Navigating Patterns: Unveiling and Ranking Routes

Felix van der Waals
felix.vanderwaals@studenti.unitn.it
University of Trento
Trento, Italy

Dennis Götz
dennis.goetz@studenti.unitn.it
University of Trento
Trento, Italy

Unai Marin
unai.marin@studenti.unitn.it
University of Trento
Trento, Italy

KEYWORDS

frequent routes, frequent sequences, datasets, K-means, clustering, minhashing, Jaccard similarity

1 INTRODUCTION

This report aims to explain and argue about the solution we have given to the following problem, which we summarize briefly:

A company is responsible for delivering different kinds of merchandise among different cities. In order to do such a thing, it hires drivers to whom assigns specific routes they should follow. For instance; go from Verona to Trento and deliver 10 apples, then go to Venezia and deliver 4 bananas and 3 pens, etc. The problem resides in the fact that the driver often adds and omits cities and merchandise at his will, which causes a considerable discrepancy between what they are told to do (standard routes), and what they actually do (actual routes). Our job is to come up with a solution to the following three tasks, so that the company can use it to minimize the above discrepancy: propose better standard routes; for each driver, sort the existing standard routes depending on his tastes; propose an ideal standard route for every driver.

Another application of a solution to this problem might be found in internet traffic. When it is possible to track consumer behaviour on a site, one could use the activity of the user as merchandise by saying that certain actions fall in the same category (type of merchandise). The navigation between pages can be seen as city-connections (going 'from' the current page 'to' the next page when doing 'these' actions). Then the actions on a page can be seen as actions that caused the consumer to go to the other page. The result of the patterns could, in turn, be used to improve website design and track consumer behaviour.

The interest of this problem lies in converting the data into a format for which Data Mining techniques can be used in a proper way. How connections between attributes can be kept intact while minimizing the amount of processing power needed to find trends. If we do not convert the data into a different format it has infinite possibilities; in a way we need to say when something is related to something else. In this report clustering, K-means, Minhashing and frequent itemset mining are used, among others. These methods, by themselves, can not contribute much to a solution, but combined in a clever way they can be used to make sense and profit from an enormous database.

To start with, we are working with a huge amount of data, often pushing us to be clever in the implementation of the techniques we just mentioned. Of course, running algorithms takes time, and we sometimes have to opt for a less precise path for the sake of efficiency.

Secondly, the lack of information about the company and its interests. For example, we do not get to know their data, so we are forced to generate pseudo real drivers in the most general fashion possible. That is, we have to simulate reality, just for us to be able to test and run the solution we come up with. However, the latter should do the job for real drivers and, more specifically, the ones that the company hires, which of course are unknown to us. Furthermore, the ambiguity of the tasks: we want to come up with $better$ routes, but what makes a certain route *good* in the first place? Well, one that minimizes the discrepancy with the actual routes, but how can we know how $similar$ two routes are? We want to find $frequent$ routes, but what decides if a route is frequent or not?

In the attempt to solve these questions, an issue arises. Words like frequent, good, reasonable, enough, start to pop up. All of them are human adjectives, therefore our opinion, and a solution based on concepts like that risks to be not data driven and fail under specific input data. In fact, with this project we have learned that, most probably, no perfect solution exists, as modeling the infinitely complex reality is doomed to have flaws.

In this report we propose a solution to the problem and describe the way of generating synthetic data with which the solution have been tested.

2 RELATED WORK

Before describing the proposed solution some common concepts and algorithms will first be explained in this section. First K-means with it's underlying concepts will be explained. Then the Apriori-concept and lastly Minhashing and Jaccard similarity.

2.1 K-means

In the proposed solution the K-means clustering method is used. This algorithm clusters points in a space. In the solution Euclidean space is used. The algorithm works by choosing K random samples in the dataset and calculating for each point in the space to which of the K points it is the closest. When this is done, the average is taken and now K centroids exist. All points are again assigned to a specific centroid and this process is repeated until the centroid doesn't change as much anymore. If the centroid does not converge the amount of iterations can be limited. [1]

The problem with K-means clustering is that the amount of clusters that the algorithm is looking for needs to be defined. When clustering data that is unknown this is problematic because nothing can be said about the expected amount of clusters. To solve this there is a techique that uses the silhouette width to predict the best amount of clusters. [1]
The silhouette width is defined as follows:

$$S_i = \frac{b - a}{max(a, b)} [2] \quad (1)$$

Where a is the mean distance of the samples within the same cluster and the sample i. In the equation b is the mean distance of the samples outside the cluster and the sample i. When this is done for every sample the average silhouette width can be calculated. When clustering samplesets multiple times for a different K the average silhouette width can be maximized and the optimal K for clustering can be determined.[1]

Another property of K-means clustering that needs to be taken into account is that values in different dimensions can have different ranges. This results in the prioritization of some dimensions to be clustered. This is unwanted in situations where each dimension needs to be considered as equal; a normalization method is required. When no Gaussian distribution can be assumed MinMax scaling can be used. This way all numbers are linearly assigned values between 0 and 1 based on the following formula:

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}} [3] \quad (2)$$

Where x_{max} is the maximum occurring value for x and x_{min} the minimum value. After this is applied on all dimensions, K-means can be applied.

2.2 Apriori

The Apriori algorithm is a popular method for discovering frequent itemsets in large datasets and for identifying interesting relationships or patterns within data. Here is how it works:

Given a set of sets, it starts by finding frequent elements (singletons) by going through all the sets. The word frequent must be understood as having more individual occurrences than the threshold; that is, the threshold is a user-defined parameter representing the minimum number of occurrences required for an item to be considered frequent. Afterwards, it combines frequent singletons to form couples, then frequent couples to form triples, and so on.

The key idea behind the Apriori algorithm is the "Apriori property," which states that if an itemset is frequent, then all of its subsets must also be frequent. In other words, if $\{A, B\}$ is a frequent itemset, then $\{A\}$ and $\{B\}$ must also be frequent. However, thinking on its contrapositive turns out to be even more useful; if $\{A\}$ is not frequent, then trying to find frequent sets that contain A is a waste of time and memory.[4]

Besides, Apriori is also used for association rules, that is, finding out whether A is often in the same subset as B and so on. Nevertheless, that principle will not be used in the proposed solution, for the following reason: in this problem we will have sequences, not sets, meaning that the order is important. Every element is connected to the next one, so they can not be seen as independent individuals. However, the Apriori principle holds even for sequences, and it has turned out to be really useful. Furthermore, as it will be explained, apart from discarding elements that are not frequent, we also discard connections that make no sense, precisely because order is important in this case, so the algorithm is even faster.[4]

2.3 Minhashing and Jaccard similarity

Minhashing is a technique to convert sets of items into signatures by using hashing functions. These signatures have length n where n is the amount of hash functions used. Specifics will not be giving but the most important of Minhashing is that when combining hash functions that cause collisions the set can be described with less data. This holds because for every hash function some specifics are preserved and the combination of these hash functions gives the (almost) unique ID signature. [4]

When the Jaccard similarity is needed between two sets, every element needs to be compared to every other element. The computation time needed to find the Jaccard similarity increases quadratically with the amount of elements in the sets meaning that this is not an efficient way to compare sets. However, with Minhashing signatures the Jaccard similarity can be estimated. This can be done by comparing every nth with every nth element between the two signatures. The number of equal elements in the signature divided by the total length of the signature is the estimation of the Jaccard signature. The more hash functions the signature has, the better this estimation is. However, the time to compute the signatures and compare also increases.

3 SOLUTION

For the solution, the programming language Python in combination with Jupiter Notebook has been used. The proposed solution is composed of 3 parts, one for each task:

The first part finds a list of new standard routes with all the actual routes as an input. Shortly, it first breaks routes into trips, manipulates trips to join those that are similar, and then tries to find maximal frequent sequences of trips within the actual routes.

The second part takes all the actual routes from a specific driver and some routes that are to be ranked based on preference, that is, based on those actual routes. It sorts them out by converting all the routes into sets and those sets into signatures, using Minhashing. Then all routes that need to be sorted are compared with all the actual routes from the driver, and are sorted from most to least similar.

The third part uses the first and second part to find the perfect route for a specific driver. Intuitively, the first part took a set of actual routes and output the *best* (in a way that the offered routes capture the nature of other routes too) and most frequent subsequences. Then, the second algorithm sorts a set of routes based on the preferences of a driver. As a consequence, applying one algorithm after the other one on a single driver and finally picking

the first option from the sorted routes results in finding the ideal route for that driver. Of course this is limited to the data the driver has, if the driver did not drive routes that he liked the consequence is that this solution can only give the route that is the best possible route.

Lastly there is a section in which possible ways to optimize the solution are discussed.

3.1 Finding standard routes

Let us recall the task: given the standard routes that the company has offered and the actual routes that the drivers have driven, the aim is to provide the company with new standard routes in a way that the discrepancy with the tastes of the drivers is minimized.

A naive way of solving this first problem would be to count the occurrences of every actual route and rearrange them all from most to least frequent. Then, the company could pick as many routes as it wanted, and they would always be the "best" ones, would they not?. In fact, this method has several problems: first of all, each driver has its own preferences, which may differ from day to day, so most probably we would encounter too few occurrences per route, which would then mean that no route is especially frequent. Besides, routes would be treated as individual elements, in a sense that picking the most frequent one gives no information about other routes, rather than that they are less frequent. In short, we need a way to reduce complexity. Here is what we do in few words:

Firstly, we treat every possible trip as an individual element, where "trip" involves city-connection and the delivered merchandise. As it will be explained, these trips can in some sense be seen as Euclidean points in some spaces. Shortly, each city-connection defines a different space. Then, points from a specific space are clustered; in this way, similar trips are put together, and after picking a representative, that point would be giving information not only about the specific routes it belongs to, but also about other similar trips. Note that in this way, once all trips are substituted with the respective representatives, memory and diversity of items is enormously reduced. Finally, we convert every actual route into a sequence of representative trips, and try to find maximal frequent sequences among them. The latter concept will further be explained in detail. It is strongly believed that this method solves the problem in an elegant way, simultaneously overcoming the issues that were explained at the beginning of this part.

To do so, the problem is again split, this time into two parts. The first part will explain how trips are generalised into similarity-groups and the second part will explain how maximal frequent sequences are found.

3.1.1 Clustering subspaces. In this section the connection between city-pair and merchandise is determined. Because merchandise has a quantity, clustering can be used. Firstly the space is divided into subspaces of city-pairs (the first grouping). Secondly, these subspaces are divided into possible clusters with specific merchandise (the second grouping). To apply clustering, the data needs to be transformed into vectors.

To do so, the space of all possible trips is divided into subspaces where every different connection between two cities introduces a new subspace, independent of the route it is from. Within these subspaces, every type of merchandise that has occurred at least once appears as a new dimension. For example, Trento-Verona would define a euclidean space where each axis corresponds to some merchandise that has occurred at least once in that city connection.

In order to do this as efficiently as possible, for every city-pair a dimension map is created, describing for every type of merchandise the position it holds in the vector, which is fixed for every subspace; the first dimension may be "Tea" and the second could be "Rice" depending on the city-pair. This is done by going through all the trips in the actual routes. Then the map is built as a dictionary, and has the following form:

{ 'Verona-Modena': ('Tea', 'Rice', 'Beer', 'Pens', 'Pasta', 'Butter'),
'Modena-Pisa': (Pens', 'Pasta', 'Butter', 'Milk', 'Chocolate'),
...
'Pisa-Siena': ('Milk', 'Chocolate', 'Yogurt', 'Tomatoes', 'Cheese'),
'Pisa-Modena': ('Milk', 'Chocolate', 'Yogurt', 'Tomatoes', 'Cheese') }

This map is used for several tasks in this subsection and is referenced as dimension map.

The next step is to convert all the trips of the routes into data points in each subspace. All the data is again scanned and for every trip it creates a vector with the help of the dimension map. Basically, for every route, for every trip, for every merchandise, look at the quantity, find the index of that merchandise in the subspace of the city-pair and plug the number in a vector. Finally, a dictionary is created with every city-pair together with all the points that belong to its subspace.:

```
{
    'Verona-Modena': [
    [0, 15, 14, 0, 2, 6],
    ...
    [10, 1, 0, 10, 0, 0]
    ],
    ...
    'Pisa-Siena': [,
    [0, 4, 7, 0, 0, 5],
    ...
    [8, 3, 0, 25, 0, 0]
    ]
}
```

This list will be referred to as the list with data-points (The vectors are now called data-points).

Every action that will be explained from this point forward, until it is said explicitly, will be performed on the data-points of a specific city-pair. However, the analogous processes are carried out with all the other spaces. Data-points specific to a city-pair will now simply be referred to as data-points.

Once trips are converted into Euclidean points, K-means clustering can be applied. Before this can be done some parameters need to be found and the data needs to be normalised. The normalisation that is used is MinMax-scaling. This will prevent any preference for a specific merchandise. Indeed, if the quantity of, for example, pens, is normally above 50 while other elements are below 10, the pen-dimension is likely to dominate the other ones, which is obviously undesired. MinMax-scaling prevents this by taking the data-points and converting them into floating numbers between 0 and 1 considering the minimum and maximum value for every dimension.

In the first place, the K needs to be specified, for which the K-means clustering is applied on smaller subsets of the data-points. The amount of samples that are used is given by the following function s(x):

$$s(x) \begin{cases} x & \text{if } x \leq 100 \\ 98 + 3 * (\sqrt{x - 100}) & \text{if } x > 100 \end{cases} \tag{3}$$

Where x is the amount of points. In the code the output of s(x) is rounded. This function is used to prevent excessive time spent on finding K when having long lists of data-points while at the same time allowing for some growth on the amount of clusters. This function was found empirically, and turns out to do the job, considering time-expense and efficiency. The maximum amount of clusters that needs to be tested is found by using the following function:

$$k_{max}(x) = \sqrt{s(x)} \tag{4}$$

The square root in the equation is used for the same reason as for the $s(x)$ formula. The output of this function is also rounded. A visualisation of the two functions can be found in the Figure below:

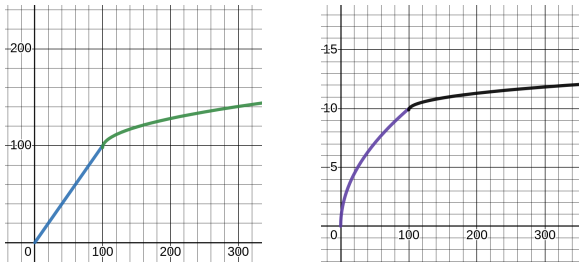

Figure 1: Graphs of functions $s(x)$ and k_{max} respectively with amount of data-points on the x-axis

Once the quantity of samples is fixed, elements are randomly and without replacement sampled out of all the data-points. However, it could happen that all samples turn out to be the same: there could be just one point in the set of candidates, or even if there are more, they could all have the same value for every dimension. So it is checked whether the samples that are taken are all the same (this is done by checking if at least one of the points is different than the first point in the list). If all the samples in the sample data-points are the same, new samples are picked. This is done 5 times and if the samples are still all the same it can be said that K must be 1 because, with high confidence, only one unique data-point exists in the data-points. (This is mainly done to prevent errors because silhouette score can not be applied on 1 unique sample)

When there are at least two different points within the samples and the maximum expected clusters is bigger than 2, then it is tried to find the best K. If the maximum expected clusters is 2, it means that the amount of samples is in the set: [4, 6]. In this case clustering will be ignored. K-means (as described in RELATED WORK) is used for every K between 2 and the maximum expected clusters, which has been done via the "sklearn" library in Python. The average silhouette score is calculated for every K; if the maximum of all those values is above 0.7, the K corresponding to that maximum silhouette score is taken as the K for the clustering on the entire set of data-points. The threshold of 0.7 was found by trial and error. This value is rather high because it is preferred to only have one cluster rather than many "bad" ones, the reason being that the more clusters we have the more split is our data, which will later imply less frequent items and therefore loss of patterns. In other words, we want to make sure that clusters are considerably good. Once the K is known, the normalisation will now be applied on the whole set of points and K means is run. Then a dictionary is created containing every point in the subspace and the cluster it belongs to. This dictionary has the following format:

```
{
    'Verona-Modena': {,
    [0, 15, 14, 0, 2, 6]: 1
    ...
    [10, 1, 0, 10, 0, 0]: 0
    },
    ...
    'Pisa-Siena': {,
    [0, 4, 7, 0, 0, 5]: 2
    ...
    [8, 3, 0, 25, 0, 0]: 0
    }
}
```

The numbers after the data-points in this dataset are specifying the cluster it is assigned to. Multiple duplicate points are mapped to one entry in this format. This means for example that if there exist 10 duplicate points in a subspace only one point is added to the dictionary for a specific city-pair.

Once clusters are specified, it is necessary to decide what point to take as a representative. Intuitively, every trip will be substituted by this representative, and it should somehow minimize the discrepancy with the other points. The very first option that was thought was the centroid, as this would minimize the sum of the distances with all the other points. However, this may result in a centroid consisting of a combination of merchandise that is very different from what we see among the actual routes. The reason

for that is that a generic point in the subspace will most probably have non-zero values for every dimension; however, actual routes rarely have positive values for every possible merchandise. In order to avoid this, we choose to pick the nearest point to the centroid, which indeed is a driven trip. As a downside, this nearest point could be far away, if the cluster is quite spread; then our representative would not be ideal. Nevertheless, it is thought to be best option. Finally the point is denormalized. For clarification: the variable has a format similar to this:

```
{
    'Verona-Modena-0': {repr: (23, 0, 0, 0, 21)},
    'Verona-Modena-1': {repr: (0, 15, 42, 0, 21)},
    ...
    'Lecce-Verona-3':  {repr: (0, 0, 21)}
}
```

The previously described actions are performed for every possible city-pair resulting in a dictionary as above. The loop where actions are performed on the data for every city-pair is ended here.

The final step is to convert the original routes into sequences of city-pairs with a cluster specifying the type of merchandise. This is done by scanning all the routes and substituting every trip with its cluster identifier. The result has the following format:

(
('a1', 'I', 's1', ('Modena-Pisa-0', 'Pisa-Siena-0')),
...
('a1532', 'A', 's4', ('Modena-Milano-2', 'Milano-Siena-0')),
)

The first attribute gives the identifier for the actual route. The second attribute gives the driver id that the route was driven by. The third attribute is the id for the standard route that was assigned to the driver. The last element is the ordered set of trips. The data that is used for finding the maximum frequent sequences is the converted routes, the dimension map and the representative point for each cluster.

3.1.2 Finding maximal frequent sequences. As we have seen, every route is converted into a sequence of cluster-trips. The aim of this subsection is to explain how maximal frequent subsequences are extracted from the so-called dataset. One thing has to be clear, the goal is not to find frequent sequences, but subsequences. For the sake of simplicity, let us suppose the standard route (1,2,3) is given to a considerably large group of drivers. Let's say all of them add 4, and afterwards each driver adds a random number at the end and at the beginning. It might look like: {(10,1,2,3,4,6), (8,1,2,3,4,100),..}. The offered route will be (1,2,3,4). The Apriori property takes a considerable place in this final process; not frequent items are automatically discarded and even frequent items are only combined when "it makes sense". This is how it is done:

Firstly, a threshold T needs to be defined; anything below T is considered not frequent. The choice of T will be discussed later on, but for now, let us stick to a general value. First step is to create a dictionary freq= where the occurrence of every trip is stored. The result will be something like:

{Milano-Bergamo-0 : 2000, Trento-Verona-4 : 150,...}.

Then, a prune() function is used where prune(freq) outputs two different things: the updated dictionary (all items the value of which is below T are eliminated), and the set of items that were eliminated. Not being eliminated will be referred to as "passing the pruning", and later on will be given the reason for which we store the latter set. Once we have the pruned dictionary with single frequent trips, they are combined to form couples (length-2 routes), and it is done by a function called couples(). Before explaining what the output looks like, it needs to be stated the criterion under which routes are offered.

Once couples are created, we will count the number of their occurrences, keep them in a dictionary, prune again, form triples, count the occurrences,... and so on. Of course, for each singleton, the process terminates: it can not be extended indefinitely and still be above the threshold. So we state the following: a sequence is offered if and only if its singletons are frequent and it can not be extended into a larger frequent sequence. In this way, we will find maximal subsequences that are very frequently part of actual routes (in this section actual route really means cluster-actual-routes), and that, if extended, they are no longer frequent. Sticking to the above example, (1,2), (1,2,3), are also frequent, but they can be extended to (1,2,3,4) which is also frequent. However, this is not true for (1,2,3,4); it most probably can not be extended to something frequent, so we offer it. That explains why we store the sequences that don't pass the pruning; if an extended sequence is not frequent enough any more, we go back one step and offer it.

How are couples created? Suppose we have a set of trips {(Milano-Trento), (Venezia-Trento), (Trento-Verona)}. If we were to extend those trips to 2-trip-routes, one naive way to do it would be to create all possible couples. However, not every combination makes sense. For example, (Milano-Trento, Venezia-Trento) is not possible as a route. Moreover, checking whether a certain combination makes sense before creating it will save us a lot of memory. Therefore, we only create feasible couples. Besides, we also store the singletons we could extend, and the ones we could not. The latter will directly be offered, as they are frequent and can not be further extended to a frequent sequence (satisfies the criterion). Why do we store the singletons that could be extended? Well, in case the extension is no longer frequent; if so, we would offer the singletons that were extended to something not frequent, as those too satisfy the criterion. The output of couples({(Milano-Trento), (Venezia-Trento), (Trento-Verona)}) would be:
([(Milano-Trento,Trento-Verona)], [(Venezia-Trento)], [(Trento-Verona), (Milano-Trento)]).

Next step would be to count the number of occurrences of the couples and store it in a new $freq$ dictionary, and then prune it. We then introduce a function combine(), which is the exact analogue of couples(), but input is a list of tuples of at least length 2. Why is it really necessary to divide this task into two different functions then? Well, for two singletons to be combined, it was needed that the last city of a trip be the same as the first of another trip, and for

that purposes, we had to manipulate strings of characters. Now, if we are given two longer sequences, say (1,2,3,4) and (2,3,4,5), what we have to check is whether the last three elements of the tuple coincide with the first three elements of the other one, in order to create the tuple (1,2,3,4,5). This is a tuple-wise comparison; nevertheless, the structure remains the same, and we also store sequences that could and could not be extended, for the same reasons we do in the case of couples().

The process looks like:

- Count singleton frequencies.
- Prune singletons.
- Make couples.
- Prune couples; for every not frequent couple, directly offer the singletons that form it.
- Combine couples into triples; offer the couples that could not be extended.
- Prune triples; for every not frequent triple, directly offer the couples that form it.
- Combine triples into quartets; offer the triples that could not be extended.
- etc...

As was said, the process eventually terminates, as there will be a point where no extension is longer frequent. It also has to be mentioned that the number of occurrences are stored all along the process, so that we are able to sort the list of offered routes from most to least frequent. Finally, sequences are converted back into routes, with the correct format.

There is still one issue to discuss: how is threshold T specified? Note that if a certain sequence (2,3,4) is offered under T, by lowering T, that is, by requiring less occurrences in order to be frequent, (2,3,4) is never lost, but in any case extended to something like (1,2,3,4), if that were to happen. As a consequence, $M \leq T \Rightarrow \text{offered\_list}_M \geq \text{offered\_list}_T$. Even though, we pay the price of offering less frequent sequences, intuitively "worse" sequences.

Several options have been studied, and although one of them seems to be the most reasonable, each one has its ups and downs.

- Option 1: Start with a high threshold, say the length of the whole dataset (to ensure that the offered list is empty) and lower it until we get at least as many standard routes as the company offers. Nevertheless, this is strongly not recommended. In fact, due to the fact that drivers change routes freely, it could happen that regardless of the routes they were commended, all drivers follow strictly the same route, so the algorithm would never output more than one route, and it would enter into an infinite loop. Besides, it seems that the problem of the company is precisely that the way they design standard routes has some flaws, possibly the quantity too, so we should not force ourselves to give them the same number of standard routes as they already have. Finally, running the algorithm several times takes time. This option has been dismissed.
- Option 2: fix a reasonable frequency, say %5 of the whole dataset. That is, if there are 100.000 actual routes consider a sequence frequent if it occurs more than 5.000 times. The advantage of this method is that the algorithm is only run once. As downsides, it relies on our decision that %5 is reasonably frequent, and it does not take into account input data. This option has also been discarded.
- Option 3: Let D be the number of actual routes and N the amount of standard routes the company has commended; this information is captured from the actual routes. Then, assuming every standard route is given with the same probability, we would expect $\frac{D}{N}$ to be the number of actual routes driven for each standard route. Considering that, when clustering, different routes collapse into the same cluster-route and that many of them are likely to have overlappings, then asking for a certain sequence to happen in at least half of the $\frac{D}{N}$ actual routes seems reasonable. In other words, we ask each sequence to occur in at least $T = \frac{D}{2N}$ actual routes. Intuitively, the more standard routes were commended the more spread we expect data to be, so a smaller threshold should be used. We show some examples:
  - $N = 20 \Rightarrow 1/2N = 0.025 \Rightarrow$ frequent means "at least 2.5%"
  - $N = 40 \Rightarrow 1/2N = 0.0125 \Rightarrow$ frequent means "at least 1.25%"
  - $N = 100 \Rightarrow 1/2N = 0.005 \Rightarrow$ frequent means "at least 0.5%"
  - $N = 5 \Rightarrow 1/2N = 0.1 \Rightarrow$ frequent means "at least 10%"
  - $N = 1 \Rightarrow 1/2N = 0.5 \Rightarrow$ frequent means "at least 50%"

As it can be seen, there are extreme cases, for instance N=1, where the threshold seems a bit too high. In those cases lowering the threshold is recommended, but not needed (in theory).

3.2 Sorting routes

The goal of this task is to sort some routes based on the routes that a specific driver has driven (intuitively, his tastes). This is done by comparing every route that needs to be sorted (the company's standard routes) with every route the driver has driven (specific driver's actual routes). Then the average of the similarity score is taken for a specific standard route that needed to be sorted. This is done for every route and then they can be ranked based on the average score, describing how much the drivers like the route i.e. how many times the driver has done similar things. Thus, it is assumed that a driver likes a route if he often drives similar routes. To do this it needs to be defined when a route is similar. This is done by taking certain aspects of a route and adding it to a set of keywords that define the route. Then every set can be converted into a signature using Minhashing and these signatures can be compared by using Jaccard Similarity and therefore approximating the Jaccard similarity of the original set with keywords.

First the route has to be converted into a set. This set needs to give information about the route. The following properties of the route are used:

(1) Every city that occurs in the route. Example: 'Milano'

(2) Every city-pair that occurs in the trip. Example: 'Milano-Rome'
(3) Every type of merchandise that occurs in the route. Example: 'Apple'
(4) Every type of merchandise together with a binned quantity indication. Example: 'Apples-Medium'
(5) Every city-pair in combination with the type of merchandise carried for that trip. 'Milano-Rome-Apple'
(6) Every city in a trip in combination with the product carried from or to this city. Example: 'Milano-Apple'
(7) Every type of merchandise in combination with the other type of merchandise carried within the trip it was carried. Example: 'Apple-Banana'

Number 1 will create some similarity between routes that have the same cities in them where order is disregarded.
Number 2 will give information about the connections between the cities within the route. With this information it is almost always possible to retrieve the order in which the cities where visited. Only when a loop is made (a city is visited twice) the original route can not be retrieved, but this can be ignored.
Number 3 gives information about types of merchandise in the entire route. This causes the routes to have some similarities if there is a specific merchandise that they both carried independent of the trip.
Number 4 uses a binned quantity where first the maximum and the minimum for that amount is found independent of the city-pair. These values are used to create 3 partitions; "small", "medium" and "large". The partitions are created using the following formulas:

$$border_{small-medium} = min + \frac{1}{3} * (max - min) \tag{5}$$

$$border_{medium-large} = min + \frac{2}{3} * (max - min) \tag{6}$$

Where min is the minimum value that has occurred for that specific type of merchandise and max is the maximum value of that specific type of merchandise. When creating the sets with keywords the value of borders is used to determine if the amount of items carried is "small", "medium" or "high". This way the amount of items is taken into account without introducing big complexity (introducing possibly infinite possibilities for every type of merchandise).
Number 5 keeps the connection between merchandise and the city-pair.
Number 6 connects cities to merchandise. This means that if a type of merchandise is in any way related to a city (either "from" or "to") it will be added to the set.
Number 7 keeps the connections between types of merchandise within a trip. If the driver decides to go to a different city with the same set of merchandise it will still have some similarity with the original route.

The created sets can be converted into signatures using Minhashing, with the "datasketch" library in Python where 128 permutations are specified. Then the signatures of all routes driven by one driver are compared to each standard route. The comparison is also done using the "datasketch" library and it is carried out by calculating the percentage of equal permutations of the signature. This results in an estimation of the actual Jaccard similarity between the sets. Then, the average similarity is computed, which is defined as the similarity between a driver and a standard route. The list is sorted based on the similarity score and only the top 5 elements are taken. This will be inserted into a JSON file. This is done for every driver.

3.3 Finding the perfect route

To find the perfect route for a driver the two algorithms are combined. First new standard routes are generated based on only the actual routes of the driver. Then the proposed routes are sorted based on similarity by using the sorting method previously discussed. This will result in a route that is based on the preferences of the driver because it has been driven by the driver many times and it is the most similar to all the routes the driver has driven compared to the other proposed standard routes. Of course this is method is used for every driver so all drivers get a "perfect" route.

3.4 Possible optimalization methods

In this subsection some methods are mentioned that could be used to optimize the given solution. They were not used due to lack of equipment, time and/or programming skills.

First of all, map and reduce can be used for some parts of the solution. In generating new standard routes it can be implemented to find the dimension map. The data is divided and the individual workers find possible dimensions (map) after which the result is combined (reduce). After that the city-pairs are packages of data and can be divided between the workers after which they combine the results again. The next step, where the actual routes are converted into ordered sets, can also be improved with map and reduce by dividing the actual routes into equal parts and generating the ordered sets after which the result can be combined. For finding the maximal frequent sequences the map and reduce can be used when counting occurrences. This way the overall speed of the algorithm can be improved.

The second method to possibly improve the solution is by having a better memory management. When using Python it is not possible to specify precisely how memory is used and how data is stored. When using a programming language like C the solution could be improved while also improving the programming complexity. Furthermore, a way to store entities in memory taking less space can be used. For example when dealing with the trip "Milan-Rome-1" it can also be stored as three integers where the "from" and "to" city and the cluster is specified with a single integer. This reduces memory size and therefore decreases the need for reads from memory which makes the solution slower due to the memory wall problem. Overall, memory limitations can be taken better into account when using a different programming language.

Another method that could slightly improve the solution is dimensionality reduction. When the subspaces are generated there are possibly a lot of dimensions. Although it is limited because no products carried between another city-pair are considered, it could still be possible to apply dimensionality reduction to check if there exist some dimensions that can be ignored. Then the clustering

phase will be more efficient because it can use less dimensions.

Lastly, some method to perform entity linkage could be performed. In the current solution it is assumed that every city and every merchandise is a different entity. In a real-world application this can possibly not be assumed because Monaco may be referred to as München or there exist 10 different types of milk. In combination with field knowledge (by having access to the data of the company) this method might improve the proposed solution.

4 EXPERIMENTAL EVALUATION

In this chapter, we will introduce the input dataset specifically generated for our tasks, providing deep insights into its creation. Additionally, we will present the employed similarity measure utilized to assess the effectiveness of our algorithms. The core of this section revolves around the comprehensive evaluation of our solutions, coupled with the interpretation of the results.

4.1 Data generation

To create a diverse input dataset for training and testing our algorithms, we devised a process that involved generating 20 standard routes. These routes were constructed using two lists comprising 20 potential cities and products, along with corresponding quantities ranging from two to 30. Each standard route consisted of a minimum of three and a maximum of seven trips, with the number of products for each trip falling within the same range.
Note: we generated two supplementary sets of standard routes, each consisting out of 50 and 100 routes, and applied them to our algorithms. However, as the outcomes exhibited high similarities, we opted to concentrate our attention on the results derived from the set of 20 standard routes and present only those in our analysis.

The generation of a standard route followed a systematic approach. Initially, a random number of trips for the route was selected, along with a random starting location city. Subsequently, in a loop, a destination city was randomly chosen for each trip from the set of cities not yet included in the route. The number of products for each trip was determined randomly, and the specific products, along with their quantities, were then selected. After completing a trip, the destination of that trip became the starting location for the subsequent trip. This iterative process ensured the creation of diverse and realistic standard routes, incorporating variability in both the number of trips and the products carried, while preventing repetition of cities within a single route.

To operationalize our standard routes and facilitate the creation of actual routes, we initiated the formation of 10 drivers. To characterize these drivers, we compiled a list of 10 unique attributes and assigned two or three attributes to each driver. With these feature combinations we created 10 different characters, what will be useful to evaluate the performance of our algorithms. The aim is to find these attributes of the drivers in the standard routes and in the optimal route we recommend to them in problem two and three. Our possible features for the drivers are described in the following:

- The attribute "*likes short routes*" means that the driver can cut trips off the route if it has more than four trips. E.g. if the standardroute has six trips, he cuts with linearly increasing probability zero, one, two or three trips off the route. The probability distribution in this case looks like: cut zero trips with about 5%, one trip with about 18%, two trips with about 32% and three trips with about 45%.

- The attribute "*likes long routes*" means he can add a trip to the route if it has less than six trips. E.g. if the standardroute has four trips, he adds again with linearly increasing probabilities zero, one, two or three trips to the route. Also cities that are already contained in the route can be selected.

- The attribute "*likes a city*" means that the driver can add specific cities to the route if they aren't included in the standardroute yet. E.g. if he likes Trento and Verona, he adds zero, one or two of these cities with linearly increasing probabilities to the route as trips.

- The attribute "*dislikes a city*" means that the driver can cut specific cities off the route if they are included in the standardroute. E.g. if he dislikes Lecce, Palermo and Milano and only Milano is in the standardroute, he cuts no city or Lecce with linearly increasing probabilities off the route.

- The attribute "*likes few products*" means that he can cut off products of a trip if it contains more than four different types of products. The procedure works like in the feature "likes short routes", but with products.

- The attribute "*likes many products*" means that he can add products of a trip if it contains less than six different types of products. The procedure works like in the feature "likes long routes", but with products.

- The attribute "*likes a product*" means that he can add specific products to a trip if they are not included yet. The procedure works like in the feature "likes a city", but with products. Additionally, he is able to also increase the quantity of his liked products. E.g. if the quantities any of his liked products are below 20, he increases the quantity of all these products up to at least 20 with probability of 85%.

- The attribute "*dislikes a product*" means that he can cut off products of a trip if they are included in it. The procedure works like in the feature "dislikes a city", but with products. Additionally, he is able to also decrease the quantity of his disliked products. E.g. if the quantities any of his disliked products are above 10, he decreases the quantity of all these products down to 10 or less with probability of 85%.

- The attribute "*likes high quantities*" means that he can increase the quantity of products in a trip. E.g. if in a trip the quantities of any products are below 15, he increases the quantity of all these products at least up to 15 with probability of 70%.

- The attribute "*likes low quantities*" means that he can decrease the quantity of products in a trip. E.g. if in a trip the quantities of

any products are above 15, he decreases the quantity of all these products to 15 or less with probability of 70%.

Figure 2 shows how we assigned the features to the drivers to create the characters:

Driver	likes short routes	likes long routes	likes a city	dislikes a city	likes few products	likes many products	likes a product	dislikes a product	likes high quantities	likes low quantities
A	Yes	-	-	-	-	-	-	-	-	Yes
B	-	Yes	-	-	-	Yes	-	-	-	-
C	-	-	Yes	-	-	-	-	Yes	-	-
D	-	-	Yes	Yes	-	-	-	-	-	Yes
E	-	Yes	-	-	Yes	-	-	-	-	-
F	Yes	-	-	-	-	Yes	-	-	-	-
G	Yes	-	-	-	-	-	Yes	-	-	-
H	-	-	-	Yes	-	Yes	-	Yes	-	-
I	Yes	-	-	-	-	-	-	-	Yes	-
J	-	-	-	-	-	-	Yes	-	-	Yes

Figure 2: drivers attributes distribution

In the table you can identify three trends. Firstly, more drivers prefer short routes instead of long routes. Secondly, "likes many products" appear more often as an attribute of a driver than "likes few products". Lastly, more drivers like low quantities compared to high quantities. These global trends among all the drivers can be useful for our algorithm in the first part to identify and suggest mainly standard routes with these features to achieve less discrepancy.

Each driver with one of the attributes "likes a city", "dislikes a city", "likes a product" or "dislikes a product" was provided with an additional list that contained his cities/products he likes/dislikes. These cities and products are shown in the tables below:

Driver	City 1	City 2
C	Trento	Padova
D	Rome	Trento

Driver	City 1	City 2	City 3
D	Milano	Palermo	Lecce
H	Lecce	Palermo	-

Figure 3: liked (left) and disliked (right) cities by the drivers

Driver	Product 1	Product 2
G	Beer	Cheese
J	Chocolate	Honey

Driver	Product 1	Product 2
C	Water	-
H	Milk	Water

Figure 4: liked (left) and disliked (right) products by the drivers

In the final phase of data generation, we aim to create $10E4$ actual routes. For each of the 20 standard routes, we iteratively generate $5 * 10E3$ actual routes. The process involves randomly selecting a driver and applying their unique attribute combination to influence the route. Attributes cause actions such as cutting or adding trips and products or modifying quantities as described above.

4.2 Comparison method

To assess the effectiveness of the newly recommended standard routes in minimizing discrepancies between designated standard routes and their respective actual routes, a robust similarity measure is essential. An approach similar to the Jaccard similarity is employed iteratively in the following procedure: Given a standardroute and an actual route of this standardroute, first, we need to identify which trips in these two routes belong together. This is done by comparing the destination city of all trips as key. If a city destination of the standardroute wasn't found in the actual route's cities, the trip was cut by the driver and gets a similarity score of zero assigned. The same happens if an additional city was found in the actual route, which was added to the route by the driver. For all correctly taken city destinations the merchandise is compared. Here we divide the intersection of correctly loaded product quantities by the union of all product quantities (Jaccard). E.g. if the standardroute has the following trip to Bologna:

('Milk':5, 'Water': 18, 'Tea': 18, 'Apples': 22, 'Potatoes': 19)

But the driver "H" with attributes "likes many products" and "dislikes a product" (Milk and Water) instead loaded this merchandise to his trip to Bologna:

('Water': 8, 'Tea': 18, 'Apples': 22, 'Potatoes': 19, 'Bananas': 8, 'Tomatoes': 15)

Here the driver took the correct quantities of "Tea" (18), "Apples" (22), and "Tomatoes" (19) with a total of 59. Furthermore, he cut the product "Milk" (-5) entirely, reduced the quantities of "'Water'" (18-10=8) and added "'Bananas" (+8) as well as "Tomatoes" (+15) to the trip. This leads to the following similarity score which is comparable to Jaccard since you compute intersection over union:

trip similarity score =

$$= \frac{59 + 8}{(59 + 8) + (|-5| + |-10| + 23)} = \frac{67}{67 + 38} = 0.6381 \quad (7)$$

This score is computed for every identified trip pair in the standardroute and actual route. If a city appears twice in the actual route, the score is computed for both trips and the higher score is taken. The trip with lower similarity score is very likely the one added by the driver and receives a similarity score of zero. In the end the final similarity score between the routes is computed by taking the mean of all the trip similarities. E.g. driver "D" with attributes "likes a city" ("Rome" and "Trento"), "dislikes a city" ("Milano", "Palermo" and "Lecce") and "likes low quantities" implemented in the given actual route four trips correctly with only modified merchandise and trip similarity scores of 0.8, 0.95, 0.75 and 0.8. But he also added a trip to "Trento" and cut the trip to "Palermo", what leads to two new trip similarity scores of zero. All in all, our similarity score is the mean of all these trip similarity scores:

route similarity score =

$$= \frac{1}{6} * (0.8 + 0.95 + 0.75 + 0.8 + 0 + 0) = 0.55 \quad (8)$$

It's noteworthy that the method employed to compute similarity scores lacks the capability to identify similarities in merchandise between trips with different destination cities. Our drivers do not possess an attribute such as "change the city" yet they manage to carry the correct merchandise of the standard route. Since this could be a realistic attribute it would make sense to address this limitation. Introducing a mechanism like Minhashing, as implemented in our solution for problem 2, would enable the identification of similarities even in cases involving different destination cities. However, evaluating a method with the method itself can not give new information. Furthermore, this is a method based on the method

that is used to create the data. Meaning it can only measure the difference for attributes drivers have.

4.3 Results

4.3.1 Evaluation of solution for problem 1. As described in the data generation section, we added some trends into the behaviour of the drivers. Given our recommended standard routes as output of our first algorithm we can have a deeper look into them now and try to identify these trends. Furthermore, we will analyse the appearance of all the liked/disliked cities and products by the drivers compared to their appearance in the original standard routes, which were given as input data to create our actual routes.

The tables in Figure 3 showed us, the cities "Trento", "Padova" and "Rome" are preferred, while "Milano", "Lecce" and "Palermo" are omitted by certain drivers. To assess whether our algorithm effectively captured driver priorities, we conducted an evaluation by counting the occurrences of these cities in the recommended standard routes. This count was then compared to their frequency in the original standard routes. To ensure a fair comparison, we standardized the total number of cities in the recommended standard routes to match the scale of the original standard routes. For instance, if the recommended standard routes have less routes or trips in total, it is very likely that most of the cities appear less often. To address this discrepancy, we applied a scaling factor to the number of appearances for each city in the recommended standard routes:

$$factor = \frac{\text{total number of cities in original standard routes}}{\text{total number of cities in recommended standard routes}} \tag{9}$$

This leads to comparable counts for each city and the percentage change in appearances from original to recommended standard routes can be computed. This evaluation aims to ascertain the algorithm's ability to discern attributes such as liking or disliking a city and its proficiency in identifying these cities. Consequently, a city that is favored should exhibit a significantly higher frequency in the recommended standard routes compared to a disliked one. The outcomes for the six cities mentioned earlier are visually represented in the following figure:

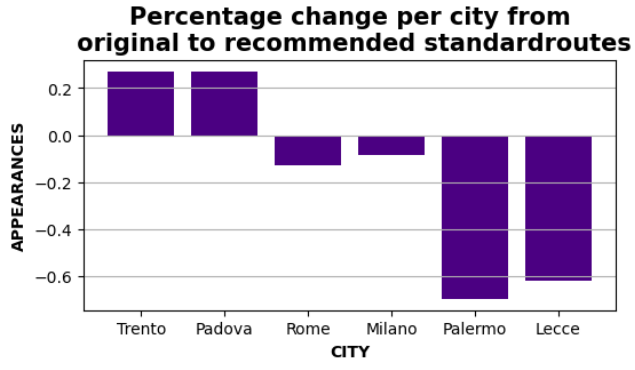

Figure 5: percentage change in appearance of liked/disliked cities

The figure illustrates a marginal increase in appearances for the favored cities, "Trento" and "Padova" within the recommended standard routes. Conversely, there is a more pronounced decrease in occurrences for the disliked cities, "Palermo" and "Lecce." It is evident that the algorithm faced challenges in accurately identifying the preferences for "Rome" and "Milano." This is possibly caused to the limited input, as only one driver expressed a preference for or against these cities, making it insufficient for the algorithm to robustly recognize these preferences. Nevertheless, the substantial decrease in appearances for "Palermo" and "Lecce" coupled with the generally correct trend of increase or decrease, is a positive indication. It suggests that the algorithm has the capability to effectively handle attributes related to city preferences.

The same procedure can be applied to evaluate liked and disliked products. Notably, "Honey", "Chocolate", "Beer" and "Cheese" are favored, while "Water" and "Milk" are omitted by specific drivers. The figure below depicts the change in occurrence for these products in the recommended standard routes.

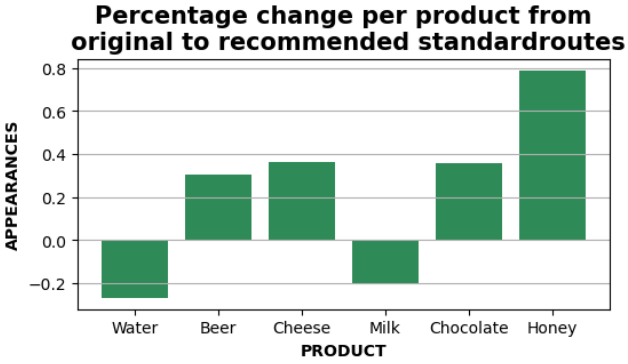

Figure 6: percentage change in appearance of liked/disliked products

This figure further highlights the algorithm's proficiency in discerning the preferences of individual drivers. Particularly noteworthy is the 0.8 increase in the occurrence of "Honey" which stands out prominently, while the occurrences of all other products are adjusted in the correct direction.

Based on the assessment outcomes for specific cities and products, we can now delve into evaluating the algorithm's ability to discern broader trends among our drivers. Beginning with the observation that four out of 10 drivers prefer shorter routes, as opposed to the two who favor longer routes, we count the occurrences for all numbers of trips in a route for the original and recommended standard routes. Since the number of trips in the data generation for the original standard routes is randomly chosen, one would expect their distribution to be approximately equal, whereas the recommended standard routes should ideally align with the trend of the drivers.

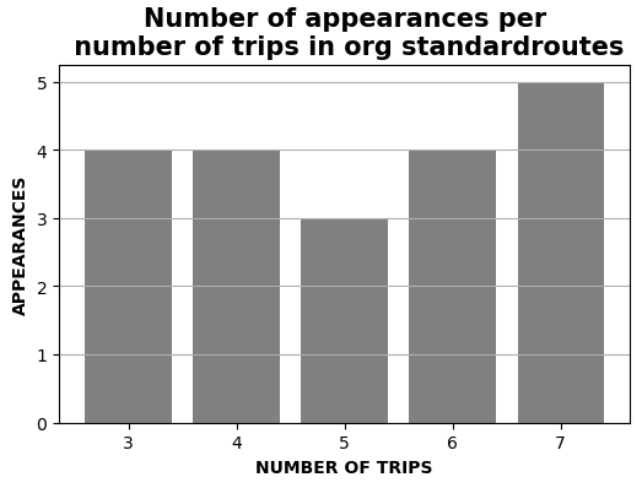

Figure 7: appearance of number of trips per route in original standard routes

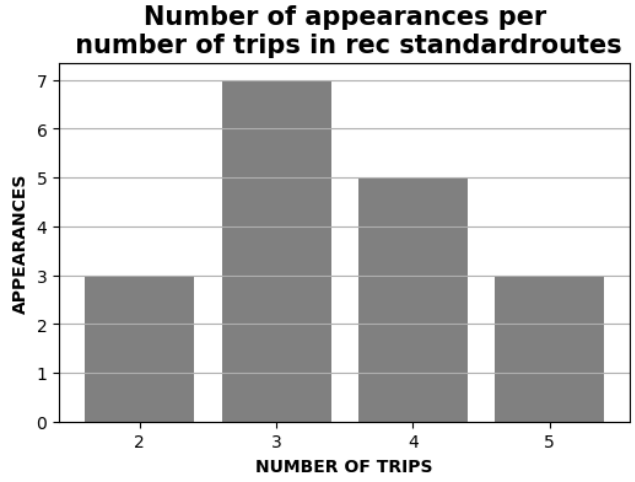

Figure 8: appearance of number of trips per route in recommended standard routes

Figure 7 illustrates the anticipated distribution of occurrences based on the number of trips in the original standard routes, while Figure 8 portrays the desired outcome for the recommended ones. Trips with length of six and seven aren't even considered in the recommended standard routes and the most frequent route length is three trips with seven appearances. This allows us to claim, that the algorithm successfully identified this trend among the drivers. Notably, in our data generation process, standard routes have a minimum length of three trips. However, due to the influence of drivers with the attribute "dislike a city" being able to eliminate disliked cities from a short route, some actual routes may consist of only one or two trips. Consequently, the recommended standard routes also possibly suggest routes with less than three trips (same holds for the maximum of seven trips per route).

The same comparison can be made concerning the drivers' tendency to include many different products in a trip. In our case, only one driver favors few products, while three prefer to load at least six products per trip in a route. The occurrences per number of products in the trips of original and recommended routes are visualized in Figures 9 and 10:

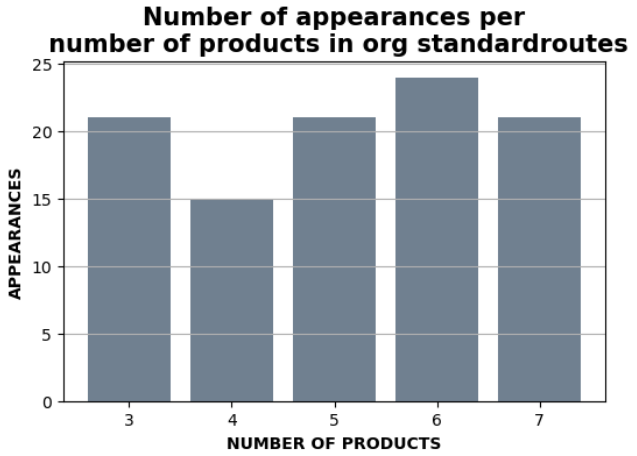

Figure 9: appearance of number of products per trip in original standard routes

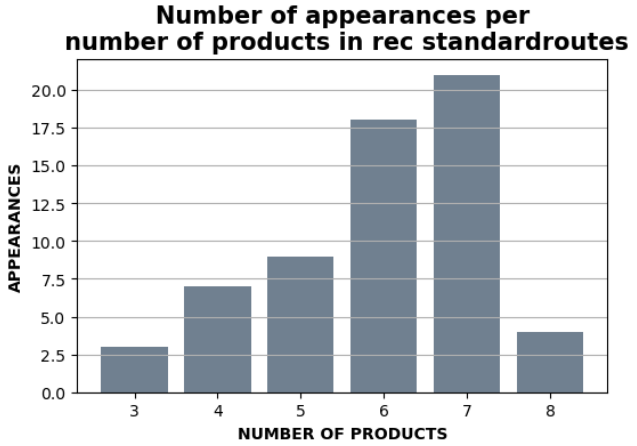

Figure 10: appearance of number of products per trip in recommended standard routes

Once again, the distribution in the original dataset appears to be roughly equal, whereas the recommended standard routes evidently align with the drivers' tendency to carry many products. Notably, the recommended standard routes may include trips with more than seven products, as previously mentioned, but in this case influenced by the attribute "likes a product".

The third trend implemented in among all drivers is their tendency to prefer smaller quantities. Specifically, three drivers favor quantities of 15 and below, while one does the exact opposite. The occurrences per quantity of a product in the original and recommended routes are illustrated in Figures 11 and 12:

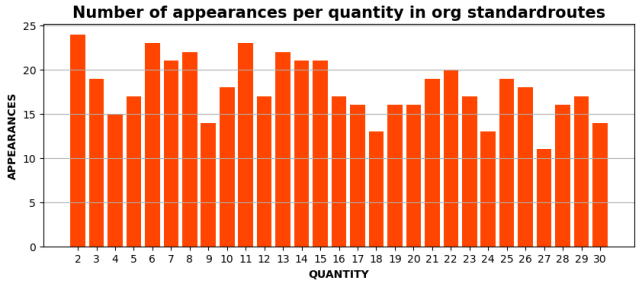

Figure 11: appearance of quantities in original standard routes

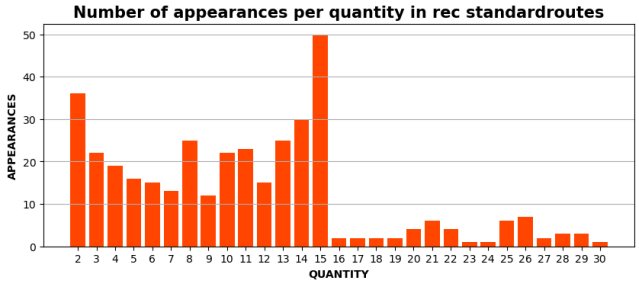

Figure 12: appearance of quantities in recommended standard routes

Furthermore, Figure 12 exhibits the desired results, indicating a significantly decline in occurrence of quantities in the range from 16 to 30. The notable frequency of the quantity 15 is likely caused to the effect that it is the only quantity acceptable for both attributes—liking high or low quantities.

Finally, we made use of the previously introduced similarity score function, in order to assess the efficacy of our algorithm in mitigating discrepancies between standard and actual routes. Therefore, we replicated the generation of $10E4$ actual routes, this time utilizing our recommended standard routes as input, while maintaining the same drivers and hyperparameters for the drivers' features. To conduct a comparative analysis, we initially computed the similarity score for each original actual route with its corresponding standard route. Subsequently, we performed the same calculation for the new set of actual routes generated using our recommended standard routes as origin. The original set of actual routes yielded an overall mean similarity score of 0.6074. Remarkably, our recommended set of standard routes demonstrated a notable improvement, reducing the discrepancy with an equivalent number of standard routes by approximately 6%, resulting in an overall similarity score of 0.6668. Given the huge dataset of a $10E4$ actual routes, underlines the significance of our findings. It can be confidently asserted that our recommended set of standard routes not only outperformed the original set but also effectively narrowed the gap between standard and actual routes.

Upon obtaining similarity scores for all actual routes, we segmented the data according to each driver's set of routes. Subsequently, we calculated the mean similarity score for each driver, separately analyzing their performance in both the original dataset and the dataset generated with our recommended route origins. This approach provides a comprehensive overview of individual driver performance in comparison to their previous records. The following graph presents the mean similarity scores for a specific driver for both sets of actual routes:

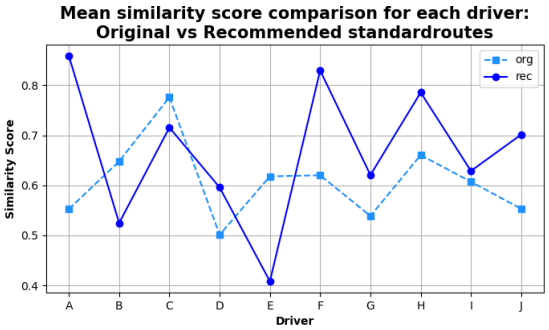

Figure 13: mean similarity score per driver with recommended standard routes

The observation that most drivers have higher similarity scores with the recommended set is not very surprising compared to their performance in the original set, given the overall improvement in similarity scores for the recommended dataset. This is particularly evident among drivers who followed the trends that were introduced in the data generation subchapter. For instance, drivers "A", "F", "G", and "J", all identified with the attribute "prefers short routes", outscore their previous performance. While "A" and "F" achieve remarkable high performances with the recommended set since they align with two additional trends — "A" with the preference for low quantities and "F" with a preference for a diverse range of products. Furthermore, driver "H" demonstrates a substantial performance increase, attributable to a preference for the trend to many products, the previously mentioned significant reduction in disliked cities "Palermo" and "Lecce" and a decrease in his disfavored products "Water" and "Milk". It is noteworthy that even drivers with three attributes, such as the aforementioned driver "H" and driver "D" achieved performance improvements. Conversely, drivers deviating from these trends experienced a decline in performance with the recommended standard routes. The standout negative performers are represented by drivers "B" and, especially, "E". The latter, in particular, faced challenges primarily due to a departure from two trends – a preference for longer routes and trips involving only a few products. Notably, considering the substantial volume of actual routes, each driver encounters every standard route multiple times, including those that may not align with their individual preferences.

Since the drivers show controversy attributes like favoring short or long routes, it is impossible to satisfy all drivers' preferences with only one set of standard routes. This realization underscores the utility of recommending a tailored set of standard routes for each driver, aligned with their personal preferences, as elaborated in the subsequent subchapters.

4.3.2 Evaluation of solution for problem 2. To assess the efficacy of our solution in the second part, we generated a fresh set of actual routes. This involved utilizing the five recommended standard routes selected from the entire pool of original standard routes for each driver as input, resulting in the creation of $5E2$ actual routes per driver and a cumulative total of $5E3$ new actual routes. Following this, we calculated the similarity scores once again and proceeded to compare them with the initial scores obtained using all original standard routes. The mean similarity score of the original set was only marginally surpassed by approximately 3%, resulting in an overall similarity score of 0.6373. The result for the drivers' individual scores is visualized in Figure 14:

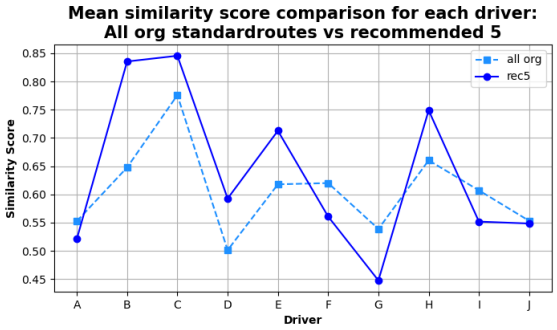

Figure 14: mean similarity score per driver with recommended 5 standard routes

Notably, drivers "A", "F", "G", and "I" characterized by the attribute "prefers short routes" experienced a noteworthy decline in performance and were unable to enhance their previous scores. This can be attributed to the absence of route length consideration in our route properties within solution two. Addressing this weakness in our algorithm would be essential to achieving improved results for drivers with this specific attribute. Conversely, drivers such as "B", "C", and "E" demonstrated an ability to increase their performance compared to their scores with the original set. This positive outcome suggests that our algorithm effectively identified similarities between these drivers' actual routes and the routes present in the original standard routes.

4.3.3 Evaluation of solution for problem 3. We applied the same method as described above to evaluate the effectiveness of our solution for the third part. In this instance, we provided the drivers with their perfect route as input, generating an additional set of actual routes totaling 1000. By computing similarity scores, we were able to once again compare the mean score and the drivers' individual performance with their perfect route to their scores derived from the original standard routes. This time, the original set of standard routes, with a mean similarity score of 0.6074, was decisively surpassed by the set of perfect routes, which achieved an impressive score of 0.9511. The ensuing graphs visually depict the results of the individual drivers' performances:

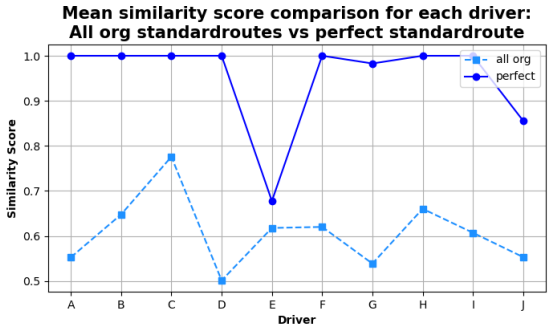

Figure 15: mean similarity score per driver with perfect route

The graph highlights the robust performance of the algorithm, with eight out of 10 drivers achieving a mean similarity score of over 98%. While this may seem somewhat unrealistic, it is a logical outcome considering that, for instance, a driver favoring short routes may not seek to alter a perfect route with only three trips. However, two drivers, namely "E" and "J" did not achieve such an exceptional result. The challenge in the perfect route for driver "E" is caused from its length, being one trip too short. Given the driver's preference for longer routes with a minimum of six trips, the perfect route, comprising only five trips. As for driver "J" while the perfect route aligned with the attribute favoring low quantities, the second attribute, liking the products "Honey" and "Chocolate" presented a slight contradiction. Given the fact that he preferred loading a minimum quantity of 20 for these favored products. This was obviously a poorly chosen feature combination for the driver, which became visible in the results of its perfect route. Despite this, both drivers still managed to surpass their original scores, and it is clear that the algorithm was capable to accomplish the given task. Remarkable is also that the two drivers "D" and "H" with three different attributes achieved a mean score of 100%.

4.4 Runtime solutions

An important parameter to evaluate is how the compution time needed for the solution grows when increasing the amount of input routes. This is done for the "finding standard routes" algorithm and the "sorting routes" algorithm.

This is tested by taking a limited amount of samples out of the file with actual routes. This is done in such a way that the "diversity" of the routes is kept intact. This is mentioned because the way the data is generated the first routes in the file all follow the same standard route meaning that their diversity is limited to diversions from the standard route. Limiting the amount of routes to the first X routes in the file would cause the diversity to change for different limits, which is not wanted for equal comparison. Therefore X random samples without replacement are taken from the actual routes file, these are used. It should also be mentioned there is some time added that stays constant, this is needed for loading the data. This

is the case because the data needs to be filtered for a specific driver and after that for a specific limitation of routes meaning that this time is not influenced by the data limit. This does not influence the growth of the curve it only adds a constant.

First, the time taking by the "finding standard routes" algorithm is measured. This is done by limiting the data of one file to first $10E3$, and then taking steps of $10E3$ and stopping at $10E4$. This is done 3 times to be able to measure the mean and the standard deviation. The time taken by the algorithm is not expected to be deterministic because K-means starts with a random sample meaning that the time to get to the result can change. Furthermore, because the file of $10E4$ actual routes is randomly sampled the routes may also be different, possibly causing there to be some change in spent time. The time is measured by using the "time" package of python. The measurements were done with a computer with the following (only the most important) specifications:

-Intel® Core™ i7-10750H CPU @ 2.60GHz × 12
-15,3 GiB DDR4 RAM
-NV137 / Mesa Intel® UHD Graphics (CML GT2)

The "actual20.json" file was used for this measurement. The result for the "finding standard routes" can be seen in the figure below:

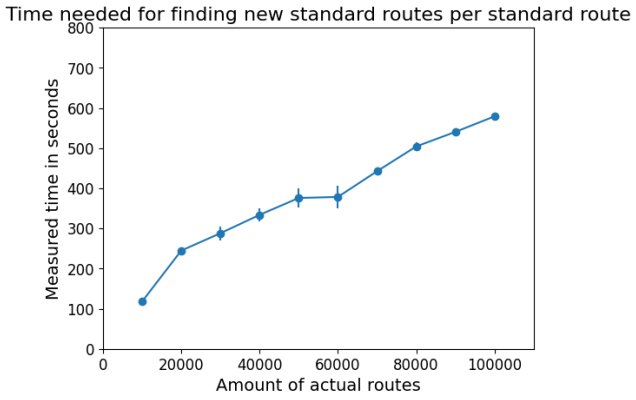

Figure 16: Graph of measured time spent on finding new standard routes with on the y-axis with on the x-axis amount of actual routes

In the figure it can be seen that it grows very fast when going from $10E3$ to $20E3$ routes. After that it shows somewhat linear behaviour. This graph can be explained by the way the solution is programmed. When running the algorithm it was noticed that the slowest part of the algorithm is the clustering part. This is mainly due to the part where the K is found for the clustering. The amount of samples and expected clusters is only limited after the datasize within a specific cluster is above 100. The probability that this happens increases when the amount of routes increases. Which it does a bit; in the beginning it shows a faster growing part after which it seems to be fairly linear. Which is to be expected because $\sqrt{x^2} = x$ and finding the K-means is expected to show quadratic behaviour while the samples are limited to the square root after some point. However, when the amount of actual routes is increased much beyond $10E4$ while keeping the same data diversity the K-means algorithm is expected to start ruling the growth. This would mean that the growth would start to show quadratic behaviour. This is expected because the growth of the samples that are taken to find the K grows with a minimal amount and the K-means is expected to grow quadratically which in the end would result in a quadratic behaviour.

The same test is performed for the algorithm that sorts the routes for every driver. The limitation of the data is done the same way. However, because each driver only drives a part of the routes in the "actual20.json" file we start at $10E2$ actual routes for every driver, resulting in $10E3$ actual routes (because there are 10 drivers). Then, each time $10E2$ actual routes per driver are added until the limit of $10E3$ is reached resulting in roughly $10E4$ actual routes (roughly because each driver did not drive exactly $10E3$ routes). The routes that need to be sorted are the standard routes in the file "standard20.json". The same computer is used and the same method to measure the time. It is again done 3 times because even though the time is not expected to be different each time, other processes on the computer might influence the measurements. These measurements that were done result in the following graph:

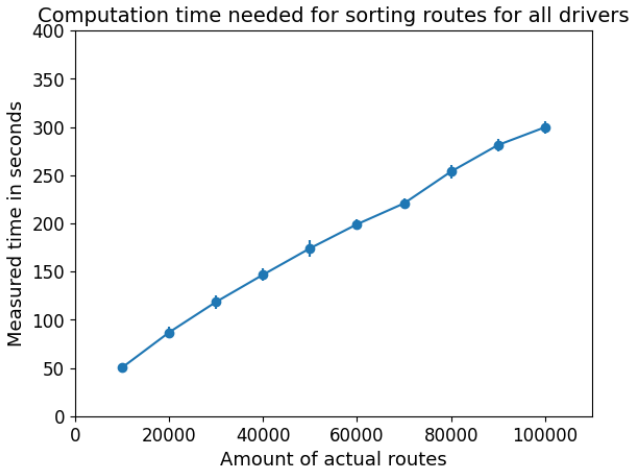

Figure 17: Graph of measured time spent on sorting routes on the y-axis with on the x-axis amount of actual routes

In this graph, linear behaviour can be seen. This is also expected because for every time a route is added the same operation needs to be done on that route. It needs to be converted into a signature and compared to some other, constant routes. This tasks stays the same every time.

The last solution is the two solutions added together meaning that the growth of this solution is determined by the ruling one. We expect that to be the first one. However, we do not have evidence so it can only be suspected. This is not measured because the addition of measured time for the two algorithms would roughly give the same result.

5 CONCLUSION

In conclusion, three algorithms were successfully created for the three problems: propose better standard routes; for each driver, sort the existing standard routes depending on his tastes; propose an ideal standard route for every driver. The first solution uses a combination of clustering and finding frequent maximal sequences, the second solution uses Minhashing signatures to estimate Jaccard similarity and rank the routes based on their average score. The last one uses the two solutions to generate the perfect route.

When evaluating the results, the three trends that were introduced were correctly recognized by the algorithm. Ranking the routes seemed to have a problem because it was not able to take length of the routes into account. The algorithm to create perfect route was able to spot almost all preferences and give the accurate route to satisfy the driver's wishes.

REFERENCES

[1] [n. d.]. K-means Cluster Analysis · UC Business Analytics R Programming Guide. https://uc-r.github.io/kmeans_clustering#gap. (Accessed on 01/14/2024).
[2] [n. d.]. sklearn.metrics.silhouette_score — scikit-learn 1.3.2 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html. (Accessed on 01/14/2024).
[3] [n. d.]. sklearn.preprocessing.MinMaxScaler — scikit-learn 1.3.2 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html. (Accessed on 01/14/2024).
[4] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. [n. d.]. Mining of Massive Datasets. http://infolab.stanford.edu/~ullman/mmds/book.pdf. (Accessed on 01/14/2024).