both the actual routes and the standard routes are then transcribed into a .json file with the assistance of a 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 following:

- 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.

Certainly, 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 things randomly extracted are the driver and the standard route they must execute. Since both do not have intrinsic characteristics but are randomly generated and equal in expected value, this random component is not taken into consideration. So the first random modification is to change a city within the route. The probability that not even one city is modified is as follows:

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

After that, the probability of adding or removing a trip, and thus a city, from the route comes into play. This also depends on the number of trips, as the previously shown probability is for adding a trip first, between each existing trip, or last, while the probability for removing it is for each existing trip. The probability that not even one trip is added is as follows:

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

So now we consider the probability that a trip is removed; the probability depends on the number of trips, which has just been

altered by those that have been added. In particular, now the number of trips has become the initial number plus the expected value of the binomial distribution used to calculate the probability of adding a trip. Specifically, $E(X) = (n_{city} + 1) \times P_{Cadd}$, where $E(X)$ is the expected number of trips that have been added. The probability of not removing a trip is then:

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

It is then possible to calculate the expected number of trips that have been removed: $E(X) = [n_{city}-1+(n_{city}+1)\times P_{Cadd}]\times P_{Crem}$ where X is the random number of trips removed. Therefore, the total expected number of cities after these modifications is:

$$\begin{aligned} n_{city\_new} = n_{city} \\ + (n_{city} + 1) \times P_{Cadd} \\ - [n_{city} - 1 + (n_{city} + 1) \times P_{Cadd}] \times P_{Crem} \end{aligned}$$

Now we move on to the analysis of the probabilities that the merchandise undergoes changes, assuming that the number of trips is the just calculated expected number of cities - 1.

The first modification that the merchandise basket can undergo is that an item is replaced by another not present. Therefore, for each trip and for each item in each trip, this probability must be considered. The probability that no item is modified is:

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

Now we need to consider the probability that items can be added; the reasoning is similar to that of the trips:

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

So the expected value for the number of items added is $E(X) = P_{Madd} \times [n_{city\_new} \times (n_{item} + 1)]$. Similarly to the trips, the probability of reducing the number of items follows:

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

From here, it is possible to derive the expected number of items removed: $E(X) = \{n_{city\_new}\times n_{item}+P_{Madd}\times[n_{city\_new}\times(n_{item}+1)]\} \times P_{Mrem}$ So the total expected number of items in a row is:

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

In the end, the probability of not modifying the quantities of each individual item is calculated:

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

The 7 calculated probabilities are for leaving the standard route unchanged in one of the seven randomization phases. However, the probabilities are not independent of each other, so the probability of obtaining an actual route that is identical to a standard route cannot be calculated as a simple product of the previous ones. A way to compute the probability P to not change a standard route is:

$$\begin{aligned} P = P_1 \times (P_2|P_1) \times (P_3|P_2 \wedge P|1) \times (P_4|P_3 \wedge P_2 \wedge P_1) \\ \times (P_5|P_4 \wedge P_3 \wedge P_2 \wedge P_1) \times (P_6|P_5 \wedge P_4 \wedge P_3 \wedge P_2 \wedge P_1) \\ \times (P_7|P_6 \wedge P_5 \wedge P_4 \wedge P_3 \wedge P_2 \wedge P_1) \end{aligned}$$

This means just that n_{city} and n_{item} do not change; so the final formula is:

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

Therefore, in light of this probability, it is possible to adjust the various error probabilities, with the awareness that 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 anyway 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 the language used is Python, object-oriented programming is leveraged to its fullest for processing routes. In fact, while lists and dictionaries were used during the creation phase, when handling data from JSON, it is preferred to create specific classes for objects. Every object that was initially created as a dictionary during the creation process now has its own class. More precisely, the following classes have been created: Merchandise, Trip, StandardRoute, and ActualRoute (which is a subclass of StandardRoute). Each key in the dictionary has become the name of an attribute, with the content of the dictionary associated as its value.

3.2 Recommended standard routes

This section is dedicated to the creation of the first output. The task involves constructing a set of recommended standard routes, replacing the existing one. In contrast to the previous set, which was generated by software and tailored to the company's needs, the new set should be based on driver's unknown agenda and preferences. It is therefore assumed that whenever a driver deviates from the standard route in the current set or alters the transported goods, they are aligning with their agenda or preferences. Given that a driver's preferences can only be deduced from the routes actually traveled by the driver, and it is indeterminable whether a route taken by a driver was assigned to them due to error or personal necessity, the set of routes utilized for generating this output is that of actual routes. The algorithm is founded on the hypothesis that the most pronounced indication of a driver's preference is not observed in individual deviations from standard routes, but rather in the repetition over time of routes, city stops, and transported goods. For this reason, the only decisive set for generating the initial output is that of actual routes.

Theoretical concept

The underlying concept of the following algorithm is to aggregate the actual routes into clusters within a suitable space and derive the recommended standard routes as centroids of these clusters. This approach allows each actual route to be "represented" within a cluster, while the centroid serves as the balancing point that mediates among all routes. Additionally, it is feasible to constrain the number of clusters to match the number of standard routes, ensuring coherent recommended standard routes. However, this does not impose a requirement that each recommended standard route represents an equal number of actual routes. To apply this technique, it is imperative to first construct a suitable space for the analysis.

Space definition

It all starts with the creation of a vector space. Let N be the number of different cities that appear in the actual routes, M the number of different types of merchandise, and T the number of different trip combinations found in the actual routes (for example, a trip from "Trento" to "Verona" is saved as "Trento:Verona"). The space is going to have $N + M + P$ dimensions, one for each city/type of merchandise/trip. Each actual route is represented by a point in the space. The coordinates of these points depend on the cities, the types of merchandise and the trips: every dimension corresponding to cities takes value 5 if that city appears in the actual route, 0 otherwise. Similarly, each trip present in the route will have value 10 in that coordinate, while 0 if absent. Regarding the merchandise, its value when present in a route is significantly lower. This is done to ensure that the merchandise is considered in the distance calculation for clusters but is not overly decisive. Specifically, the value assigned to the merchandise in a route is the quantity of that specific merchandise in the route divided by the sum of all quantities of all merchandise in the route. With this process, every actual route is represented as an array of float values (where every value of the array corresponds to a coordinate on the space). The array of arrays representing the actual routes is then saved in a file called coordinates< run_id >.csv. In the first row of the file the list of dimensions will also be written.

Clustering with pyspark

The k-means algorithm of pyspark is used. It takes in input a table of actual routes. In particular, each row represents an actual route and each column is a dimension among the ones in the space. Every cell of this table contains the value of the actual route in that dimension. This data are extracted from the coordinates< run_id >.csv file saved before. The file is then read using the pyspark method $read$, which allows the framework to convert directly the content of the file in a DataFrame (data type used to perform the clustering). Providing the additional options $header$ and $inferSchema$ to the $read$ method, pyspark reads the first row of the file as the header of the table and infer the data type of the columns automatically. Let K be the number of standard routes. K is used as a second parameter, so that the number of resulting clusters is equal to the number of standard routes and the program will produce K recommended routes. After clustering the data, the program provides the values of the centroids.

Build routes from cluster centroids

Now it is necessary to build back the recommended standard route, starting from the centroids. As it appears clear, for every centroid, every dimension with value equal to 0 would not be considered in the recommended route, that is the same for the types of merchandise and the trips. In order to extract the relevant cities from the clustering, the following steps are done for each cluster center. The values of the cities present in the cluster center are sorted in descending order, removing the values equal to zero. After that, it is needed to have the number of cities that will be in the recommended resulting standard route. This number can be obtained by taking the average number of trips per route and adding one (because each city in the "to" field is the city present in the "from" field of the previous trip and the +1 is the city in the "from" field of the first trip). The minimum between this value and the length of the array of values is set as a threshold index and the value of the array in the position of this index (-1) is the threshold value. Every city coordinate that has a value greater or equal than the threshold value is going to be in the recommended standard route. The same process is done to find the relevant trips for the current cluster center, but this time the average number of trips per route will not be incremented. Differently, all the merchandise that have value greater than 0 is taken into consideration for the entire route. In particular, to select the merchandise carried in each trip, a user-defined function has been employed. This function takes as input a set of actual routes and the space in which they are defined as points. It then returns, for each destination city of every trip, the merchandise that is most frequently carried and the average quantity in which it is carried. This way, it is sufficient to input all the actual routes of a single cluster along with their corresponding space into the function to obtain the most frequent merchandise and the average quantity carried for each destination city. The decision to divide the output by destination city rather than by trip, as it might initially seem more logical, stems from two considerations. First, from a practical standpoint, for a transport company that can

load any type of resource in any city, the transport focus is on the city where the products are delivered. Second, by narrowing the dataset to a single cluster, the likelihood of having two trips in the recommended standard route going to the same city is very low. Therefore, it is not worth burdening the program by computing trips instead of cities for a very remote scenario where two trips have the same destination (and thus the same merchandise).

In conclusion, the results obtained from the actual routes have been synthesized into clusters, each with identified centroids, which have been translated into recommended standard routes. The expected outcomes with varying input data are that the more drivers faithfully follow the standard routes, the less these routes will differ from the recommended standard routes. This makes sense, as it implies a reduced impact of the driver's agenda and preferences on driving decisions or a preference for features already present in the standard routes. In both cases, it is reasonable to expect minimal variations in the standard routes. Similarly, if the routes were to significantly differ, noticeable changes in the recommended standard routes would be observed.

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 , and define their similarity, has been considered. Since it is possible that the customer company possesses vast amount of data. In this scenario, a need to reduce the dimension of the data to process emerges. For instance, supposing the company's data consists of 100 drivers with an average of 300 travelled actual routes and 200 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. To overcome this problem, the authors provide another possible solution, that aims 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 second 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 the 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 calculate 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 section, 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. Particularly, it is assumed that a driver prioritises the trips over the simple destination. Furthermore, the ideal route is designed starting from the path, and the merchandise is then assigned accordingly to the destination. Moreover, the needs of the company were considered secondary to the preferences of a driver. Also, we suppose that a route can only pass through a city only once. For clarity reasons, from now on a trip refers to a path between a starting and ending cities, 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 fixated as the 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-Harabasz 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

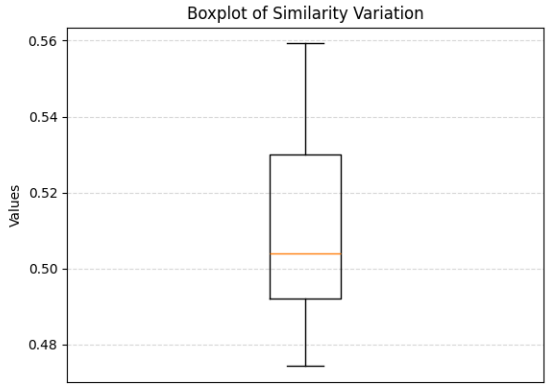
To test the second output, two different approaches were considered: the first one involves using the traditional method of the train-set/test-set, where the dataset was randomly split into two parts. The results obtained using only the train set were then projected onto the test set to assess its robustness. 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.

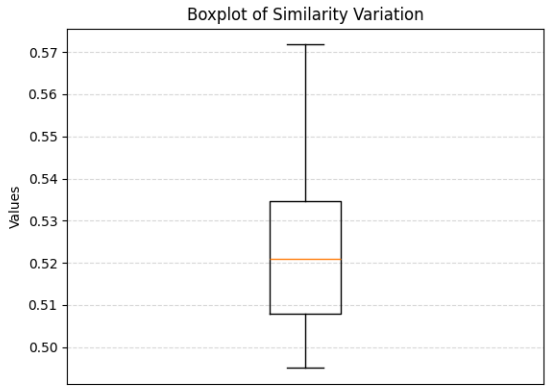
Training - Test

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

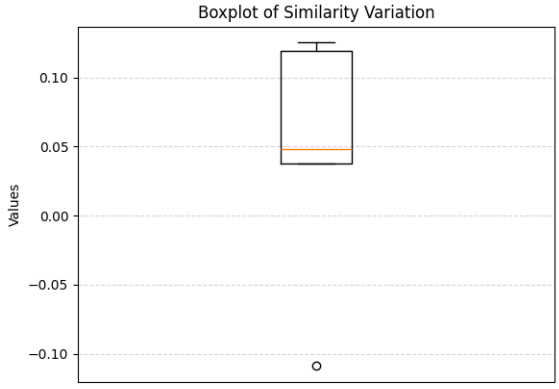
Regarding the second test, the significance of the 5 recommended standard routes compared to all other standard routes was tested. To do this, the average distance of the top 5 standard routes from the preferences is compared with the average distance from all the remaining standard routes and the preferences. 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 three different solutions has highlighted the different approaches available, but also provided an understanding of their respective strengths and considerations.

The extensive testing carried out in the fourth section of our study highlights the ability of the first proposed solution to generate meticulously ordered and non-overlapping clusters. These clusters serve as the basis for the efficient derivation of centroids, which in turn contribute to the creation of a curated set of recommended standard routes. The analysis shows that these recommended routes outperform the original routes, indicating a tangible and quantifiable improvement. This highlights not only the effectiveness of the initial solution, but also its potential to improve route optimisation. The ability to derive superior routes from the identified clusters reinforces the viability of this solution as a valuable tool for optimising route planning processes, promising tangible benefits in terms of efficiency.
The evaluations carried out for the second method demonstrated its effectiveness in successfully identifying five standard routes for each driver. These selected routes show a significant increase in similarity when compared to the actual routes driven by the drivers. This result underlines the ability of the method to optimise route selection and make it more similar to the patterns observed in real driving scenarios. The results 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 patterns.
The results of the tests conducted underline the effectiveness of the third method in creating a standard route for each driver. This method has demonstrated its ability to identify and recommend routes with a greater similarity to those actually travelled. The increased similarity index not only validates the effectiveness, but also highlights its potential to make a significant contribution to route optimisation. By consistently producing standard routes that closely match the patterns observed in real-world travel, this methods provides drivers with routes that better reflect their preferences and historical travel behaviour.

It is important to note that while the code is designed to run even with minimal data, the greater the amount of data fed to the code is and the better the output will be. This is an inevitable issue given the nature of the data: as more data is gathered, the preferences of the drivers and the pattern in the data become clearer and lead to a well defined output.

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] Corte dei Conti Europea. 2018. Le sfide da affrontare per un settore dei trasporti efficiente nell'UE. https://doi.org/10.2865/474101

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

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

[6] 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

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