On Recommendation Systems in Logistics

Redi Vreto
mat.248027
University of Trento
Trento, Italy
redi.vreto@studenti.unitn.it

Pietro Ferrari
mat.239952
University of Trento
Trento, Italy
pietro.ferrari@studenti.unitn.it

Riccardo Peiretti
mat.248323
University of Trento
Trento, Italy
riccardo.peiretti@studenti.unitn.it

1 INTRODUCTION

Nowadays, with the advent of the internet, recommendation systems have started playing a pivotal role in our society. The ubiquity of these systems cannot be understated; they are paramount for making systems like the Google search engine, YouTube videos, social media posts, Netflix movies, Amazons products, Spotify music, and many more, work seamlessly.

The process of recommending items in the physical world is relatively straightforward, typically involving the display of the most popular items while overlooking less popular ones. On the other hand, online organizations, despite having the ability to exhibit their entire product range, must contend with the challenge of not being able to present everything at once. As a result, they rely on sophisticated recommendation systems that tailor suggestions based on the user they are interacting with, rather than relying solely on popular choices.

In this paper, various recommendation systems within the context of logistics applications will be discussed. These recommendations will be tailored for both individual users and the entire set of drivers, in similar fashion to online and physical organizations. The application of recommendation systems in this context is crucial as it benefits both the company and its employees. In fact, employees can align their choices more closely with their needs, while companies gain insights into the rationale behind their drivers' decisions.

Applying traditional programming techniques to address this challenge is possible but will inevitably yield sub-optimal results. Therefore, in this context, data mining techniques such as Locality Sensitive Hashing (LSH) and recommendation systems, among others, were employed. The results speak for themselves; they consistently outperform a baseline solution created using basic processes. In one out of three cases, significantly improved results were attained, both in terms of the quality of the proposed solution and the time required to achieve it.

Clearly the most challenging aspect of the paper is going to be dealing with abstract data when data mining techniques usually interface with low level data structures such as sets and vectors. Additionally, there will be numerous difficult decisions to navigate regarding hyperparameter estimation and algorithm selection in this context.

In summary, this paper will first introduce a recommendation system that suggests a set of routes for all drivers based on clustering and dimensionality reduction. Secondly, we will recommend five routes to each driver using more traditional recommendation systems, including collaborative filtering and content-based filtering, both leveraging LSH. Finally, mirroring online recommendations, we will propose a single route to each driver.

2 RELATED WORK

In this section, for the reader's convenience, we provide a summary of the main techniques and technologies adopted for this project.

Distance metric. Let d be any distance metric, then, d is a *true distance metric* if and only if it satisfies all of the following axioms:

- The distance from a point to itself is zero: $d(a, a) = 0$;
- The distance between two distinct points is always positive: $d(a, b) \neq 0$;
- The distance from a to b is always the same as the distance from b to a: $d(a, b) = d(b, a)$;
- The triangular inequality holds: $d(a, c) \leq d(a, b) + d(b, c)$.

Cosine similarity. The cosine similarity is a similarity measure applied to two n-dimensional vectors. It is defined as follows:

$$cos(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{||\vec{a}|| \cdot ||\vec{b}||} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \cdot \sqrt{\sum_{i=1}^{n} b_i^2}}$$

We can exploit the above similarity measure to obtain a distance metric, which we call the *cosine distance*:

$$\text{cosine distance}(\vec{a}, \vec{b}) = 1 - cos(\vec{a}, \vec{b})$$

While this is a good distance metric in many applications, we should note that it does not meet the criteria to be considered a true distance metric as it does not satisfy the triangular inequality.

Recommendation systems. Recommendation systems come in different shapes and sizes, and they can be classified into three main groups:

- *Content-based systems* (CB) analyze properties of the items recommended. In short, these systems try to extrapolate information from the user's actions, build a user profile, and use it to understand what the user likes and does not like when making recommendations [7];
- *Collaborative filtering systems* (CF) recommend items based on similarity measures between users and/or items. For instance, when predicting the rating of a film I for a user U, one approach is to identify the n users most similar to U and use the average rating given by these users as a prediction. Alternatively, we can find the n most similar films to I that user U has already rated and compute the average [7];
- *Hybrid systems* can be formed by combining the previous two techniques. Different strategies may be employed to integrate the two approaches based on the specific problem at hand [2].

We use a *utility matrix* to represent the necessary information for a recommendation system. The utility matrix, often denoted as U, is a mathematical representation used in recommendation systems where the rows represent users, the columns represent items, and the entries are the rating given by the users. If we have n users and m items the utility matrix would look like this:

$$U = \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,m} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n,1} & u_{n,2} & \cdots & u_{n,m} \end{bmatrix}$$

Here, $u_{i,j}$ represents the rating of user i for the item j.

Similarity search. In various segments of this project, we leverage fast similarity search techniques. Faiss, a library developed by Facebook AI, facilitates efficient similarity search. Thus, when presented with a set of vectors, we can index them using Faiss. Subsequently, by using another vector (the query vector), we conduct a search within the index to identify the most similar vectors [6].

Clustering. Clustering is a process within unsupervised machine learning that involves grouping a set of data points into subsets. Formally, Given a set of n points $X = \{x_1, x_2, ..., x_n\}$ and a set of k clusters $C = \{c_1, c_2, ..., c_k\}$, the goal is to partition each point $x \in X$ in a cluster $c \in C$, i.e. find a function $f : X \rightarrow C$.

K-means. K-means is a clustering algorithm that only works with euclidean distance and circular clusters [11]. Choosing a good value for k in the K-means algorithm is a demanding task that is very easy to get wrong. Ideally, we seek a value for k such that any further increase does not result in a significant decrease in the overall distances from the centroids. In other words, when graphing k against the sum of squared distances from the closest centroid, our goal is to pinpoint the *elbow*. A technique to do just that is presented in [10]: for any continuous function f, there exists a function K_f that defines the amount of curvature of f at any point:

$$K_f(x) = \frac{f''(x)}{(1 + f'(x)^2)^{1.5}}$$

Thus, to determine the optimal elbow, we need to identify the point at which $K_f(x)$ is maximized, essentially finding the point where the derivative is zero.

Dimensionality reduction. Dimensionality reduction is the process of mapping high-dimensional data into lower-dimensional data. Formally, let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of n points in $\mathbb{R}^n$. The goal is to find a mapping function $f : \mathbb{R}^n \rightarrow \mathbb{R}^k$, where $k \leq n$, such that the transformed points $Y = \{f(x_1), f(x_2), \ldots, f(x_n)\}$ preserve as much information as possible.

A prominent technique to accomplish that involves exploiting Singular Value Decomposition (SVD). The process behind SVD is quite involved. In short, it consists of transforming a matrix $A \in \mathbb{R}^{n\times m}$ into three matrices $U \in \mathbb{R}^{n\times n}$, $\Sigma \in \mathbb{R}^{n\times m}$, and $V \in \mathbb{R}^{m\times m}$, such that $A = U\Sigma V^T$ [12].

3 SOLUTION

3.1 Preliminary definitions

In this paper, we are working with a logistics company. The company has a specific number of drivers, operates in a defined set of cities, and delivers a particular set of products. The company follows a set of standard routes that it instructs its drivers to take and keeps track of the actual routes each driver follows, along with the corresponding delivered merchandise, when working on a standard route. The resulting route is referred to as an actual route and is associated with the driver who completed it.

Definition 3.1. Let D be the set of all drivers, C, the set of all cities, and P the set of all products.

Definition 3.2 (Trip). A trip represents the movement from one city to another, all while carrying merchandise. Formally it is a tuple $C \times C \times M$, where M is the carried merchandise, i.e. the set of products such that $M = \{p_1, p_2, \ldots, p_m\}$ where, for each $p \in M$, $p \in P \times \mathbb{N}$, where $P \times \mathbb{N}$ indicates the product carried and the amount.

For example, if we move from Milan to Rome whilst carrying 3 beers, 2 cokes and 3 bottles of water the trip tuple will be equal to (Milan, Rome, ((Beer, 3), (Coke, 2), (Water, 3)).

Definition 3.3 (Route). A route is a set of trips, i.e. a set $R = \{t_1, t_2, \ldots, t_n\}$ where each $t_i \in T$

For instance, if we have the same trip we had earlier and we add the trip (Rome, Naples, (3, Coke)) we would obtain the valid route

(Milan, Rome, ((Beer, 3), (Coke, 2), (Water, 3))
(Rome, Naples, (Coke, 3)).

Note that even just one trip alone is a valid route. On the other hand, a route where the end cities do not coincide, is not valid. For example, the route

(Milan, Rome, ((Beer, 3), (Coke, 2), (Water, 3))
(Bari, Naples, (Coke, 3)).

Is not valid because in the first trip we ended up in Rome and in the second one we are in Bari.

Henceforth, we will refer to the set of all possible trips as T, and the set of all possible routes as R.

Definition 3.4 (Standard routes). A set of routes $S = \{r_1, r_2, \ldots, r_n\}$ where each $r_i \in R$.

Definition 3.5 (Actual routes). A set of tuples $A = \{a_1, a_2, \cdots a_n\}$ where each $a_i \in R \times D$.

3.2 Part one

3.2.1 Problem statement. Input: The set of standard routes S and The set of actual routes A. Output: A set of recommended standard routes such that the overall divergence, i.e. how different each route and actual route will be, is minimal.

3.2.2 Representing routes as vectors. Given the upcoming tasks, it would be highly convenient if our routes could be represented as vectors. To achieve this, we define the features to be extracted from a route as follows: add an entry to the vector for each element in $C \times P \times DIRECTION$, where $DIRECTION$ is an enum containing the entries TO and $FROM$. In other words, we are trying to understand how much each concept is present in the route. If a route includes the trip (Rome, Milan (Coke, 100)), then the features (Rome, Coke, FROM) and (Milan, Coke, TO) are likely to have high values. If a route never goes to Rome, then all the features containing Rome will be set to zero, indicating that that particular feature is not present whatsoever. If a route delivers a particular element to the same city twice, then the value of the feature is the sum of the quantities of the two deliveries.

Algorithm 1 Route to vector

```
Input: $r \in R$                          ▷ A route from the set of routes
1: $v \longleftarrow$ new float $[2 \cdot |C| \cdot |P|]$
2: for $i \longleftarrow 1$ to $2 \cdot |C| \cdot |P|$ do
3:     $v[i] \longleftarrow 0.0$
4: for each $t \in r$ do
5:     for each $(p, c) \in t.merchandise$ do
6:         $v[r.\text{first}, p, \text{FROM}] \longleftarrow v[r.\text{first}, p, \text{FROM}] + c$
7:         $v[r.\text{second}, p, \text{TO}] \longleftarrow v[r.\text{second}, p, \text{TO}] + c$
8: return $v$
```

As reported in the algorithm 1, the algorithm takes as input a route from the set of routes R and returns a vector of features. In lines one to four, a new vector is created, and all its entries are initialized to zero. The reason for having $(2 \cdot |C| \cdot |P|)$ entries is that two features are created for each pair of cities and products—one indicating how much this route contains the feature *bring product to city*, and the other *take away product from city*.

In line four, there is an iteration through each trip in the route. Line five itemizes each product and its count from the merchandise of the trip. Finally, in lines six and seven, the count of the product is added to the respective feature. For example, if the trip goes from Rome to Milan with 3 cokes, then the feature (Rome, Coke, From) will be incremented by 3.

As the reader might have already guessed, this conversion is by no means lossless. However, devising such a conversion would lead to extremely high-dimensional data, which would render data mining algorithms unusable. Thus, it was decided to just extrapolate the fact that a routes takes some product to and from some city. At the end of the day, the only information lost pertains to the interconnections between cities, such as the specific order of cities in a route, like Rome following Milan. Unfortunately, it is very easy to end up with high-dimensional data as the input scales up. Let's suppose we have 5000 cities and 10000 products; in that case, the number of features would be $2 \times 5000 \times 10000 = 10^8$

3.2.3 Choosing a similarity measure. At this point, we have devised a procedure to convert routes to vectors. The next step is to apply algorithm one to each route in the set of routes to have all the routes in vector form. Now, our goal is to build the utility matrix. However, before we can do that, we need to choose a similarity measure. Since we are working with vectors, we can either use cosine similarity or try working with the Euclidean distance by treating the vector's components as an n-dimensional point. The way we can use the Euclidean distance as a similarity metric is by subtracting each value from the maximum value overall. For example, suppose the maximum distance is 100, and the distance between two examples is 30; in this case, the similarity would be $100 - 30 = 70$.

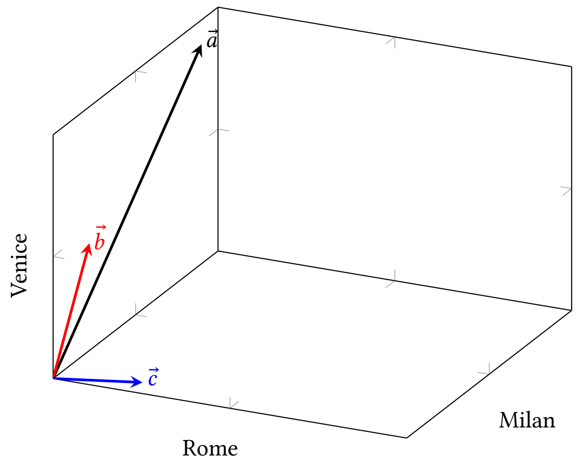


Figure 1: Difference between euclidean and cosine distance

As we are about to illustrate from figure 1, both methods have their shortcomings. Let's consider a scenario with only three features: Milan, Rome, and Venice. The Euclidean distance between $\vec{b}$ and $\vec{c}$ is small, while the distance between $\vec{b}$ and $\vec{a}$ is much larger. However, this does not make sense because $\vec{b}$ and $\vec{c}$ have no features in common, whereas $\vec{b}$ and $\vec{a}$ do share some values.

On the other hand, if we use cosine similarity, $\vec{b}$ and $\vec{a}$ appear close, whereas $\vec{b}$ and $\vec{c}$ are almost orthogonal, indicating high dissimilarity. It is true that $\vec{b}$ and $\vec{a}$ share some features, but their magnitudes are significantly different. So, should $\vec{b}$ and $\vec{c}$ be considered closer instead? Well, this depends on the specific problem at hand.

As mentioned earlier, we are dealing with highly dimensional data, and consequently, we will encounter the *curse of dimensionality*. This phenomenon implies that as the number of features increases, the feature space grows exponentially. The problematic consequence is that all points will seem to be very distant from each other [1], rendering data mining algorithms less effective [14]. In addition to that, we expect our data to be very sparse, because each driver will only work on a small number of cities in a real-world scenario, therefore, the curse of dimensionality effects will be even greater. Cosine similarity proves to be less affected by the curse of dimensionality because, as long as the direction of two vectors is approximately the same, a reasonably high similarity value is obtained. In contrast, Euclidean distance lacks this guarantee.

In light of all these facts it was decided to settle for the cosine similarity.

3.2.4 Building the utility matrix. At this point, we have all the routes in input in vector form and we have chosen a similarity measure to quantify how similar two routes are. Since we are working with cosine similarity, the entries in the utility matrix will be ranging from -1 to 1.

Before we proceed any further, let us take a moment to recall what a utility matrix is. The utility matrix, often denoted as U, is a mathematical representation used in recommendation systems

[1]in terms of euclidean distance

where the rows represent users, the columns represent items, and the entries are the rating given by the users. In our case the users are the drivers D, the items are the standard routes S, and the rating is a function from the set of drivers D and the set of standard routes S to the range $[-1, 1]$, i.e. $f : S \times D \rightarrow [-1, 1]$. To sum up, $u_{i,j} = r$ means that the driver i has rated the route j with r.

We have no such thing as a rating in input though. So how do we determine how much a driver likes a standard route? Well, there are many ways to fill the utility matrix, the most common one is to fill it based on the driver's behavior. In our scenario, when we assign a standard route $s \in S$ to a driver $d \in D$, to estimate how much the driver *liked* that route, we calculate the cosine similarity between the standard route $s \in S$ and the actual route $a \in A$ the driver followed. This is a great way to extract the driver's preference without actually knowing them, and here's why:

- If the cosine similarity is close to 1, it means that the drivers likes what was in the standard route because they stuck to the plan. Geometrically, the two vectors will point roughly in the same direction;
- As the similarity approaches -1, the drivers likes less and less what was in the standard route because they do something different. Geometrically, the two vectors will point in opposite directions.

Now that we understand how to fill a single entry in the utility matrix, the remaining task is to replicate this process for each pair of standard routes and actual routes in input:

Algorithm 2 Build utility matrix

```
Input: $S, A$                          ▷ The standard and actual routes
1: $U \leftarrow$ new float $[|D|][|S|]$
2: for $i \leftarrow 1$ to $|D|$ do
3:     for $j \leftarrow 1$ to $|S|$ do
4:         $U[i][j] \leftarrow$ None
5: for each $(s \in S, a \in A)$ do
6:     $U[a.driver][s] \leftarrow \cos(a, s)$      ▷ a.driver is the driver who performed the actual route
7: return $U$
```

The first four lines initialize the utility matrix with None entries, which is a special value indicating that there is no rating. Then, lines five and six repeat the process described earlier for all pairs of valid standard routes and actual routes. A pair of a standard route $s \in S$ and an actual route $a \in A$ is valid if and only if the standard route s in the actual route a is exactly the same.

3.2.5 *Building the user profiles.*

Now that we have the utility matrix, we can use it to create a profile for each driver. A driver profile is a vector, used in recommendation systems, with the same components of the route vector that describes the driver's preferences. The end goal is to have a vector of features where a value close to +1 indicates that that feature tends to appear in routes the user likes, and a value close to -1 tends to appear in routes the user does not like. Let us go through an example to understand how we are going to create the driver's profiles. Imagine we want to determine the value for the feature Milan for a driver $d \in D$. We consider all the routes the driver has rated that have a non-zero value for Milan and just take the average. If those routes have rating equal to 0.3, 0.4 and 0.5 then we set the feature Milan to $\frac{0.3+0.4+0.5}{3}$. Assuming the driver d gives an average rating 0.3, we normalize because we only want to know how much more than the average this driver likes the feature. i.e. the rating will be $\frac{(0.3-0.3)+(0.4-0.3)+(0.5-0.3)}{3} = 0.1$.

Algorithm 3 Build driver's profiles

```
Input: $U, S$                          ▷ Utility matrix and standard routes
1: $avg \leftarrow$ new float $[|D|] = 0$
2: $cnt \leftarrow$ new float $[|D|][2 \cdot |C| \cdot |P|]$
3: $V \leftarrow$ new float $[|D|][2 \cdot |C| \cdot |P|]$
4:
5: for $i \leftarrow 1$ to $|D|$ do      ▷ Calculate avg rating for each driver
6:     $avg[i] \leftarrow mean(U[i])$
7:
8: for each $d \in D$ do
9:     for each $s \in S$ do
10:        if $U[d][s] \neq$ None then
11:            for $f \leftarrow 1$ to $2 \cdot |C| \cdot |P|$ do
12:                if $s[f] \neq 0$ then
13:                    $V[i][f] \leftarrow (U[d][s] - avg[d]) * s[f]$
14:                    $cnt[i][f] \leftarrow cnt[i][f] + 1$
15:
16: for $i$ in $|D|$ do
17:     for $j$ in $2 \cdot |C| \cdot |P|$ do
18:         $V[i][j] \leftarrow V[i][j]/cnt[i][j]$
19: return $V$
```

Lines five and six compute the average rating each driver gives to the routes they have tried. In lines eight to fourteen, we perform the process described above for each driver and each standard route they performed, but also taking into account how much each feature is present. In the example above, if the vector routes containing Milan had magnitudes 1, 2, 3, then the formula would become $\frac{(0.3-0.3)*1+(0.4-0.3)*2+(0.5-0.3)*3}{3} = 0.26$. This addition is essential to assign more weight to routes that feature a particular aspect more prominently.

3.2.6 *Reducing the number of dimensions.*

As mentioned in 3.2.2, we are dealing with highly dimensional data. Hence, it would be advantageous to reduce dimensionality to improve the algorithms efficiency and eliminate redundant features. To achieve this, we will leverage singular value decomposition. However, we won't be implementing the algorithm from scratch, as there is already a state-of-the-art implementation available [1], [9].

We have a very big decision to make though: How may dimensions should we reduce to? If we decrease the dimensionality too much we will lose too much information, whereas if we keep too many dimensions our algorithm will be very slow.

Before we can discuss how many dimensions to drop, we need to define a metric for the *information loss*.

Definition 3.6 (Information loss). Cumulative information loss, in percentage, when performing dimensionality reduction on a pool of n driver's profiles. Defined as the sum of the cosine distances between the original vectors, and the remapped reduced vectors. By remapping we mean the process of transforming a k-dimensional vector to an n-dimensional vector, where $k < n$, in such a way that the $n - k$ missing dimension are set to 0.

Here is the algorithm that calculates the information loss.

Algorithm 4 Information loss

```
Input: $V, V'$    ▷ The driver's profiles and the remapped drivers profiles
1: $\Delta \leftarrow 0$
2: for each $(v \in V, v' \in V')$ do
3:     $\Delta \leftarrow \Delta + (1 - \cos(v, v'))$
4: return $\Delta/(|V| + |V'|)$
```

Here is a real example of the trade-off between information loss and dimensionality reduction where there are 20000 drivers and 1000 dimensions.

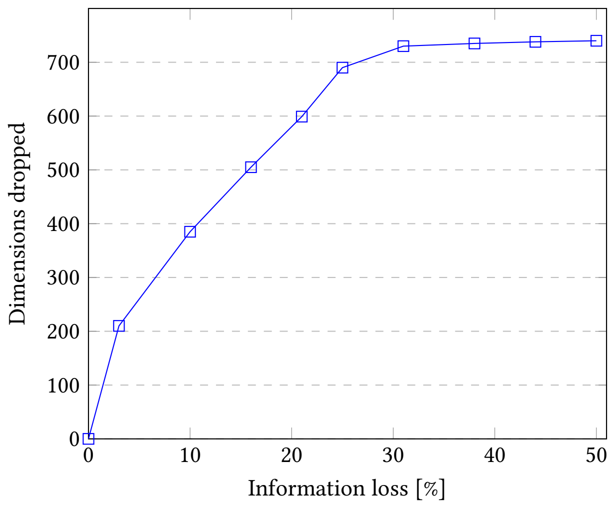


Figure 2: Trade-off between information loss and dimensionality reduction on a sparse matrix

Let us introduce the parameter l which indicates, in percentage, the maximum amount of loss we are willing to tolerate when performing dimensionality reduction. We decided to set l to 15% because on the numerous tests we ran, it never seemed to produce bad results.

The only thing left to do now is finding the number of dimension to drop in such a way that the information loss is less than l, i.e. 15%. In order to do that, we perform a binary search. The procedure, illustrated in algorithm 5, works as follows:

In line six, the function SVD performs dimensionality reduction on the given matrix and the number of dimension in input. The function svd_inverse_transform remaps the matrix to the original number of dimensions by adding entries equal to zero.

3.2.7 The long tail phenomenon. Before discussing the clustering of the driver's profiles, let us ponder on the *long tail phenomenon*. The long tail phenomenon refers to the distribution pattern where a large number of unique or niche items collectively have a substantial share of the overall distribution, even though each individual item has relatively low popularity or sales. In other words, the majority of occurrences or transactions are not concentrated on a small set of popular items (head), but are spread across a diverse array of less popular items (tail) [15]. The phenomenon is illustrated in figure 3. The items are ordered by popularity on the x-axys and the y-axys indicates how popular the corresponding item is. Physical institutions, which have a limited amount of space, usually only recommend items on the left of the horizontal lines. On the other hand, online organizations are forced to recommend items from across the distribution because they cannot show too many items at once.

Algorithm 5 Dimensionality reduction

```
Input: $V, l$    ▷ The driver's profiles and the maximum loss
1: $i \leftarrow 1$
2: $n \leftarrow 2 \cdot |C| \cdot |P|$
3: $Y \leftarrow V$
4: while $i \neq n$ do
5:     $mid \leftarrow \frac{i+n}{2}$
6:     $Y \leftarrow svd(V, mid)$
7:
8:     if information_loss(V, svd_inverse_transform(Y) $\leq l$ then
9:         $n \leftarrow mid$
10:    else
11:        $i \leftarrow mid + 1$
12:
13: return $Y$
```

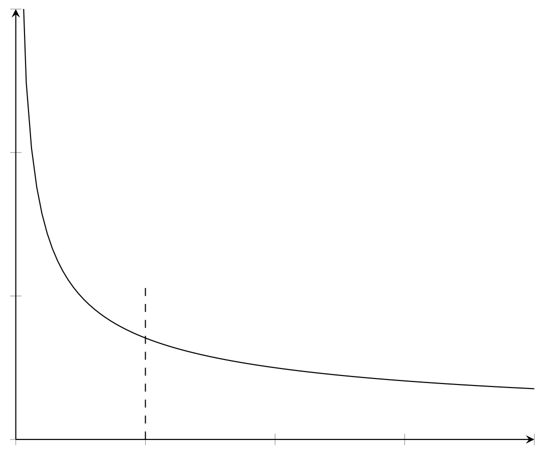


Figure 3: The long tail

It is reasonable to assume that we are dealing with such a distribution when we consider the items as the set of standard routes. This is because a significant number of drivers will be operating in big cities, while a smaller but still noteworthy portion of drivers will be working in smaller cities.

Since we cannot know beforehand to which driver will be assigned which route, we have to sample routes across the distribution, ensuring to select a proportional amount of routes to the popularity.

3.2.8 *Clustering.*

Why do we cluster. After all the work we have done so far, we have, for each driver, a profile that expresses their preferences in terms of the features we have defined. It would be great then to have a way to somehow use these preferences to make recommendations for the standard routes. In other words, we would like to create some new routes that share as many features as possible with the collection of drivers we have. With this approach, we are confident to get the best possible outcome because we are recommending routes that contain features the drivers like. Needless to say, not all drivers will like every route, however, there is a high probability that any route is somewhat preferred by a significant portion of drivers.

In Figure 4, let's consider a scenario where only two features are plotted to represent the driver's profiles on the graph. In this case, four distinct clusters emerge, with the red, orange, and blue clusters positioned closer to the head of the distribution, and the yellow cluster closer to the long tail. Our goal is to recommend a larger number of standard routes for the more significant clusters and fewer for the smaller clusters.

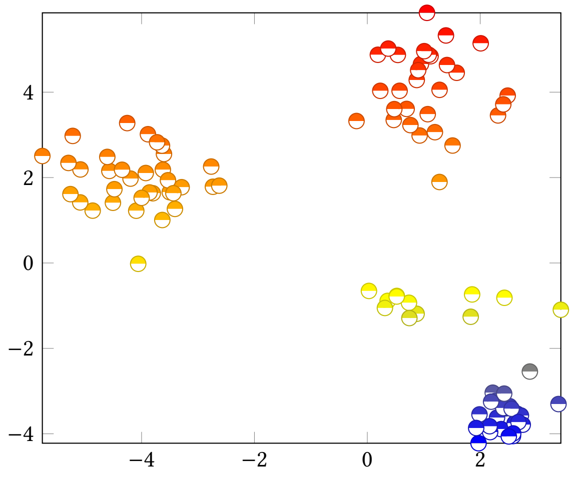


Figure 4: Example of driver's profiles distribution with two features

Choosing a clustering algorithm. Now we know that we want to find clusters, thus we need to choose a clustering algorithm. There are myriad options to choose from, each one being a trade-off between computational complexity, assumptions on the data, and quality of the results.

We expect the driver's preferences to be very divided because someone working in Milan will probably have very little preferences in common with a driver working in Rome. Among the drivers working on the same region on the other hand, there will be some features in common and others will be different. The good thing is that if all drivers do not like to go through Milan, the procedure will not recommend routes that go through there because no driver has that preference. On the contrary, if everyone likes to go through Bergamo instead, the procedure will recommend routes that pass through Bergamo. Let us now analyze the options we have.

Hierarchical clustering. Hierarchical clustering is a clustering technique where we create clusters by iteratively merging or splitting clusters until convergence. The former is called bottom-up clustering, whereas the latter is often referred to as top-down hierarchical clustering. These strategies are great in terms of the quality of the results they produce. However, they are impractical when it comes to very large amounts of data because the computational complexity is $O(n^2 \cdot logn)$[7]. To sum up, we are not going to use this strategy because the price to pay in terms of computational complexity is too high.

BFR Clustering. BFR clustering is another option that looks very good at first glance. However, this algorithm makes very strong assumptions about the data and doesn't guarantee to produce higher quality clusters. In addition to that, BFR clustering is best suited for extremely massive amounts of data that do not fit in memory [7]. Since the amount of data in input depends mainly by the number of drivers, and even a gigantic company such as Amazon has less than 20000 drivers, we decided not to consider this technique.

DBSCAN clustering. DBSCAN makes no assumptions about the data and is able to find very good clusters in general. What is more, it is robust to outliers. Finally, unlike K-means, it does not require the specification of the number of clusters. There is a major drawback though: when the density of the clusters is different, the algorithm will classify the less dense points as outliers. In other words, it won't assign a point belonging to low-density cluster to any cluster. This issue is illustrated in figure 5 where there are four clusters, two of which have a high density and two with a lower density. The red and blue clusters will not be recognized because they have a much lower density.

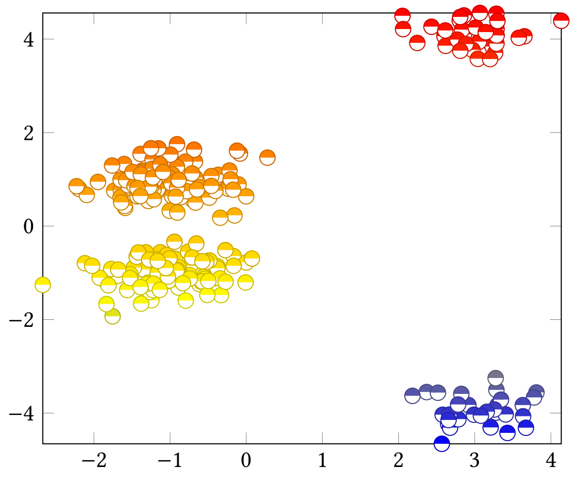


Figure 5: Four clusters with different densities

This is a huge problem for us because we expect features related to highly populated cities to have a high density of drivers, and smaller cities to have less dense agglomerations. Thus, THE DBSCAN procedure cannot be used as is. Now we could use a variation of DBSCAN, often called HDBSCAN, which takes care of the aforementioned problem. However, we still have no guarantee that the results will be any good. Due to all of these reasons, it was decided not to use DBSCAN or any of its variations.

K-means. K-means is a simple very simple procedure. Its primary advantage lies in scalability to massive datasets [7], though it assumes euclidean distance and circular clusters, which are significant limitations [7]. Since we can get around these limitations and the results are expected to be quite good, we are going to use this algorithm. The results are going to be of high quality because we are going to get centroids which come from high density areas in the feature space. In addition to that, since centroids are the average values of all its components, we will get a route which is somewhat liked by everyone in that cluster.

Clustering with K-means. The first issue we have to face is the fact that K-means only works with euclidean distance and our distance measure of choice is the cosine distance. As it turns out though, the squared euclidean distance of two normalized vectors is proportional to the cosine distance; let us see why.

The euclidean distance of two vectors $\vec{a}$ and $\vec{b}$, can be rewritten as

$$\begin{aligned} ||\vec{a} - \vec{b}||^2 &= \underbrace{\vec{a}^T\vec{a}}_{=1} + \underbrace{\vec{b}^T\vec{b}}_{=1} - 2\vec{a}^T\vec{b} \\ &= 2 - 2\vec{a}^T\vec{b} \\ &= 2\left(1 - \vec{a}^T\vec{b}\right) \end{aligned}$$

Note that $\vec{a}^T\vec{a} = 1$ and $\vec{b}^T\vec{b} = 1$ because $\vec{a}$ and $\vec{b}$ are normalized vectors. Now, the cosine distance can be rewritten as

$$\begin{aligned} 1 - \cos(\vec{a}, \vec{b}) &= 1 - \frac{\vec{a}^T\vec{b}}{||\vec{a}||||\vec{b}||} \\ &= 1 - \vec{a}^T\vec{b} \end{aligned}$$

In this case $||\vec{a}|| = ||\vec{b}|| = 1$ because the norm of normalized vector is equal to 1. $1 - \vec{a}^T\vec{b}$ appears in both equations, in fact the square of the euclidean distance is twice the cosine distance.

To sum up, we found out that the euclidean distance squared is equal to twice the cosine distance for normalized vectors, i.e. euclidean distance$^2 = 2 \cdot$ cosine distance. Therefore, all we need to do in order to solve our problems is normalizing each driver profile, i.e. divide each component of each driver profile by the length of the vector as shown in algorithm 6.

By normalizing the driver's profiles we introduce a new issue: we cannot simply take the centroids 'as it is' because they are scaled and will not reflect the original data. For example, if a centroid outputted by the K-means procedure is $\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$, the original centroid could be anything, as long as the first and second component are equal. It could be $(10, 10)$, $(0.01, 0.01)$, or $(10000, 10000)$; literally anything, as long as the first and second component are equal! Clearly this is an issue that needs to dealt with.

Algorithm 6 Normalize driver's profiles

```
Input: $V$                                         ▷ The driver's profiles
1: $len \leftarrow$ new float $[|D|]$                 ▷ Norm of each driver profile
2: for $i \leftarrow 1$ to $|D|$ do
3:     for $j \leftarrow 1$ to $2 \cdot |C| \cdot |P|$ do
4:         $len[i] \leftarrow V[i][j]^2$
5:     $len[i] \leftarrow \sqrt{len[i]}$
6:     for $j \leftarrow 2 \cdot |C| \cdot |P|$ do
7:         $V[i][j] \leftarrow V[i][j]/len[i]$
8: return $V$
```

Here is how we solve the aforementioned problem. Suppose we have a cluster c which contains a certain set of points $c = \{x_1, x_2, \ldots, x_n\}$. In order to get the centroid of the cluster with the original scale, all we have to do is take the average of the points$\{x_1, x_2, \ldots, x_n\}$ *before* they were normalized.

Finally, let us illustrate the procedure to obtain the centroids. The K-means algorithm has not been implemented because a state-of-the-art parallel implementation is provided by [9]. The idea is quite straightforward: reduce the driver's profiles, normalize the driver's profiles, apply the K-means algorithm, scale back the centroids using the previously explained strategy, and map the vectors to the original number of dimensions as explained in section 3.2.6. The approach is shown in algorithm 7:

Algorithm 7 K-means clustering

```
Input: $V$                                         ▷ The driver's profiles
1: svd ($V$)                                       ▷ Dimensionality reduction
2: normalize ($V$)
3: centroids = K-means ($V$, choose-k(V) )
4: scale-back ( centroids )
5: return inverse-transform ( centroids )
```

In line 3 we make a call to the function choose-k, which we will discuss in the next section, because we still have discusses how we choose the hyperparameter k.

The complexity of K-means is $O(TkN)$ where N is the number of points and T is the number of steps it takes the algorithm to converge. The upper limit for T is very very large, therefore the default algorithm is too slow [7]. The Sci-kit learn implementation, however, limits the number of steps to 300, making it efficient for large datasets. Moreover, it performs multiple runs with different initial points chosen using sophisticated heuristics.

3.2.9 Choosing k in K-means clustering. The hyperparameter k refers to the number of clusters we want the algorithm to create. As briefly discussed in the related work chapter, there is a procedure in [10] to pick it optimally. The idea presented essentially graphs the sum of squared distances to the closest centroid for all points against the parameter k and locates the *elbow*. In other words we keep applying the K-means algorithm for bigger and bigger values for k while keeping track of the sum of squared distances from the closest centroid at each step. Finally, if the elbow is located at e, we choose $k = e$.

For example, in figure 6 the elbow is somewhere in the neighborhood of the vertical line, thus the optimal value for k is about a little over 20.

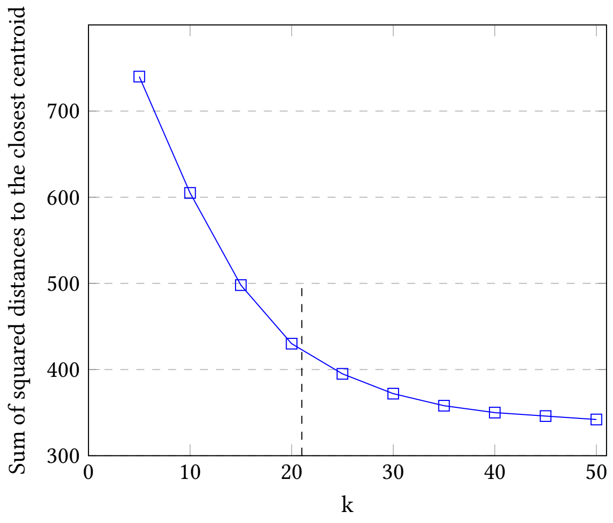


Figure 6: Elbow location

Note that this procedure involves applying the K-means clustering algorithm n times in order to check the cumulative distance from the closest centroid at each step. It was decided to set n to 50, which is a good trade-off between efficiency and precision of the solution.

3.2.10 From centroids to recommendations. After all this work, we finally have a set of centroids to work with. For the third part of this project, we will devise a function that given a user's preferences, outputs the perfect route for them, i.e. the route that when performed, yields the least amount of divergence. What we are gonna do then, is feed this function our centroids to create recommendations. Let us call this yet-to-be-defined function *perfect_route*, and see what the procedure looks like in algorithm 8.

Algorithm 8 Centroids to recommendations in JSON format

```
Input: $C$                                        ▷ The centroids
1: recommendations ← ∅
2: for each $c \in C$ do
3:     for $i$ ← 1 to $|S|/|C|$ do
4:         $c'$ ← $c$
5:         for each feature ∈ $c'$ do
6:             if feature ≠ 0 then
7:                 feature ← ±5%
8:         recommendations.add(perfect_route ($c'$))
9: return recommendations
```

It was decided to output as many recommended standard routes as there are in input, consequently, if the number of clusters is c and the number of standard routes is $|S|$, each centroid will output $\frac{|S|}{c}$ routes. Additionally, each non zero centroid feature will be randomly modified by plus or minus 5% in order to accommodate as many driver's preferences as possible.

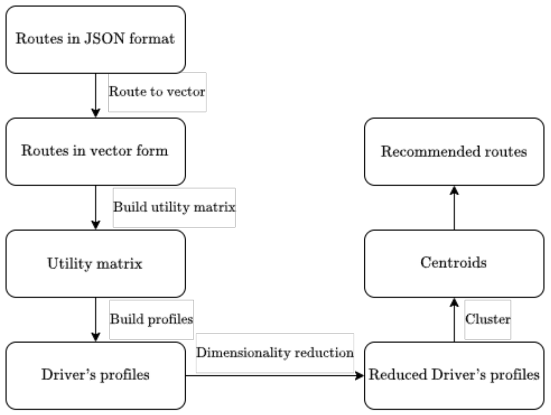


Figure 7: High level process for the first part of the solution

3.2.11 Putting everything together. In conclusion, let us go through the solution as a whole, which is illustrated in Figure 7 above:

- In the first step, we convert the standard and actual routes from JSON format to vectors;
- In the second step, we use these vectors to create a utility matrix where the rows are drivers, the columns are standard routes, and the rating function is the cosine similarity between a standard route and an actual route a driver performed;
- In step three, we make use of the utility matrix to create a profile for each driver;
- In step four, since the number of features can be very high, we perform dimensionality reduction on the driver's profiles;
- In step five, we cluster the driver's preferences (i.e. the driver's profiles);
- Finally, in the last step, we leverage the function used in part three which, given a driver's profile, outputs the perfect route for that driver. Expect that instead of inputting drivers profiles we input the centroids calculated in the previous step.

3.3 Part two

3.3.1 Problem statement. Input: The set of standard routes S and the set of actual routes A. Output: For each driver, the five recommended standard routes that if the driver does them, it minimizes the diversion.

As discussed in the related work section, there are three basic architectures for building a recommendation system: collaborative filtering, content-based, and hybrid. Here's a deeper explanation of how they work.

3.3.2 Collaborative filtering. First of all, we focus on collaborative filtering system that measures the similarity of users by their item preferences and/or measure the similarity of items by the users who like them. There are two techniques of collaborative filtering called user-user and item-item, which have a quite similar approach to fill the blank values in the utility matrix, but there is a conceptual difference between the two [4].

(1) User-User: in user-user technique, users form a group called neighborhood, and recommendations are generated on the preferences of the neighbors of user *U*. Indeed, for a user U, the cosine similarity - described in section 2 - is computed between the rating vectors of users who have rated the same items *I* of user *U*. This process identifies n similar users to *U*. Then, the rating for an item *I*, which *U* has not rated, is determined by selecting n similar users from the similarity list who have rated item *I*. This is achieved by calculating the rating as a weighted average of the ratings of the similar users, where the weights representing the similarities of these users to the user *U* [7].
(2) Item-Item: item-based filtering techniques compute predictions using the similarity between items, and not between users. Therefore, the initial step is to find the route pairs, and the drivers *D* who have rated for both routes *R* in the item pair. After this, compute the similarity between all pairs of items, using the cosine similarity - described in section 2. This defines how similar each pair of items is. Then, the missing rating of a route *i* for a driver *d* is determined by taking the weighted average of the ratings for the similar routes that the driver *d* has rated, where the weights are given by the similarity between the target route (*i*) and each similar route (*j*) [8].

Both these algorithms require the utility matrix - described in section 3.2.4 - and the cosine similarity measure - introduced in section 2 - to find the most similar drivers/items. While, the output is a matrix $n \cdot k$ where n is the number of drivers D, and k the top routes recommended to d. In our case $k = 5$. Below the pseudocode for a better explanation of one of the two created algorithms.

Algorithm 9 Item-Item algorithm

```
Input: $U$                                              ▷ The utility matrix
1: $sim \leftarrow (cos\_sim(i^*, j^*)$ for $j$ in $R$ if $i \neq j)$ for $i$ in $R$
2: for $d$ in $D$ do
3:     for each $i$ in $R$ do
4:         if $d_i$ not rated($i$) then
5:             $d_i$ next rated($i$) $\leftarrow w\_avg(sim\_list_{ij} \cdot d_j$, if $j$ in $d)$
6: return $U$
```

3.3.3 Content based. Content-based recommendation systems belong to the category of recommendation systems which are designed to propose items to users by considering the attributes or content of the items and the user's expressed preferences. This methodology is based on information derived from the behaviors and choices of the driver, utilizing route vectors described in section 3.2.2, and user profiles outlined in section 3.2.5. With profile vectors for both drivers and routes, it is possible estimate the degree to which a driver would prefer a route by computing the cosine similarity between the driver's and item's vectors [7]. Specifically, it emphasizes more on the analysis of the attributes of items in order to generate predictions, and recommendations are formulated based on driver profiles, utilizing features extracted from the content of routes that the driver has previously done [4]. Below, the pseudocode for an explanation of how the algorithm works.

Algorithm 10 Content based algorithm

```
Input: $profile, route, k$
1: $result \leftarrow \emptyset$
2: for each $i$ in $profile$ do
3:     $sim \leftarrow [(\cos(profile[i], route[j]), j)$ for $j$ in $route]$
4:     $sort(sim)$
5:     $res \leftarrow [arr[l][1]$ for $l$ in range(k)]
6: $result[i] \leftarrow res$
7: return $result$
```

The general idea is to calculate the cosine similarity between each element in *profile* and the elements in *route* in line three. In line four, we sort these similarities. Finally, in line five, we return the indices of the first five elements, which are the most similar ones. The final result is a data structure that associates each element in *profile* with a list of k indices corresponding to the most five similar elements in *route*.

3.3.4 LSH. The techniques described earlier can be greatly sped up by employing simhash with Locality Sensitive Hashing(LSH). The goal of LSH is to find pairs of similar items. The set-up is pretty simple:

- Firstly, we create a signature for all items. Signatures are short vectors representing items that preserve similarity;
- Secondly, we hash each signature into buckets.

Now, say we have an item *I* and we want to find the most similar item to *I*. In order to do that, we simply hash the item *I* and see in what bucket it gets put in. Once we know the bucket, we simply take the most similar item to *I* in the whole bucket.

Intuition. The idea is to randomly generate N hyperplanes and a signature vector for all items by categorizing the i-th entry as 1 if the vector falls above the i-th hyperplane, and 0 otherwise. Research [5] has proven that two similar vectors will, in all likelihood, fall in the same region of space, i.e. have similar signature vector and thus be put in the same bucket, when they are similar. This behavior is illustrated in Figure 8 where two similar vectors $\vec{a}$ and $\vec{b}$ are unlikely to be in different portions of space in terms of the defined hyperplanes.

Computing the signature. Given an item $\vec{a}$ and and hyperplane $\vec{h}$ we have that $\vec{a}$ is *above* $\vec{h}$ if and only if the dot product $\vec{h} \cdot \vec{a}$ is positive, otherwise it is *below*. This behavior is illustrated in Figure 9.

In order to compute the signature vector we verify where each item is positioned in relation to all the random hyperplanes. Specifically, for each hyperplane we set the i-th entry of the signature to 1 if the vector is above the i-th hyperplane and 0 otherwise. If we combine each vector row by row, we get the signature matrix, which we will be hashing into buckets. In algorithm 11 the procedure we just described is illustrated.

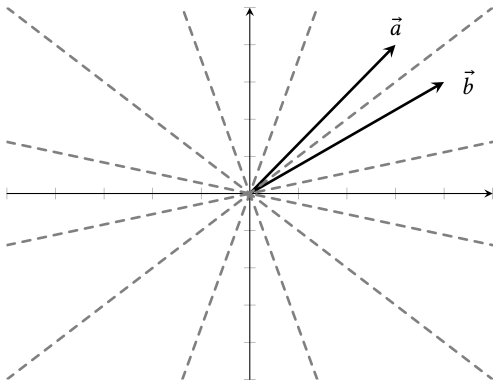


Figure 8: Two similar items $\vec{a}$ and $\vec{b}$ are more likely to fall into the same region of any random hyperplane than when they are dissimilar

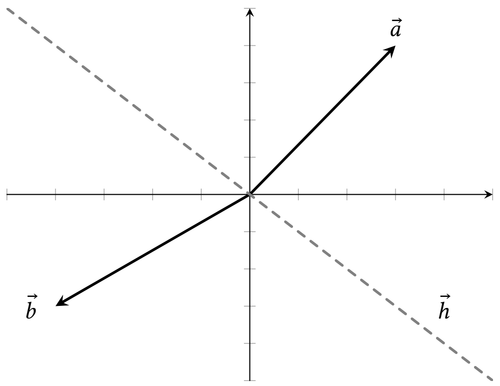


Figure 9: In this case, $\vec{a}$ has a positive dot product and $\vec{b}$ has a negative dot product

Algorithm 11 Signature matrix computation

```
Input: $H, I$                    ▷ The set of hyperplanes and items
1: $U \leftarrow \varnothing$
2: for each $\vec{i} \in I$ do
3:     for each $\vec{h} \in H$ do
4:         $U[\vec{i}][\vec{h}] \leftarrow$ sign $(\vec{h} \cdot \vec{i})$
5: return $U$
```

Now that we have the signature matrix, we put each row into some bucket via a hashing function. In order to find out which items are the most similar within a bucket, we calculate the *Hamming distance*. The Hamming distance is defined as the number of entries in which two vectors differ. For example, the vectors 1001 and 1000 have an Hamming distance equal to 1 because only the last bit is different.

We have taken a linear complexity function, which required us to compute the distance between our query vector and all of the previously indexed vectors — to a sub-linear complexity — as we no longer need to compute the distance for every vector — because they're grouped into buckets.

To sum up:

- We create a signature vector for all vectors;
- We hash these signature using an hash function;
- We hash the items we are looking for in order to find the corresponding bucket;
- We make use of the Hamming distance so that we know which items are most similar within a bucket.

All of the work presented in this section has been implemented leveraging the FAISS library, which greatly boosts performance due to the fact that insertion and lookup are parallelized [6]. The number of random hyperplanes used is 128, which according to the library, is a great trade-off between performance and accuracy.

3.3.5 Applying LSH to all the filtering techniques. Now that we defined a similarity search technique, we can use it to greatly improve performance for user-user filtering, item-item filtering, and content based filtering.

Content based filtering and LSH. In order to give a rating for route r to a driver d, we simply look for the 5 most similar routes to d. The reason why we can compare drivers and routes is that the driver's profiles and the standard routes have the same exact features when considering content based filtering. Due to the fact that we need to find routes, we index them using simhash and LSH. This schema is illustrated in Figure 10.

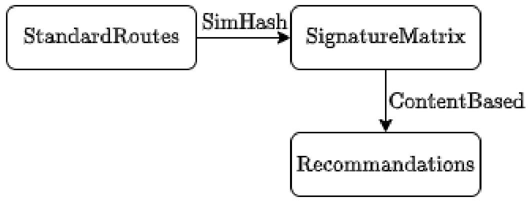


Figure 10: Architecture of content based LSH

Item-item collaborative filtering. In order to give a rating for route r to a driver d, we find the 5 most similar routes to r that driver d has rated and take the average as a rating estimate. Since we need to find similar routes, we will be indexing routes. The schema is illustrated in Figure 11.

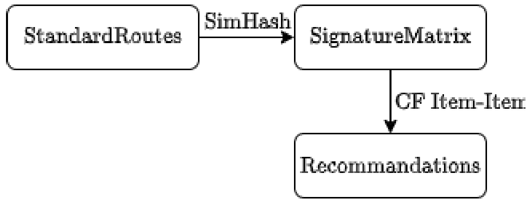


Figure 11: Architecture of item-item LSH

We fill the utility matrix completely and return the five highest rated routes for each driver.

User-user collaborative filtering. In order to rate a route r for a driver d, we find the 5 most similar drivers to d and take the average rating those drivers gave to r as a rating estimate. Since we need to find similar users, we are going to index the utility matrix. The schema is illustrated in Figure 12.

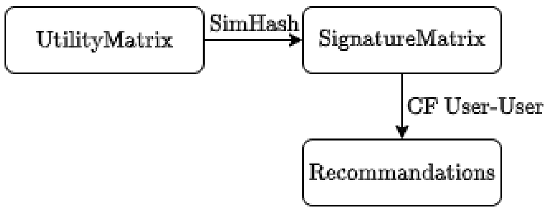


Figure 12: Architecture of user-user LSH

Again, we fill the utility matrix completely and return the five highest rated routes for each driver.

The item item and content based filtering with LSH will be used to create an hybrid recommendation system, as reported in the section below *Hybrid system* 3.3.6.

3.3.6 Hybrid system. Recent studies have demonstrated that, in certain scenarios, an hybrid approach could be more effective in some cases. Nowadays, a clear example of hybrid recommendation system is how Netflix works. Indeed, their recommendations are formulated by analyzing the viewing and exploration patterns of users with similar preferences (collaborative filtering), and by suggesting movies that share features with films that a user has rated highly (content-based filtering) [13]. At this point of the paper, once recommendation system algorithms have been defined in section 3.3.2 and 3.3.3, the next step involves developing an hybrid recommendation systems to find out a better quality of the recommendations. Depending on the type of the problem being considered, different strategies may be used [3].
Basically, the developed idea is to combine together the item-item collaborative filtering and the content based with the implementation of the Locality Sensitive Hashing method (LSH), introduced in section 3.3.4. The aim is to integrate the advantages of these approaches to provide more precise recommendations. Below, there is the architecture of the hybrid recommendation system.

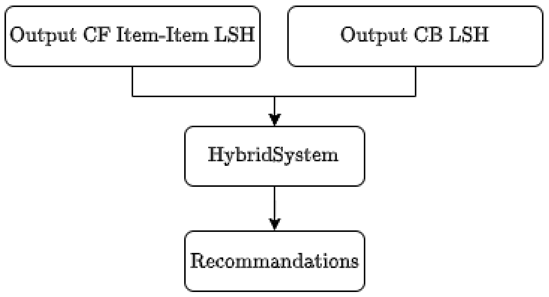


Figure 13: Architecture of hybrid recommendation system

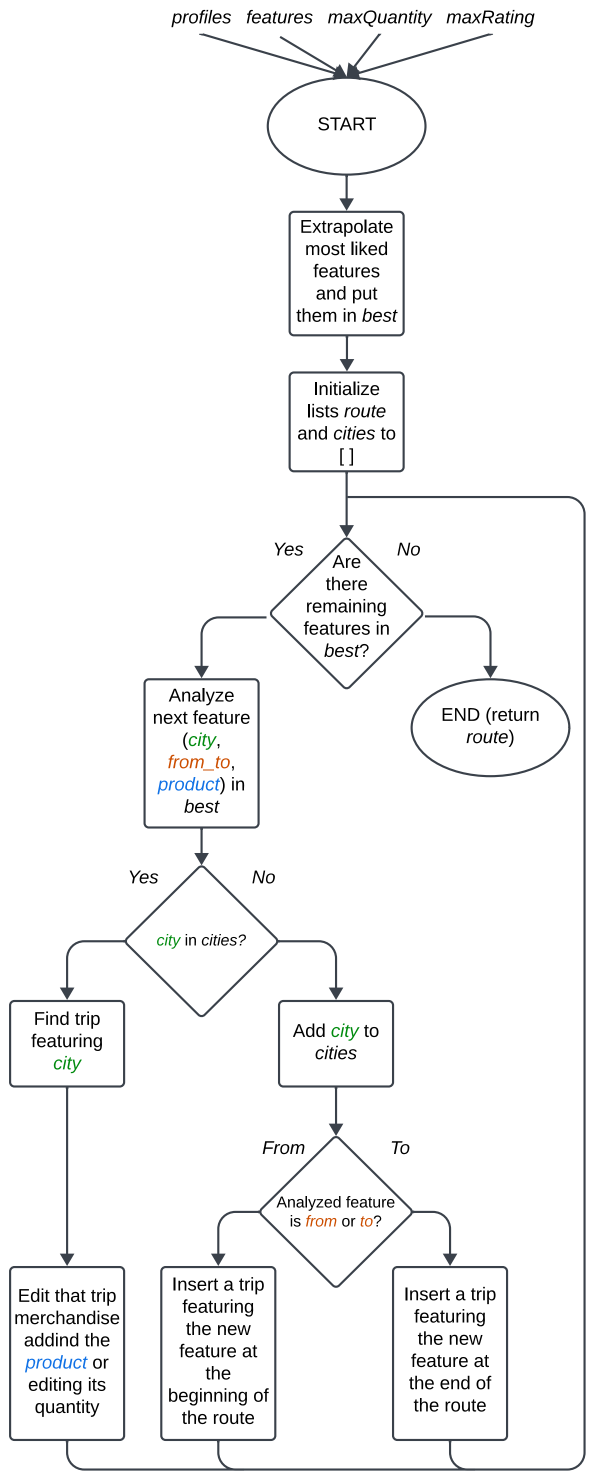
3.4 Part three

3.4.1 Problem statement. Input: The set of all standard routes S, the set of all actual routes A, and the set of all drivers D. Output: A route r for each driver in D.

3.4.2 General explanation and flowchart. To achieve the third objective of the project, we have to recommend, for every driver, a route that we think he or she will follow perfectly, or at least with low discrepancy. In order to do that, we define a function to build a route from scratch, for every driver, based on his or her preferences. The parameters passed to this function are:

- *profile*: the vector that contains the ratings of a driver for every feature calculated in part one (i.e. taking away bread from Rome). An example of it could be: [1, 3, 5, 1, 6, 2, ...];
- *features*: the dictionary that associates every feature to an index. It is used to understand what's the feature to which a specific rating is associated since this is not a known information in *profile*. An example of it could be: {("Rome", True, "bread"): 0, ("Venice", False, "wine"): 1, ...}, meaning that the rating for the feature "taking away bread from Rome" can be found in the vector *profile* at position 0, while the rating of the feature "bringing wine to Venice" can be found in position 1;
- *maxQuantity*: an integer representing the maximum transported quantity in a single trip among every product;
- *maxRating*: an integer representing the highest given rating among every driver and feature.

Here's a flowchart to better understand the process to create the route:


profiles
features
maxQuantity
maxRating
START
Extrapolate most liked features and put them in best
Initialize lists route and cities to [ ]
Are there remaining features in best?
Yes
No
Analyze next feature (city, from_to, product) in best
END (return route)
city in cities?
Yes
No
Find trip featuring city
Add city to cities
Analyzed feature is from or to?
From
To
Edit that trip merchandise addind the product or editing its quantity
Insert a trip featuring the new feature at the beginning of the route
Insert a trip featuring the new feature at the end of the route

In the following subsections, we analyze the main steps of the function:

3.4.3 Finding which features the driver likes the most. In algorithm 12, we extrapolate the most favorite features of the driver. To do this, we compute the average rating given by the driver without taking into consideration ratings <= 0 because doing otherwise

Algorithm 12 Compute best features

```
Input: $profile$
1:  $best \leftarrow \{\}$
2:  $avg \leftarrow cnt \leftarrow 0$
3:  for $rating$ in $profile$ do
4:      if $rating > 0$ then
5:          $avg$ += $rating$
6:          $cnt$++
7:  $avg$ /= $cnt$
8:  for $rating$ in $profile$ do
9:      if $rating > avg$ then
10:         put relative feature in $best$
11: return $best$
```

would result in an average too low for our purpose. After having iterated through every rating of the driver, we iterate once again adding to *best* whichever feature has the relative rating greater than a threshold that is equal to 1.8 times the average. At the end, we have the most favorite features of the driver inside *best*[2].

Algorithm 13 Find trip with specified city

```
Input: $feature, route$
1:  $city, fromto, product \leftarrow feature$
2:  $found \leftarrow False$
3:  $i \leftarrow 0$
4:  if $fromto$ then
5:      $loc \leftarrow$ "$from$"
6:  else
7:      $loc \leftarrow$ "$to$
8:  while not $found$ do
9:      if $route[i][loc] == city$ then
10:         $found = True$
11:     else
12:         $i++$
13: return $i$
```

3.4.4 Finding which trip in the route contains the city of the features. In algorithm 13, we loop through the trips in the route that we're building to find the specific trip in which there is the city of the feature. Since the city could be in two trips (one as the "from city" and the other as the "to city"), it's important to find the right trip according to the feature's *fromto* part. That's why we use the *loc* variable.

After having found the right trip, there can be two cases in which it's not enough to simply modify the trip. These two cases are:

(1) The one in which the feature's city is in the first trip as the "from city", but the feature's *fromto* part is False (meaning that the feature specifies about bringing the product TO that city);

[2] *best* is a dictionary in which keys are features (city, from-to, product) and values are tuples formed by the rating for that feature as the first component and the quantity of product as the second one. Here's an example of element for *best* dictionary: ("Rome", True, "banana"): (3.4, 19), meaning that the driver likes taking away from Rome 19 bananas with a rating of 3.4.

(2) The one in which the feature's city is in the last trip as the "to city", but the feature's *fromto* part is True (meaning that the feature specifies about taking away the product FROM that city).

In case 1, it is necessary to insert a new trip at the beginning of the route while in case 2 it is necessary to create a new trip to append at the end of the route.

Algorithm 14 Edit merch of trip

```
Input: $merch, product, quantity$
1: if $product$ in $merch$ then
2:     $merch[product] \leftarrow int((merch[product] + quantity)/2)$
3: else
4:     $merch[product] \leftarrow quantity$
```

3.4.5 Editing the trip's merchandise. In algorithm 14 we simply check if the product to be inserted in the trip already exists in it. If not, we insert it in the quantity specified, otherwise we compute and assign the average between that quantity and the one in which the product is already present (line 2).

Algorithm 15 New/edit trip

```
Input: $feature, route$
1: $city, fromto, product \leftarrow feature$
2: $cities.append(city)$
3: if $fromto$ then
4:     if $route[0]["from"] == ""$ then
5:         $route[0]["from"] \leftarrow city$
6:         add/edit product in route[0] merchandise
7:     else
8:         Insert new trip with product at the beginning of route
9: else
10:    if $route[-1]["to"] == ""$ then
11:        $route[0]["to"] \leftarrow city$
12:        add/edit product in route[-1] merchandise
13:    else
14:        Append new trip with product at the end of route
```

3.4.6 Managing cases where the city is not in the route yet. In algorithm 15, we firstly append the new city to the list of cities in order to "not forget" that the route now has that city in it.

Next, we check whether the city has to be put in a "from" or a "to" field. In the first case, we also check if the first trip already has a "from city". If not, we put our new city in it and we edit its merchandise adding or modifying the product quantity. In the other case, we create a new trip and insert it at the beginning of the route.

In case that the city has to be put in a "to" field, then we check if there's already a city in the "to" field of the last trip. If not, we modify that trip placing our city in its "to" field and then we modify the merchandise adding the product or modifying its quantity. In case that the "to" field already has a city, we create a new trip with the new city and containing the product and we place it at the end of the route.

3.4.7 Checking route validity. There can be 2 cases in which the built route do not meet the validity requirements:

(1) The route has no starting city (meaning *route[0]["from"] == ""*). In this case, we place a city in the "from" field;
(2) The route has no finishing city (meaning *route[-1]["to"] == ""*). In this case, we place a city in the "to" field.

We know that's not the best way to handle these situations. A good way to do that would have been to look at the merchandise of the trip involving the missing city and look at the ratings of the driver for those products, then choosing the city with the greater average.

4 EXPERIMENTAL EVALUATION

4.1 Data generation

4.1.1 Generating standard.json. To generate *standard.json*, we set a number of parameters that can be easily changed:

- number of routes to be generated.
- number of trips for each route.
- merchandise for each trip.
- list of possible cities to go to.
- list of possible products to be transported.

The procedure to generate the data is totally randomical and selects cities and products from the given lists.

4.1.2 Generating driver_attributes.json. Before generating *actual.json*, we must focus on another json called *driver_attributes.json*. *driver_attributes.json* contains a list in which elements represent drivers. Every driver has an id, a bunch of probabilities representing his or her stubborness in taking actions and two random ordered lists of preferences: one for cities and the other for products. Here's a detailed explanation of the possible actions for the driver:

- *del_city*: the initial probability to delete a city from a route;
- *add_city*: the initial probability to add a city to a route;
- *del_merch*: the initial probability to delete a product from a trip's merchandise;
- *add_merch*: the initial probability to add a product to a trip's merchandise;
- *edit_merch*: the initial probability to modify the quantity of a product in a trip's merchandise.

But why do we write "initial" probability and how do we decide which values to assign? We answer to the first question in the following section. To answer the second question, for every driver we generate two random boundaries between 0 and 1 (let's say 0.1 and 0.4 for driver 1), then we proceed to generate random values between those boundaries, meaning that driver 1 is a little bit stubborn about his or her decisions. If the boundaries were 0.6 and 0.9, then driver 1 would have been much more stubborn about his or her preferences and the probability to deviate from the standard route would have increased (thanks to the generated values).

4.1.3 Generating actual.json. To generate *actual.json*, we rely on both *standard.json* and *driver_attributes.json*, cycling through every <standard route, driver> pair and computing an actual route with probability 0.5. In simple terms, every time we generate an actual route, we analyze every element of the standard route and perform actions on them according to the driver's preference lists and stubborness. To better understand the following explanation of the procedure to generate an actual route, we introduce here the concept of *bias*, which is why we used "initial" probability in the previous section. The *bias* is an integer value that specifies the preference for a city or product and it's calculated depending on both the city or product and the action to perform. If the action to perform is negative (i.e. deleting a city from the route or deleting a product from a trip's merchandise), then the bias is equal to the number of city or products (depending on the subject of the action) minus the position in which the subject of the action is located in the relative preference list. For example, to calculate driver b's probability to delete "Rome" (the city in position 3 of the list of 90 elements *cities* under *preferences* in driver b's attributes) from the route: driver b's stubborness for *del_city* / (90 - 3). Even if the subborness of this driver for deleting cities from routes was as high as 0.9, the preference for "Rome" would result in a very low probability of removing it from a route. On the contrary, the bias for positive actions, such as adding a new city or product, is simply the position of the city or product in the relative *preference* list. Using again the previous example, we can calculate the probability of driver b adding "Rome" to the route: stubborness of driver b for *add_city* / 3.

Here's the procedure to generate an actual route having the standard route and the attributes of a driver (stubborness and preferences):

(1) for every trip in the standard route:
  - perform the calculation to see whether to delete the arriving city from the route (*del_city / bias*) and if successful, then remove that city from the route by setting the next trip's starting city as this trip's starting city and by removing this trip from the route;
  - if not, then proceed with the editing of that trip's merchandise:
    - for every product in the merchandise:
      - perform the calculation to see whether to eliminate the product from the merchandise (*del_merch* / bias) and if successful, delete it;
      - if not successful, perform the calculation to see whether to modify the product's quantity (*edit_merch / bias*) and if successful, set it to 30 (initial values can be maximum 20);
    - for every existing product that is not already present in the merchandise:
      - perform the calculation to see whether to add the product to the merchandise (*add_merch / bias*) and if successful, do it by setting its quantity to 30;

(2) for every existing city that is not already present in the route:
  - perform the calculation to see whether to add the city to the route (*add_city / bias*) and if successful:
    - insert a new trip that terminates with the city in a random position of the route;
    - generate a random merchandise for the new trip;
    - edit the merchandise with the appropriate procedure (see previous points);

4.2 Time Benchmark

The time performance of the algorithms described in this paper have been measured on a machine with the following specifications: Intel(R) Core(TM) i7-1260P 2.10 GHz, 16 GB RAM. The time performance evaluation is done using four different datasets. The procedure by which the data were created is the same as described in section Data Generation 4.1, but only some parameters have been changed arbitrarily. In all four datasets, there is a fixed number of *standard routes* set to 1000; instead the number of *drivers* and *features* double each time, while the number of *actual routes* depends on *standard routes* and *drivers*. Below, the composition of each dataset D1, D2, D3, and D4 in Table 1:

	Standard	Drivers	Actual	Features
D1	1000	100	15% · std · drivers	320
D2	1000	200	15% · std · drivers	640
D3	1000	400	15% · std · drivers	1280
D4	1000	800	15% · std · drivers	2560

Table 1: Dataset used for Evaluating Time

4.2.1 *Part one.*

To evaluate the time performance for the algorithms described in the section 3.2, the datasets in Table 1 have been used. In the Figure 14, the data on the execution time for the algorithm with and without dimensionality reduction are reported.

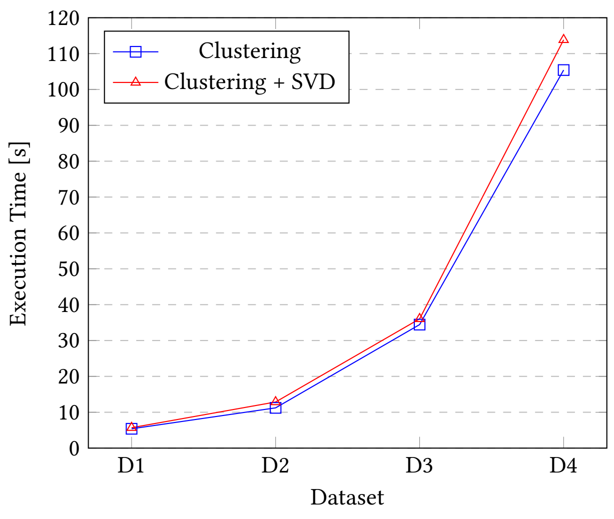


Figure 14: Evaluating Time Part One

The two versions are very similar. Performing SVD is quite expensive because it is repeated multiple times in order to have a very specific amount of information loss. Even though it is executed 30 times, since it cuts the number of dimensions so much, the clustering process takes a lot less time and outweighs the time lost when performing SVD. In most cases the number of dimensions is cut by more than 50%.

For more detailed data on the analysis of evaluating time, refer to the following Table 2 where the execution time [s] are reported based on the size of the utilized dataset.

	D1	D2	D3	D4
Clustering	5.39	11.22	34.44	105.41
Clustering + SVD	5.69	12.86	36.04	113.84

Table 2: Evaluating Time Part One

4.2.2 Part two. To evaluate the time performance for the algorithms described in the section 3.3, the datasets in Table 1 have been used. In Figure 15, the data on the execution time for each algorithm are reported.

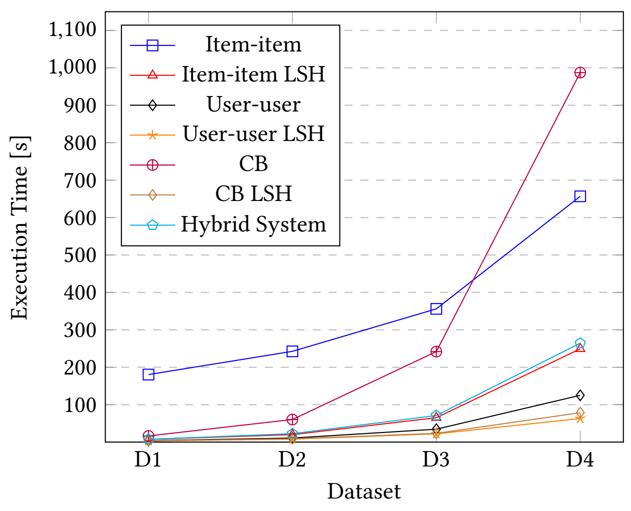


Figure 15: Evaluating Time Part Two

From these results, it can be inferred some considerations:

(1) First of all, the increase in the matrix size can lead to higher computational complexity. For instance, similarity calculation operations may take longer. The matrix size is determined by the product of the number of drivers and the number of routes. In this scenario, the number of drivers doubles while the number of routes remains constant at 1000. Therefore, if there are initially 100 drivers and 1000 routes, the matrix will be of size 100×1000. When the number of drivers doubles to 200, the new matrix size will be 200×1000. Doubling again to 400, the matrix size becomes 400×1000, and finally, with 800 drivers, the matrix size is 800×1000. Consequently, the utility matrix size is drivers $\cdot$ routes, where if the number of drivers is multiplied by n times, the final size will be n times the original number of drivers. In this case, the drivers have doubled for n = 2 three times. Therefore, the final matrix size will be 2^3, meaning it has increased eight times with the use of the dataset.

(2) Regarding the item-item CF algorithm, as shown in the Figure 15, an high execution time is observed starting from the use of D1. This may be attributed to the fact that the item-item algorithm needs to consider relationships among different routes (1000 items) to compute the cosine similarity and to generate recommendations. Since the number of routes remains constant, the execution time of the algorithm with subsequent datasets is relatively low.

(3) Furthermore, as reported in the Figure 15, user-user, content based, and all the algorithms with LSH method implementations, as well as hybrid system, exhibit low execution times at the beginning with the dataset D1 and D2. While, from the use of D3, which has larger dimensions, compared to the other mentioned algorithms, the content-based system has a significant increase in execution time. Regarding this, a longer runtime may be attributed to the fact that, in addition to the size of the matrix size, and thus taking more time for its construction, it also extends the time required for generating the driver profiles - described in section 3 - used within the CB algorithm. Indeed, the algorithm's time complexity is $O(n \cdot m \cdot features)$, where n is the number of driver, m is the number of routes, and features is $2 \cdot \text{city} \cdot items$. Increasing both the number of drivers (n), and features can lead to a more significant growth in execution time. The process of creating detailed profiles for each driver and route involves more computational complexity, as the algorithm must consider numerous attributes.

(4) To conclude, the impact of the Locality Sensitive Hashing (LSH) method has been significant. Indeed, the use of LSH has demonstrated to be beneficial by reducing the runtime algorithms. LSH's ability to perform approximate searches - as explained in section 3.3.4 - allows for a reduction in computational complexity, leading to a significant acceleration in the recommendation process. In particular, this approach has proven to be efficient in maintaining low execution times even for larger dataset dimensions as D4.

For more detailed data on the analysis of evaluating time, refer to the following Table 3 where the execution time [s] of each algorithm are reported based on the size of the utilized dataset.

	D1	D2	D3	D4
CF Item-Item	180.46	242.52	356.01	656.73
CF Item-Item + LSH	7.50	19.97	65.20	249.17
CF User-User	3.93	10.82	34.49	125.07
CF User-User + LSH	3.16	7.93	22.09	63.03
CB	16.45	60.37	241.86	987.31
CB + LSH	3.44	8.72	23.40	78.59
Hybrid System	7.54	22.49	71.12	264.24

Table 3: Evaluating Time Part Two

4.2.3 Part three. As it can be seen from Figure 16, the requested time to compute part three grows linearly with respect to the input as when the input size doubles, so does the elapsed time.

4.3 Performance evaluation

4.3.1 Part one. In this section, we will compare the performance of our solution with a baseline naive implementation. The baseline solution is quite simple: it just outputs the standard routes it was given in input.

In order to evaluate the performance, we will assign randomly one recommended standard route to a random driver and calculate

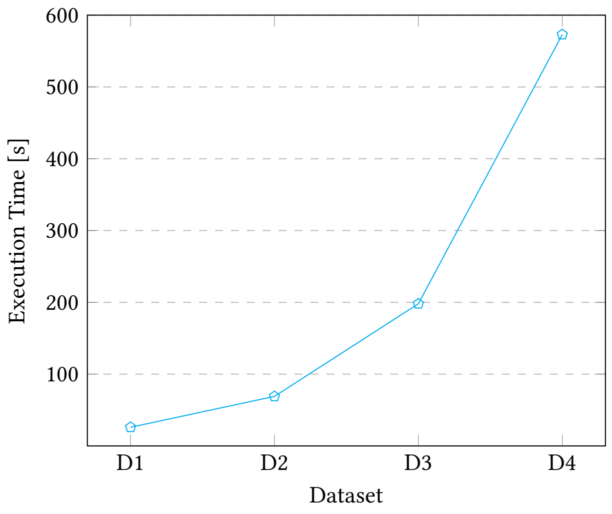


Figure 16: Evaluating Time Part Three

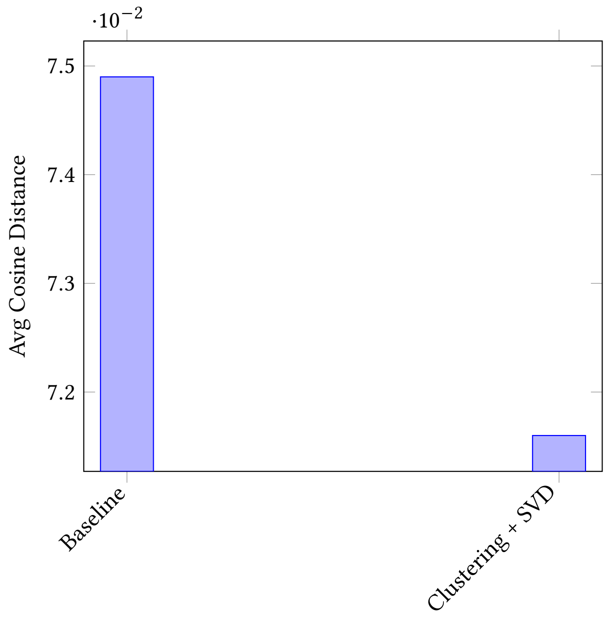


Figure 17: Performance Evaluation Part One

the divergence between the route and the resulting actual route, and we repeat this process n times (in our case $n = 5 \cdot |D|$). The way we calculate the divergence between the route and actual route is the cosine distance. Thus, the overall performance metric is the *average cosine distance.*

In Figure 17, the results of our tests are shown. The calculations were done by using 1000 drivers and 1000 standard routes.

Both solutions perform quite well, given the very low average cosine distance. Unfortunately though, our solution is not significantly better than the baseline. Here some general considerations:

- The input size is not big enough. We would have loved to test the recommendation system with a larger input size, however, our hardware resources are very limited. We strongly believe that in a situation where there are many more drivers and standard routes, our solution will overcome the baseline even more. Even an input as little as 1000 drivers and 1000 standard routes is enough to make the program run for about 10 minutes;
- The nature of the task is very random, thus making recommendations is very tough. In fact, we have no way of knowing which routes will be recommended to each driver.

4.3.2 Part two. To evaluate the precision for the algorithms described in section 3.3, the same dataset - used to evaluate the performance for the part 4.3.1 - has been adopted.

In addition to the algorithms explained earlier, a baseline solution has been added. This technique simply recommends five random standard routes to each driver. Since the cosine distance is used to calculate the divergence between the standard route and the actual one, the overall performance metric is the *average cosine distance.*

In Figure 18, the data on the precision for each algorithm is reported. Unfortunately, the results are not so significantly better than the baseline. Here, some considerations on the results and on the possible reasons:

- In the results, the item-item LSH algorithm proves to be the most effective in recommending routes to a driver. This may be attributed to the abundance of items in our dataset;
- Given limited hardware resources, we believe that the input size used was not sufficient to achieve significantly better performance compared to the baseline. Indeed, the size of the dataset could still constrain the capacity of some algorithms to learn complex patterns, especially if the variability among routes and drivers is significant, as in this research. In line with this observation, and based on the results, at the beginning, we entertained the idea that a hybrid approach could provide enhanced recommendation solutions by integrating both item-item LSH and content-based algorithms. However, this did not produce the expected results probably because the dimension of the date.

4.3.3 Part three. To evaluate the precision of the program we propose as solution for part 3, we simulate a driver following two routes. One is randomly generated (with the same method used to generate standard.json), the other is taken from *part3.json*, the file containing our suggestions for every driver. We compute then the actual routes and calculate the similarity for both of them. If the solution route is performed with less discrepancy than the random route, we consider it a success. After having followed this procedure for every driver, we can analyze the success rate and the average similarity for random generated routes and solution ones. Figure 19 is intended to represent the performance comparison between part three solutions and randomly assigned routes. We also want to mention that the success rate is 97.83% and the improvement in performance is 268% (based on similarity).

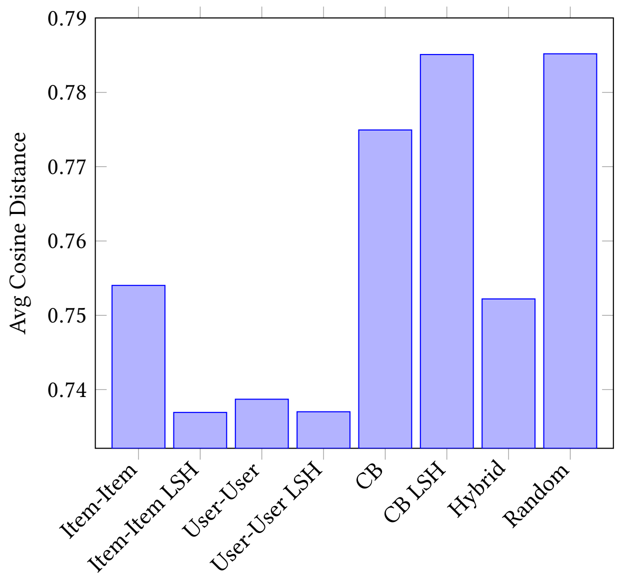


Figure 18: Performance Evaluation Part Two

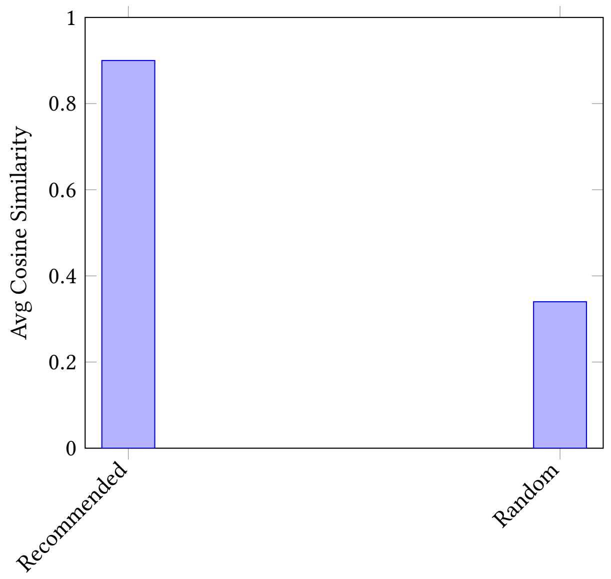


Figure 19: Performance Evaluation Part Three

5 CONCLUSION

The solutions presented and examined in this research produced results that demonstrated an improvement over a baseline recommendation system. These outcomes represent a promising starting point for integrating more sophisticated data mining techniques to further develop an advanced recommendation system that is even more effective.

In this study, various recommendation system techniques have been introduced in the logistics domain to enhance efficiency and accuracy within a transportation company. In general, the problem is addressed by employing approaches such as k-means clustering, recommendation systems (collaborative-filtering and content-based), and a hybrid solution combining item-item and content-based methods. A crucial aspect we had to manage was the runtime of algorithms, considering the large amount of data typically utilized by a transportation company. We addressed this challenge by successfully integrating the Local Sensitive Hashing method. This strategy proved to be fundamental and efficient with large datasets while also maintaining performance.

To conclude, this study provides a detailed overview and practical demonstration of recommendation systems in the context of logistics.

REFERENCES

[1] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[2] Robin Burke. 2002. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction* 12 (2002), 331–370.

[3] Zhiyuan Fang, Lingqi Zhang, and Kun Chen. 2016. Hybrid Recommender System Based on Personal Behavior Mining. *ArXiv* abs/1607.02754 (2016). https://api.semanticscholar.org/CorpusID:16504959

[4] F.O. Isinkaye, Y.O. Folajimi, and B.A. Ojokoh. 2015. Recommendation systems: Principles, methods and evaluation. *Egyptian Informatics Journal* 16, 3 (2015), 261–273. https://doi.org/10.1016/j.eij.2015.06.005

[5] Qixia Jiang and Maosong Sun. 2011. Semi-supervised simhash for efficient document similarity search. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. 93–101.

[6] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[7] J. Leskovec, A. Rajaraman, and J.D. Ullman. 2020. *Mining of Massive Data Sets*. Cambridge University Press. https://books.google.it/books?id=S4HCDwAAQBAJ

[8] G. Linden, B. Smith, and J. York. 2003. Amazon.com recommendations: item-to-item collaborative filtering. *IEEE Internet Computing* 7, 1 (2003), 76–80. https://doi.org/10.1109/MIC.2003.1167344

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[10] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. 2011. Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*. 166–171. https://doi.org/10.1109/ICDCSW.2011.20

[11] Kristina P Sinaga and Miin-Shen Yang. 2020. Unsupervised K-means clustering algorithm. *IEEE access* 8 (2020), 80716–80727.

[12] Carlos Oscar Sánchez Sorzano, Javier Vargas, and A Pascual Montano. 2014. A survey of dimensionality reduction techniques. *arXiv preprint arXiv:1403.2877* (2014).

[13] Poonam B Thorat, Rajeshwari M Goudar, and Sunita Barve. 2015. Survey on collaborative filtering, content-based filtering and hybrid recommendation system. *International Journal of Computer Applications* 110, 4 (2015), 31–36.

[14] Michel Verleysen and Damien François. 2005. The curse of dimensionality in data mining and time series prediction. In *International work-conference on artificial neural networks*. Springer, 758–770.

[15] Hongzhi Yin, Bin Cui, Jing Li, Junjie Yao, and Chen Chen. 2012. Challenging the long tail recommendation. *arXiv preprint arXiv:1205.6700* (2012).