Identifying the compensation for vaguely followed trips

Russo Riccardo
mat. 247062
riccardo.russo-1@studenti.unitn.it
University of Trento
Via Sommarive, 9, 38123 Povo, Trento

Nicolò Simonato
mat. 249206
nicolo.simonato@studenti.unitn.it
University of Trento
Via Sommarive, 9, 38123 Povo, Trento

Lorenzo Gandini
mat. 248430
lorenzo.gandini@studenti.unitn.it
University of Trento
Via Sommarive, 9, 38123 Povo, Trento

1 INTRODUCTION

The logistics industry plays a pivotal role in facilitating the continuous flow of merchandise between cities, ensuring the timely delivery of goods to meet customer demands. However, a significant challenge faced by logistics companies arises from the inherent variability in driver behavior during the execution of predefined routes. Equipped with advanced hardware like GPS, trucks are assigned predefined routes by logistics firms. These routes are strategically designed, specifying the order of city visits and the respective types and volumes of goods to be transported. Despite such strategic planning, drivers frequently diverge from these routes, altering merchandise quantities and modifying the city sequence based on personal preferences.

Such deviations between planned and actual routes introduce inefficiencies, such as increased operational costs. To address this issue, our objective is to develop a data-driven solution able to offer route suggestions that better resonate with driver predilections. By acquiring and understanding drivers' behavior, we aim to decrease the disparity between standard and actual routes. This challenge's importance lies in its capacity to boost operational efficacy in the logistics domain. By refining predefined routes using historical data on driver behavior, firms can reduce fuel usage, curtail transportation expenses, and enhance delivery schedules. Additionally, this strategy can bring a more congruent relationship between logistics firms and their drivers, integrating individual preferences.

In our research, we reveal a comprehensive system that not only recommends personalized predefined routes for each driver but also categorizes these routes according to their likelihood of deviation. Moreover, we propose a technique to formulate ideal predefined routes, aiming to reduce discrepancies and promote a more efficient logistics operation. The solution utilizes various distance metrics to measure the disparity between standard and actual routes, considering factors such as city sequences, product types, and quantities. The key components of the solution include:

(i) **Recommendation of New Standard Routes:** The solution includes the modification of standard routes by choosing, for each one, the route with the least total distance. This selection is carried out iteratively across all of them, ensuring an optimal congruence between standard and actual paths.
(ii) **Iterative Ranking for Each Driver:** For every identified driver, the solution iteratively assesses standard routes based on their divergence from the driver's actual routes. This ranking encompasses distance measures for city itineraries, merchandise types, and quantities, facilitating the identification of the most appropriate predefined route for each driver.
(iii) **Driver-Centric Analysis:** The solution employs a driver-centric approach, examining the historical routes of each driver to discern their preferences and behavior patterns. Essential data, including visited cities and transported item types on each trip, is gathered. These insights play a crucial role in comprehending driver inclinations, facilitating the alignment of routes accordingly.

2 RELATED WORKS AND TECHNOLOGIES

In this section, we provide a concise overview of the key technologies and methodologies that formed the foundation of our solution. These elements were initially fundamental to our approach and subsequently tailored to fit the operational environment, enabling us to extract meaningful insights from the dataset. While these techniques are foundational and well-recognized in the domain, their implementation was specifically adapted to meet the unique requirements and challenges of our project.

2.1 Similarity Measures

In data analysis, similarity measures play a vital role by enabling the comparison of elements within datasets. These measures provide a quantifiable means to assess the degree of similarity or dissimilarity between data points, a critical aspect for tasks such as clustering, classification, and data mining.

2.1.1 Jaccard Similarity. Jaccard Similarity serves as a metric for measuring the overlap between two sets, calculated as the size of their intersection divided by the size of their union. Mathematically, for sets A and B, it is given by:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

This index proves valuable in measuring the extent of commonality and distinctiveness between two sets.

2.1.2 Hamming Distance. Hamming Distance is a measure of difference between two strings of equal length, computed as the number of positions where the corresponding symbols are different. For strings x and y, it is expressed as:

$$H(x, y) = \sum_{i=1}^{n} 1_{x_i \neq y_i}$$

where $1_{x_i \neq y_i}$ is an indicator function that equals 1 if $x_i \neq y_i$ and 0 otherwise.

2.2 One Hot Encoding

One-hot encoding is a technique to represent categorical variables as binary vectors. For a categorical variable with n distinct categories, one-hot encoding creates a binary vector of length n. Each category is represented by a unique position in the vector, and the presence or absence of a category is indicated by a binary value (1 or 0). The process involves the following steps:

(i) Identify the distinct categories in the variable.
(ii) Assign a unique index to each category.
(iii) Create a binary vector of length n, where n is the number of distinct categories.
(iv) Set the value at the index corresponding to the category to 1, and all other values to 0.

2.3 Frequent Itemset

Frequent Itemset refers to a group of items commonly occurring together within a transactional database. This concept was utilized to discern patterns and associations in our dataset, particularly to identify itemsets with a frequency surpassing a predetermined threshold. This approach is commonly employed in market basket analysis for detecting regularly co-purchased items.

In data mining , the concept of frequent itemsets plays a crucial role. A frequent itemset is a set of items (or elements) that appears together in a significant number of transactions within a dataset. The support count of an itemset is the number of transactions in which the itemset appears. Mathematically, if I is an itemset, the support count $support(I)$ is the number of transactions that contain all the items in I. To identify frequent itemsets, a support threshold is set. Itemsets whose support count is equal to or exceeds this threshold are considered frequent, while those below the threshold are considered infrequent. Several algorithms, such as the Apriori algorithm and the FP-growth algorithm, are commonly employed to efficiently discover frequent itemsets in large datasets. By identifying these frequent itemsets, we can gain valuable insights into common trends and behaviors within the data, which is instrumental for subsequent analysis and decision-making processes.

3 SOLUTION

This section provides a detailed exposition of our proposed solution, systematically addressing the points of the given problem. Our methodology entails a rigorous adaptation and application of previously explored analytical frameworks, meticulously customized to align with the specificities of our case study. In executing this approach, our objective is to adeptly navigate and resolve each component of the problem, taking advantage of the empirical insights and recurring patterns revealed during our initial analysis phase.

3.1 Route Similarity Analysis

As part of our solution to analyze route similarities, we have developed a three-step methodology. This approach is designed to quantify the similarity between different routes in our dataset. The methodology comprises the creation of city shingles from each route and the computation of a modified Jaccard similarity score between pairs of routes. The second and third step focuses on the execution of modified version of hamming and quantity distance to include products and quantities associated in our calculation.

3.1.1 ***Jaccard Distance Function.*** The js_city function expands on the concept of Jaccard Similarity, tailored specifically for comparing routes based on city sequences. The function begins by extracting the set of cities from each route using the `shingling` function, which returns a list of cities in the order of occurrence.

The function iterates through both shingle lists, incrementing a cumulative score based on the presence and relative positions of matching cities. When the cities in both routes match and are in the same position (index) within their respective shingle lists, a score of 1 is added to the cumulative score. This emphasizes the significance of identical, sequential occurrences of cities in both routes. When the matching cities are present in both routes but are not in the exact same position, a score of 0.5 is added to the cumulative score. This accounts for partial matches, capturing situations where cities appear in different orders but still contribute to the overall similarity.

The final score is normalized by the total number of unique cities across both routes. The distance score is then derived by subtracting the normalized similarity score from 1. This distance metric reflects how dissimilar the routes are, with a lower score indicating higher similarity. The function is designed to handle edge cases, such as empty routes or routes with no common cities, by returning a distance score of 0 and 1 respectively in such scenarios.

Algorithm 1 Jaccard Distance for City Routes

```
function JS_CITY(route_1, route_2)
    shingles_1 ← SHINGLING(route_1)
    shingles_2 ← SHINGLING(route_2)
    score ← 0
    for i in range(length(shingles_1)) do
        for j in range(length(shingles_2)) do
            if shingles_1[i] = shingles_2[j] then
                if i = j then
                    score ← score + 1
                else
                    score ← score + 0.5
                end if
            end if
        end for
    end for
    union_set ← shingles_1 ∪ shingles_2
    unique_elements ← UNIQUE(union_set)
    total ← LENGTH(unique_elements)
    similarity ← score / total
     return 1 - similarity
end function
```

3.1.2 ***Hamming Distance Function.*** The hm_merch function implements the Hamming Distance concept, tailored for comparing products of the associated routes. Initially, it utilizes the create_one_hot function to convert the merchandise list of each route into a one-hot encoded format. The resulting output is a list of lists, where each inner list represents the one-hot encoded format of

a specific trip in the route. Each trip is defined as the pair (*from: City_A*, *to: City_B*) with its products.

{'from': Milan, 'to': Rome, 'merchandise':{'Milk':7, 'Oranges':10}},
{'from': Rome, 'to': Verona, 'merchandise':{'Cherry':5, 'Bread':2}},
{'from': Verona, 'to': Trento, 'merchandise':{'Milk':5, 'Water':8}}
List of merchandise in this route: Milk, Water, Oranges, Cherry, Bread.

One_hot_encoding = [[10100],[00011],[11000]]

This encoding effectively indicates the presence or absence of each item during a particular trip, enabling a bitwise comparison. The function ensures that both lists of one-hot encoded representations are of equal length by appending zeros to the shorter list, accommodating scenarios where one route has more trips than the other.

Subsequently, the function computes the Hamming Distance by evaluating the disparities in merchandise across each trip. The Hamming Distance is calculated as the sum of discrepancies between corresponding elements in each pair of lists (sum of the bitwise **xor**), normalized by the union of unique items involved in the comparison (sum of the bitwise **or** between each pair of lists). This approach provides a standardized measure of dissimilarity, allowing for effective comparison of merchandise routes.

A {'from': Ala, 'to': Avio, 'merchandise': {'Milk': 1, 'Oranges': 5}}
B {'from': Ala, 'to': Avio, 'merchandise': {'Milk': 1, 'Oranges': 11, 'Bread': 3}}
List of merchandise in this route: Milk, Oranges, Bread.

A = (one_hot : 110)
B = (one_hot : 111)
xor = 1 , or = 3
hm_dist = 1/3 = 0.33

3.1.3 ***Quantity Distance Function.*** Regarding the quantity distance, we use the normalized absolute difference. It is a measure that quantifies the relative discrepancy between corresponding elements in two vectors, normalized by the maximum value at each position. This distance metric is useful when you want to emphasize the proportion of the difference with respect to the magnitude of the values being compared. For two vectors $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_n)$, the normalized absolute difference d_i at each position is calculated as:

$$d_i = \frac{|a_i - b_i|}{\max(a_i, b_i)}$$

The comparison is conducted utilizing one-hot encoding, wherein only the quantities of products common to both trips are considered. The overall normalized absolute difference between the vectors can be computed as the sum of these individual normalized differences.

3.1.4 ***Final Distance Routes.*** These functions provide a robust framework for assessing the similarity between routes in our dataset. In the distance_route function, the final result is derived from the weighted sum of the outcomes of three distinct functions (js_city, hm_merch and qnt_dist).

The weights assigned to these functions are user-configurable allowing users to customize emphasis based on their specific preferences or requirements. The default parameters are:

(i) city_weight = 0.55
(ii) merch_weight = 0.30
(iii) quantity_weight = 0.15

This configuration prioritizes the path taken, followed by the type of merchandise carried, and finally, the quantity.

3.2 Recommended Standard Routes

Providing a recommendation to the company on what standard routes it should have, the solution to the specified challenge involves a systematic approach. By iterating through existing standards, the goal is to evaluate and potentially replace them with actual routes based on their performance.

For each standard, a list is populated containing all actual routes based on that standard. For every pair (actual - actual) and (standard - actual), the distance between the two routes is calculated using the distance_routes function from the previous chapter. The result of the new_standard function is a list of dictionaries where the key is the pair (id_route_1, id_route_2), and the value is the distance between them (e.g., [(a221, a223): 0.23, (s23, a183): 0.53]). It's important to note that the list contains only elements based on the same standard route, meaning it can only be replaced by an actual route based on it or the standard route itself.

Subsequently, for each unique ID, the sum of distances between pairs having that ID as a member is calculated, resulting in a dictionary indicating how much that ID differs from all other routes. Using the find_min_value function, we replace the existing standard route with an actual route, only if the observed actual route has a total distance value lower than the total distance value of the standard route. By doing this we obtain new standard routes that approximates the routes performed by the drivers better than the previous standard routes.

S0 {total distance= 12}

a20 {total distance= 8}

a261 {total distance= 20}

new S0 -> a20

3.3 Prioritizing Standard Routes for Minimal Diversions

In addressing the challenge of optimizing driver routes, the objective is to construct a prioritized list of standard routes for each driver. The fundamental principle guiding this arrangement is to ensure that the higher a standard route is positioned in the list, the less

deviation it entails for the respective driver. In essence, the goal is to establish an order that minimizes diversions, thereby enhancing overall efficiency and effectiveness in the driver's journey.

The standard_ranked function aggregates all actual routes for a specific driver.

Algorithm 2 Standard_ranked (first part)

```
act_driver ← []
for act in list_actual_routes do
    if act['driver'] = driver_id then
        act_driver.append(act)
    end if
end for
```

For each standard route, it calculates the distance using the previously mentioned distance_routes function, considering all actual routes taken by the driver. The result is a dictionary structured as follows: {standard_route_id: sum of distances with all actual routes}.

Algorithm 3 Standard_ranked (second part)

```
dictionary_driver ← {}
for st in list_of_standard_routes do
    for act in list_of_driver_standard_routes do
        if st['id'] ∉ dictionary_driver then
            dictionary_driver[st['id']] ← distance_routes()
        else
            dictionary_driver[st['id']]+ = distance_routes()
        end if
    end for
end for
```

Subsequently, the top 5 standard routes with the lowest sum of distances from actual routes are selected for each driver. This selection process is based on the principle of minimizing diversions, ensuring that the higher a standard route is positioned in the list, the less deviation it entails for the respective driver.

Algorithm 4 Standard_ranked (third part)

```
routeKeys ← best 5 st_routes in dizionario_driver
ranked ← {"driver" : driver_id, "routes" : routeKeys}
return ranked
```

In essence, the final output is a prioritized list of standard routes for each driver, reflecting their preferences and minimizing diversions. This approach enhances the efficiency and effectiveness of each driver's journey within the transportation system.

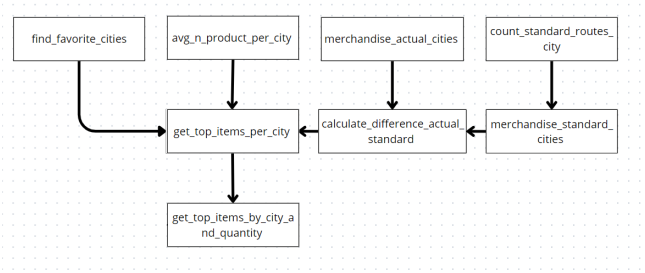
3.4 Generating Ideal Standard Routes for Each Driver

The ideal standard routes with the least divergence for each driver are derived by calling the function `analyze_driver_data`. This function orchestrates the execution of various sub-functions to obtain the optimal route for the specified driver.

Regarding city preferences, we first calculate the number of times each driver is expected to visit every city, adhering to the guidelines of their assigned standard route. We then determine the actual frequency of visits to each city by extracting data from the actual routes. However, relying solely on actual routes can lead to skewed data, as a driver might visit a city simply because it was included in their assigned standard route. To accurately assess each driver's preference for a city, we will use a specifically devised formula, explained later, that evaluates their inclination based on the frequency of visits relative to the expected number of visits. The number of selected cities is calculated by utilizing the average length of routes undertaken by the respective driver. This approach takes into account the preferred route length of the driver when determining the number of cities to be included.

We implement a similar methodology for products, extending our analysis to account for the driver's preferences in specific cities. For each city in consideration, we calculate the products that the driver delivers more frequently than required, employing the formula mentioned before to determine the deviation between the actual and expected occurrences. In determining the number of products, we employ the average of the products delivered by the driver in their preferred cities. The cities considered are those identified as preferred by the driver.

The quantity associated with each product is calculated by taking the average quantity of that specific product delivered in a particular city. The pair city-product is taken from the previous mentioned methodologies to find the most preferred cities and the associated products. The step-by-step explanation of each sub-function is provided below:



merchandise_actual_cities. : This function generates a dictionary for the analyzed driver, containing visited cities and the frequency of each product delivered to those cities. The dictionary format is {'city1': {'product1': frequency1, 'product2': frequency2, 'product3': frequency3}, 'city2': {'product4': frequency4, 'product5': frequency5, 'product6': frequency6}, ...}.

count_standard_routes. : This function returns a dictionary for the analyzed driver, containing standard routes as keys and number of times the driver has been assigned to each route as value.

merchandise_standard_cities. : This function takes as input the result of the count_standard_routes function. For each key-value pair in the returned dictionary, we select the standard route with the associated ID and retrieve all cities in the "to" field, along with all the merchandise to be delivered to these cities.

Algorithm 5 Merchandise_standard_cities (first part)

```
city_counts ← {}
for id, count in merchandise_actual_cities_result do
    for route in standard_route do
        if route["id"] == id then
            for i in range(len(route["route"])) do
                city ← route["route"][i]["to"]
                merchandise_list ← trip merchandise
            end for
        end if
    end for
end for
```

For each product associated with the city, we sum the "count" contained in the dictionary returned by merchandise_standard_cities. This way, we obtain a result like [{'city1': {'product1': count, 'product2': count}}, {'city2': {...}}], indicating that the driver in question was supposed to deliver products to the cities "count" times.

Algorithm 6 Merchandise_standard_cities (second part)

```
for item in merchandise_list do
    for city in city_counts do
        city_counts[city][item]+ = count
    end for
end for
standard_merch ← {}
for city, items in city_counts.items() do
    merch_data ← {city: dict(items)}
    standard_merch.update(merch_data)
end for
```

calculate_difference_actual_standard. : This function is used to calculate the products that a driver prefers to deliver in each city. It takes two dictionaries, `actual_merch` and `standard_merch`, corresponding to what a driver has delivered in a particular city and what they were supposed to deliver according to company guidelines, respectively. By iterating through both dictionaries, we use the following formula to determine the score for each city-product combination:

$$\text{SCORE} = (\text{count\_act} - \text{count\_st}) + \frac{\text{count\_act}}{2}$$

In this formula, `count_act` represents the number of times a driver has delivered a specific product in a given city, and `count_st` represents the number of times the driver was expected to deliver the same product in that city. The formula is not a simple subtraction between what the driver did and what he was supposed to do; instead, it gives weight to the number of times a driver has delivered a product, considering that the drivers often follow their preferences.

The concept can be better clarified with an example representing a possible scenario:

```
Driver A:
  actual Rome: {milk: 30} - standard Rome: {milk: 30}
Driver B:
  actual Rome: {milk: 1} - standard Rome: {milk: 0}
Driver C:
  actual Rome: {milk: 60} - standard Rome: {milk: 60}
Driver D:
  actual Rome: {milk: 60} - standard Rome: {milk: 10}
```

Scores:

- Score for Driver A in Rome for Milk: 0
- Score for Driver B in Rome for Milk: 1
- Score for Driver C in Rome for Milk: 0
- Score for Driver D in Rome for Milk: 50

If a simple subtraction was used like in the upper example, driver B would appear to prefer to deliver milk to Rome more than drivers A and C, since B delivered 1 unit more than expected. However, A and C delivered 29 and 59 more units of Milk to Rome than B. Following this reasoning and using the new formula, we obtain these new results

- Score for Driver A in Rome for Milk: 15
- Score for Driver B in Rome for Milk: 1.5
- Score for Driver C in Rome for Milk: 30
- Score for Driver D in Rome for Milk: 80

Thanks to our formula, Driver D remains the one with the highest score since he delivers a significantly higher number of times the specified product compared to the expected deliveries. However, Drivers A and C have a higher score than Driver B as they deliver a considerably greater number of times the milk to Rome compared to B.

avg_n_product_per_city. : This function calculates the average number of products carried by the driver to each city. Emphasis is placed on the destination city to underscore the importance of where a product arrives rather than its point of origin. The initial output is a dictionary structured as follows: {"driver_id": [["city1", number_products_carried], ["city2", number_products_carried], ...]}. The final dictionary is derived by computing the mean of products carried for each city.

find_favorite_cities. : This function identifies the cities in the preferences of each driver by calculating, as in the previous function, the score of each city for the respective driver. We can divide the function into three parts:

In the first part, the number of times the driver visits each city using the actual routes is extracted. A similar approach is adopted for the standard routes, yielding a result indicating the expected number of times the driver should have visited each city, obtaining two dictionaries of the form {city1: count1, city2: count2, ...}.

Algorithm 7 Find_favorite_cities (first part)

```
city_driver ← extract_cities_per_driver(driver)
cities_counts ← Counter(city_driver)
frequent_cities_actual ← {city: count for city, count in cities_counts.items()}
cities_to_do ← find_cities_standard(driver)
```

In the second part, we apply the formula introduced in the previous paragraph to determine the preferred cities for the driver.

Algorithm 8 Find_favorite_cities (second part)

```
result_dict ← {}
for key in frequent_cities_actual do
    to_do ← cities_to_do[key]
    done ← frequent_cities_actual[key]
    result_dict[key] ← (done - to_do)) + (done / 2)
end for
```

Finally, we compute the average length N of the routes taken by the driver in question and return the top N cities with the highest scores.

Algorithm 9 Find_favorite_cities (third part)

```
n ← calculate_avg_route_length_driver(driver)
result_dict_ordered ← Sort_Dictionary(result_dict)
result_dict_top_n ← Get_top_n_items(result_dict_ordered, n)
return result_dict_top_n
```

The result of this function, combined with the products preferred by the same driver, will be useful for creating the perfect routes associated with the driver.

get_top_items_per_city. : This function is tasked with associating the preferred cities obtained from get_top_items_per_city with the relevant products brought to each city, as obtained from relevant_merch. The number of products for each city is calculated by the avg_n_product_per_city function, which computes the average number of products that the driver typically brings to each city.

Algorithm 10 Get_top_items_per_city

```
top_items ← {}
for city, items in relevant_merch.items() do
    top_items[city] ← {}
    if items is not None then
        n ← driver_habits[driver][city]
        sorted_items ← sort_items_desc(items)
        top_items[city] ← Get_top_n_items(sorted_items, n)
    end if
end for
return top_items
```

get_top_items_by_city_and_quantity. : The final function is dedicated to calculating the relative quantity of products for each city. The goal is to take the average quantity over all instances where that driver brought a specific product to a particular city. From the result of the get_top_items_per_city function, we consider each city-product combination brought by the driver and find the average quantity brought, which replaces the score. The average_quantity_of_product function is used for this substitution.

4 EXPERIMENTAL EVALUATIONS

4.1 Dataset Generation

The dataset_generator file encapsulates the logic for generating actual and standard routes. The parameters to be determined include n_dataset (the number of datasets desired), n_standard_routes (the number of standard routes per dataset), step (introduced in order to make the size of the datasets incremental), and n_actual (the number of actual routes generated from each standard route). Additionally, the variables "groceries" and "cities" are two lists indicating the products managed by the company and the potential destinations for each trip, respectively.

4.1.1 ***Standard Routes***. : The generation of standard routes is carried out by the **"standard_routes_generator"** function, which takes as input the lists of cities and products, along with the desired number of standard routes to create. Each standard route can potentially contain an infinite number of trips, but for feasibility reasons, we have decided to limit this number between 2 and 10, maintaining the number of standard routes chosen by the user in the previous function.

For each trip, two cities (one as the starting point and another as the destination) are randomly assigned, along with a random number of products ranging from 1 to 5. Additionally, each product is assigned a quantity ranging from 1 to 50. It is important to note that the number of products per trip and the quantity assigned to each product could theoretically be infinite, but for practical purposes, these values are constrained within those ranges. Finally, each standard route is assigned a unique identifier (s0, s1, ..., sn).

4.1.2 ***Actual Routes***. : The **"actual_routes_generator"** function is responsible for generating the actual routes. It takes as input all the standard routes and the range of "n_actual" (lower and upper limits for the number of actual routes for each standard route).

For each standard route, the actual number of associated routes is calculated as a random number X within the specified range of "n_actual." Subsequently, the function duplicates the standard route X times, modifying certain fields such as the identifier ("id"). Additionally, the function introduces new fields, namely "driver" and "sroute" (representing the driver assigned to the actual route and the identifier of the associated standard route, respectively) in order to respect the well-defined format provided by the company.

The **change_city** function is designed to modify a given actual route by randomly altering its composition through operations such as changing, removing, or adding cities along the route. The input parameters include the existing actual route, a list of available cities, and a list of groceries. The function determines the number of trips in the route and randomly selects a subset of trips to undergo modification. For each selected trip, a random operation (change,

remove, or add) is performed. In the case of a change operation, a city in the selected trip is substituted with another city from the available cities, ensuring that the new city is not already present in the route. If the operation is removal, either the starting point, destination, or a random city in the trip is removed. For addition, a new city is introduced between the starting and ending points of the selected trip, along with a randomly generated list of unique groceries and their quantities. The function ensures that the modifications maintain the integrity of the route, preventing duplicate cities in the same trip.

The **change_actual** function operates on a collection of actual routes. The function iterates through each route and, for each merchandise item in the route, randomly selects an operation to modify its properties. The operations include changing the item itself, modifying its quantity, removing the item, adding a new item, or perform no changes. In the case of changing the item, the function selects a new item not already present in the trip from the available set of groceries. If the operation is to change the quantity, a new random quantity within a specified range is assigned to the existing item. For removal, the function identifies items to be removed and keeps track of them in a separate list, while for addition, it introduces new items with random quantities. After iterating through the merchandise items, the function then removes the marked items and updates the route's merchandise dictionary with the newly added items. This process is repeated for each route in the input collection.

4.2 Evaluations

During the testing phase, we observed that the code exhibited considerable slowness, leading us to investigate potential optimization strategies. One improvement made involved transitioning from a fixed to a dynamic length for one-hot encoding.

Originally, a file containing all products supplied by the company needed to be loaded to establish a fixed one-hot encoding length, set as the size of the products. The "groceries" value used by the function to create the one-hot encoding was predetermined and fixed. Consequently, if there were 3000 products in groceries.json, the length of each one-hot encoding would be 3000, with a value of 1 only in the positions corresponding to the products present in that route.

In scenarios where route comparisons involve routes with a limited number of products, the resulting one-hot encodings consist of a sparse distribution of 1s and an abundance of 0s in both cases, leading to unnecessary computational overhead due to the fact that the function uses xor and or operations, which ignores cases where both bits are set to 0. For instance, (1 1 0 0 0 0 0 ... 0), (0 1 1 0 0 0 0 ... 0) would yield the same Hamming distance even without considering the trailing 0s as you can see in the example below. This represented an inefficient use of memory and processing time.

A {'from': Ala, 'to': Avio, 'merchandise': {'Milk': 1, 'Oranges': 5, 'Strawberries':10}}
B {'from': Ala, 'to': Avio, 'merchandise': {'Milk': 1, 'Oranges': 11, 'Strawberries':5, 'Bread': 3}}

List of merchandise in this route: Milk, Oranges, Strawberries, Bread.

Old Method:

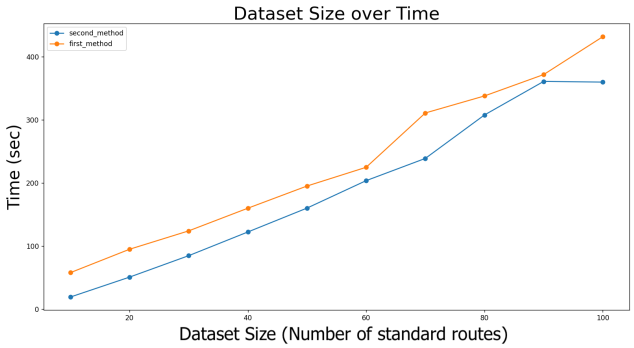
A = (one_hot : 1110000...0)
B = (one_hot : 1111000...0)
xor = 1 , or = 4
hm_dist = 1/4 = 0.25

New Method:

A = (one_hot : 1110)
B = (one_hot : 1111)
xor = 1 , or = 4
hm_dist = 1/4 = 0.25

In the refined approach we developed, the method dynamically calculates the length of the one-hot encoding specifically for each comparison, enhancing efficiency. The "groceries" value used by the function is derived uniquely for each scenario. It's obtained by taking the union of the sets of products from the two routes under comparison. Consequently, if route1 has 7 unique products and route2 has 3, the one-hot encoding will extend to a total length of 10. This strategic method effectively eliminates the necessity of allocating a vector with a size equivalent to the entire range of groceries utilized by the company. Instead, it smartly focuses only on the products pertinent to the routes in question. By doing so, this solution is expected to yield superior results in terms of both memory utilization and processing speed.

To validate our hypothesis about this advanced method, the subsequent graph provides empirical evidence. It contrasts our newly proposed solution (represented by the blue line) with the previous method (illustrated by the orange line), highlighting the enhanced efficiency of our approach.



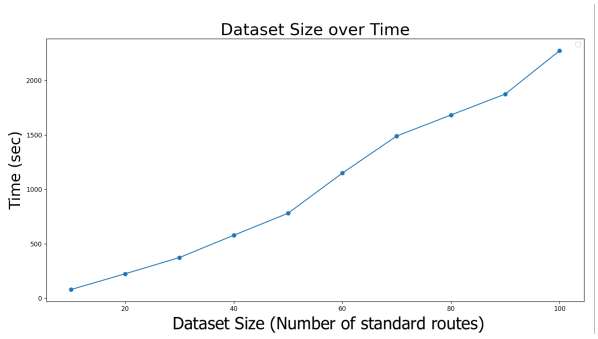
In order to rigorously test our solution, we constructed 10 distinct datasets. Each dataset was designed with an incremental increase in the number of products per trip, while maintaining a constant number of standard routes and a fixed count of actual routes for each standard route. We then applied both the first and second methods to these datasets, recording the execution times for each. The results, as illustrated in the previously graph, clearly demonstrate that the second method consistently surpasses the first in efficiency across the spectrum of all 10 datasets.

Our evaluation of the solution encompasses two distinct types of assessments. The first assessment focuses on the efficiency in terms of time, particularly with respect to the dataset's size, which is determined by the number of standard routes. The second assessment is aimed at evaluating the overall performance of our solutions.

For a detailed analysis of our solution's performance, we structured the datasets in a specific manner. We standardized the number of actual routes per standard route and progressively increased the number of standard routes by increments of 10 for each dataset (starting from 10 standard routes in dataset0, increasing to 20 in dataset1, and so forth, up to 100 standard routes in dataset9). This methodical approach ensures a uniform calculation of time per dataset, effectively eliminating any potential bias that could arise from varying factors such as the number of trips per route, the quantity of merchandise per trip, and most importantly, the number of actual routes per standard route. The latter is especially critical as it significantly impacts the time required, given that the program's calculations adhere to the binomial coefficient formula $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ in this way:

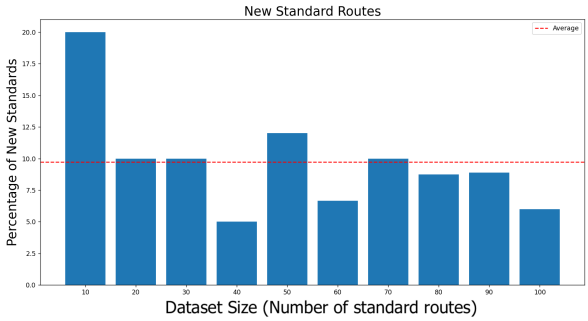
Concerning **solution/PUNTO_1.py** (Creation of new recommended standard routes), for each standard route, n equals the number of actual routes based on that standard route + 1 (including the standard route itself), while $k = 2$.

Regarding **solution/PUNTO_2.py** (Creation of the ranking of drivers' preferences), for each driver, n equals the number of standard routes plus the number of actual routes executed by the driver, and $k = 2$.



The upper graph clearly delineates that, in line with our initial expectations, there is a noticeable trend where increasing the number of standard routes correlates with an escalation in execution time. This pattern is consistently observed across all three points, namely PUNTO_1.py, PUNTO_2.py, and PUNTO_3.py, due to the fact that each of these elements systematically traverses the entirety of the standard routes.

Ultimately, we are now in a position to present the calculated percentage of newly recommended standard routes to the company. As detailed in the preceding paragraphs, these recommended routes might include a combination of those predetermined earlier or entirely new ones, all selected based on the fundamental principle of similarity.



The graph illustrates datasets with a progressively increasing number of standard routes alongside the percentage of standards replaced by new routes. This demonstrates that updated new routes outperform the previous ones as they are more "similar" to what the assigned drivers are actually doing. We observe an average increase of 10%, which could signify a significant improvement for a company.

4.2.1 Evaluations of final distance route. The validity of our solution heavily relies on the designated distance function. If this function is not sound, all the efforts invested in our work become compromised. Therefore, we present a set of intuitive examples to demonstrate the validity of our distance function. Firstly, we showcase that two identical route examples result in a distance value of 0.

```
"route1": [
  {"from": "Trento","to":"Cuneo",
    "merchandise": {"Pineapple": 47, "Grapes": 1}},

  { "from": "Cuneo"," to": "Bolzano",
  "merchandise": {"Pineapple": 47, "Strawberry": 12 }}
  ]

"route2": [
  {"from": "Trento","to":"Cuneo",
    "merchandise": {"Pineapple": 47, "Grapes": 1}},

  { "from": "Cuneo"," to": "Bolzano",
  "merchandise": {"Pineapple": 47, "Strawberry": 12}}
  ]

Final distance route = 0
```

Furthermore, we can illustrate how a slight variation in one of the two routes leads to a proportional change in the calculated distance.

```
"route1": [
  {"from": "Trento","to":"Cuneo",
    "merchandise": {"Pineapple": 47, "Grapes": 1}},

  { "from": "Cuneo", "to": "Bolzano",
  "merchandise": {"Pineapple": 47, "Strawberry": 12 }}]
```

```
  "route2": [
    {"from": "Trento","to":"Lecce",
      "merchandise": {"Tomato": 47, "Grapes": 1}},

    { "from": "Lecce", "to": "Bolzano",
    "merchandise": {"Pineapple": 47, "Strawberry": 12}}
    ]

Final distance route = 0.395
```

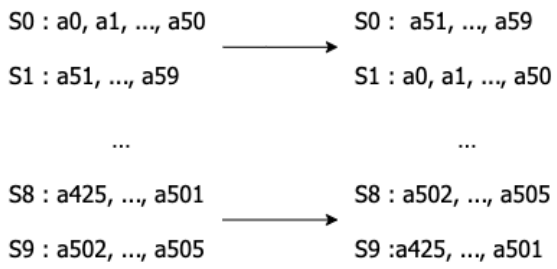
As a final example, we demonstrate that if one of the two routes is altered entirely, the total distance is significantly greater compared to minor changes as in the previous case.

```
  "route1": [
    {"from": "Trento","to":"Cuneo",
      "merchandise": {"Pineapple": 47, "Grapes": 1}},

    { "from": "Cuneo", "to": "Bolzano",
    "merchandise": {"Pineapple": 47, "Strawberry": 12 }}
    ]

  "route2": [
    {"from": "Mantova","to":"Milano",
      "merchandise": {"Bread": 47, "Coca-cola": 1}},

    { "from": "Milano", "to": "Roma",
      "merchandise": {"Egg": 47, "Cherry": 12}}
    ]

Final distance route = 1.0
```

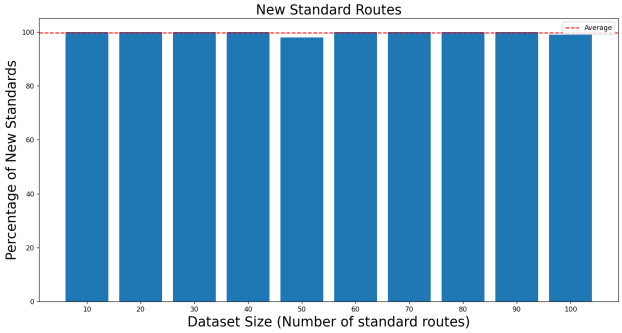
Certainly, numerous intermediate examples could be explored to further illustrate the validity of the three main components of the total distance function, namely qnt_distance, hm_merch, and js_city. To avoid unnecessary details, we have presented the most significant and apparent examples, which should suffice to demonstrate the validity of the entire function.

4.2.2 Evaluations of Point 1 (Creation of new recommended standard routes). A 10% increase is quite significant for any company, as even a 10% annual improvement can lead to millions in savings. This figure is attributable to the fact that the actual routes are based on the standard ones. Indeed, it's important to remember that actual routes are created by modifying cities, products, and quantities from the reference standard route (potentially everything, although the likelihood is low).

To assess the effectiveness of our solution, we conducted a test by changing the reference standard routes. This altered the comparison method, which previously involved comparing s0 with all its related actual routes, s1 with all its related actual routes, and so on. Now, the comparison is made between standard routes and actual routes not related to them through permutations, as shown in the figure below.



This example, performed on datasets 10-19, simulates a scenario where no driver is following company directives (it is worth remembering that even before, they could change anything within an assigned route, but the probability was lower).



As expected, if the drivers completely disregard the company directives, the percentage of new standards replacing the previous ones increases approximately to 100%.

4.2.3 Evaluations of Point 2 (creation of the ranking of drivers' preferences). To test point 2, we can extract the best 5 standard routes of a driver from any dataset available in our list. For simplicity, let's choose dataset 0 and driver A. As shown in the following code, the standard route s4 is the "worst among the top 5", meaning it is the least similar among the most similar standard routes.

```
'driver': 'A', 'routes': ['s8', 's2', 's6', 's5', 's4']
```

One way to evaluate our solution is to replicate the content of our dataset 0 into dataset 20, modifying some actual routes taken by driver A and making them more similar or ideally identical to standard route s4. In this manner, if the solution is correct, s4 should improve its position in the final ranking.

```
'driver': 'A', 'routes': ['s4', 's8', 's2', 's6', 's3']
```

As observed, s4 has significantly improved, reaching the top position. Therefore, our solution is considered valid.

4.2.4 Evaluations of Point 3 (Creation of perfect route for each driver). To test the functionality and validity of our third point, we modified the actuals in dataset 9 to assign a preference to the

specific driver. This was done to observe whether this preference would manifest in the "perfect route" for the driver.

We set a strong preference for driver A, fixing the behavior to 'Every time he needs to go to Verona, he goes to Bolzano instead'. This preferecnce was made by applying a simple instruction in the code during actual routes generation.

```
if city = 'Verona' and driver = 'A' then
    city ← 'Bolzano'
```

This demonstrates that the driver prefers Bolzano over Verona, which is considered a city not to his liking. In addition to the substitution, we ensured that Bolzano was the only "selectable" city in case of adding a destination.

["Ancona", "Arezzo", "Asti", "Bari", "Biella", "Chieti", "Como", "Cuneo", "Enna", "Fermo", "Foggia", "Genova", "Latina", "Lecce", "Lecco", "Lodi", "Lucca", "Matera", "Milano", "Modena", "Napoli", "Novara", "Nuoro", "Padova", "Parma", "Pavia", "Pisa", "Prato", "Ragusa", "Rieti", "Torino", "Trento", "Udine", "Varese", **"Verona"**]

It is important to note that the city of **"Bolzano"** is not incorporated into the list of cities utilized by the scripts generating standard routes. Consequently, whenever the city of Bolzano appears within the actual routes, it is due to the driver's discretion.

Despite Bolzano not being a designated destination, it appears in the generation of the 'perfect route' for Driver A (The json below is an example of generation of the perfect route for the driver). This is caused by the fact that the city represents one of the driver's preferred destinations.


```
"driver": "A",
"route": [
  {"from": "Trento","to":"Cuneo",
    "merchandise": {"Ananas": 47,"Uva": 1}},

  { "from": "Cuneo"," to": "Bolzano",
    "merchandise": {"Ananas": 47, "Fragola": 12,
    "Arancia": 44, "Banana": 4 }},

  { "from": "Bolzano", "to": "Latina",
    "merchandise": { "Ciliegia": 35, "Pesca": 3,
    "Kiwi": 23 }}]
```


This test was conducted to validate the impact of driver preferences on perfect route generation, demonstrating that the introduced preferences influenced the generated routes, even when the specified destination was not initially included in the set of possible destinations. This supports the effectiveness of our proposed approach in accommodating driver preferences within the route optimization system.

5 CONCLUSION

In this project, we conducted a detailed examination of driver route optimization in logistics, employing various data mining techniques. We initiated our study by analyzing route similarities, applying modified versions of Jaccard similarity, Hamming distance and quantity distance calculations to assess alignment between different routes. This involved a deep dive into the nuances of each route, examining both the sequence of cities, the types of merchandise transported, and the quantity.

The creation of new recommended standard routes was a key aspect of the problem. By introducing a dynamic approach to aligning company guidelines with actual route performance, our system identifies opportunities for route optimization. Our solution, which includes route analysis and distance computations, facilitates informed decision-making, offering valuable insights into which standard routes could be replaced with more efficient alternatives. Furthermore, the prioritization of standard routes for minimal diversions acknowledges the importance of individual driver preferences. By considering actual routes taken by drivers and evaluating their deviations from standard routes, the system ranks standard routes, providing a personalized order that minimizes diversions and enhances overall operational efficiency. By understanding driver preferences in terms of visited cities, delivered products and quantities associated, the system aims to construct routes that align with individual driver habits. This personalized approach not only enhances driver satisfaction but also contributes to overall system efficiency.

Our project stands as a testament to the potential of integrating data-driven strategies in logistical operations. The insights gained from this study underscore the importance of adapting to the dynamic nature of driver behaviors and preferences in route planning. The methodologies we employed demonstrate how data mining can be effectively utilized to improve operational strategies in logistics.