POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Optimal Vehicle Assignment for a Dynamic Ride-Sharing Service and Spatiotemporal Taxi Demand Forecasting

Supervisors
prof. Giuseppe CALAFIORE
prof.ssa Marina MONDIN
prof. Fereydoun DANESHGARAN

Candidate
Matteo PEZZETTA

Academic Year 2019/2020

Acknowledgment

I want to start by thanking my mother and my father who have supported me ever since the beginning of my experience at the Politecnico di Torino, whatever happened and whatever it took. Their critical mindset and passion for arts has shaped my behavior and taught me to focus on my goals. Thank to my brother Massimiliano, who has always encouraged me to experience everything new. You have always been an example to follow. Thank you to my sister Enrica and her kids, you have always been a shelter and a place of love. I am eternally grateful to my whole family.
Thanks to my friends for always being there, for always believing in me and in what I was doing. Thank you Laura, Beatrice, Chiara, Alvise, Gian Mattia, Nicolò, Shadra. Your support is essential to keep the mind clear and exchange ideas to grow our interests. Also, I want to thank my university mates and a special gratitude goes to Edoardo, my first friend in Turin. With your kindness you have always made me feel home. We have always been together, from our first day at the Politecnico until our last one, in California. Thank you to Alessandro, Andrea, Davide, Mattia and everyone who helped me during my university career. Luca, you are the one with who I challenged the most. Thank you for sharing with me your determination and your strength. I will never forget what we learned together, what we achieved and our precious experience in Los Angeles. Thank you all, my friends. I hope that our paths can eventually meet again, one day.
I would like to thank also my research supervisors prof. Marina Mondin and prof. Fred Daneshgaran for having given me the opportunity of living this special experience in California.
Thank you to Luca, my best friend. You have been on the front line to encourage me to not give up. Having you on my side has always pushed me to do more and to challenge myself.
Finally, I want to thank Eliza, my wonderful girlfriend. You opened my eyes with your strong personality, you made me experience the city of Los Angeles from a point of view that I would have never had either, you have always been a lovely presence during my studies.
To conclude, I want to say that this has been an incredible journey, but I am excited to move on, again, and meet new people, learn new things to keep growing and being hungry for what is yet to be discovered. If I am happy to change, it is because everyone taught me to never stop and to take advantage of every learning opportunity. This is only the beginning of the next adventure.

I dedicate this thesis and my efforts to everyone I love.

Matteo

Abstract

During the recent years, the transportation of persons and items has changed drastically. Shared mobility systems are now providing flexible public travel modes comparable to private transportation systems at accessible prices. Companies like Uber, Lyft, Postmates are developing algorithms for managing their fleets of vehicles to accommodate ever-growing demands. In 2014 Uber announced UberPool, a ride-sharing service that nowadays accounts for 20% of the total Uber rides. Ride-sharing was born to accommodate the requests of more customers in one single vehicle ride, so that to decrease the service costs both for the providers and the customers. The complexity of the problem derives from the number of constraints influencing the system. The goal is to efficiently manage the fleet, by reducing transportation costs and increasing the occupancy ratio of vehicles, while at the same time satisfying the most customers.
In this thesis, a dynamic ride-sharing algorithm for the assignment of vehicles to customers has been developed together with a demand forecasting algorithm to predict the number of pick-ups requests that will happen in the different areas of a map.
Specifically, the first part of the thesis will deal with the development of the dynamic ride-sharing algorithm.
Our ride-sharing algorithm is dynamic because it can add requests to vehicles when vehicles are already serving other requests, while meeting all the time requirements of the customers on the departing and arrival times.
Dynamic systems must be able to manage and process data that change over time, and to compute assignments under strict temporal constraints.
The algorithm works by matching requests using network theory by building a so called shareability graph, that describes which requests can be shared and which vehicles can serve them, depending on the time windows of the different tasks and on the capacities of the vehicles.
Then, on top of this graph, a set of feasible trips is generated. An integer linear programming problem (ILP) will assign vehicles to trips by choosing them from the set of feasible trips. Great emphasis will be given to the execution time of the algorithm, since the future scope is to have it running on a real application for dynamic vehicle assignment, which will need to manage real-time data from incoming requests of customers.
In our case, since it is not needed to assign all requests, imbalance between the distributions of the demand and of the vehicles can leave some customers unserved. To try to serve previously unassigned customers a rebalancing phase is designed where idle vehicles, if present, are matched with requests that are still pending via the solution of a linear programming problem. Indeed, it is imperative for transportation companies to prevent imbalances in the distribution of the fleet in the areas covered by the service by predicting what will be the demand for taxis in the different regions. To this aim, in the second part of the thesis we build a model to predict the hourly number of pickups in different areas of the map. For this thesis, we have used the dataset of the yellow taxi trips record data of New York City from three months of years 2015 and 2016. The data have been preprocessed so that to generate a dataset that can give spatiotemporal information of the pickups distribution. Also, weather conditions have been added as an additional feature to the dataset. Finally, we have made a comparison between naïve prediction approaches and more advanced techniques like gradient boosting and LSTM recurrent neural networks, investigating what could be the advantages of more complex algorithms.

Contents

List of Tables

List of Figures

Part I

Optimal Vehicle Assignment for Dynamic Ride-Sharing

Chapter 1

Introduction

With this thesis we want to build an algorithm for a dynamic ride-sharing service, where vehicles will be optimally assigned to rides so that to minimize the total time travelled by vehicles while at the same time trying to maximize the number of tasks served. To this aim, it is convenient that vehicles (or robots) can work on more tasks at the same moment. This is what we call ride-sharing, that is when rides are sharing the same vehicle in a so called *shared ride*. Therefore, the problem becomes more complex since we will have to verify fot the possibility for multiple tasks to be on board of the same vehicle given their position in space, their time requirements, the capacity and state of the robots. Moreover, it is called dynamic because there is the possibility for tasks to be added to existing rides that are already on board of vehicles. The dynamism of the algorithm will require the computation to be able to manage real-time data that are describing the status of tasks and vehicles, which are changing over time. Ride-sharing belongs to the wider class of problems called multi-robot task-a.location (MRTA) that is a subfield of the task allocation problems. We want to spend part of the introduction of the thesis to define what is task allocation and what is multi-robot task allocation, since they are needed to later discuss about dynamic ride-sharing. Indeed, all these classes of problems share most of the aspects and the way the problems are handled are similar.
Task allocation is of huge importance in the transportation industry, warehouse management, reconnaissance missions, mapping and missions in unknown environments and many other fields where it is convenient to use more than one robot to complete the task. Also, in the autonomous cars industry, having vehicles to exchange information about the environment, where each of the vehicle is required to monitor a certain portion of the space, is convenient since managing the whole set of inputs by one vehicle would be computationally impossible.
Therefore, some cooperation is needed between the robots of the team, so that to have them working for the sake of a global goal, that a single robot cannot accomplish.

In recent years, research have begun to investigate problems involving multiple, rather than single, robots. Since the beginning the complexity of the problem has grown, due to the added complexity of the tasks that teams would need to accomplish [40]. Researchers added features like time windows for tasks, spatial constraints and probabilistic and stochastic models to handle uncertainty [72].
The diversity of the various applications required also that many different approaches were studied and used nowadays. This also increases the complexity of studying such a wide topic. From the different approaches a huge classification arises.

1.1 Terminology

To begin speaking about task allocation, some definitions are needed for what concerns the terminology in this field. [72].

- A robot is an autonomous agent responsible for performing some actions. Robots in Multi-Robot Task Allocation (MRTA) are modelled as holonomic or point robots, that is, as simple as possible, since the focus of the research is not to study low level behavior of the robots and how they interact with the environment to do the task;
- A *team* is a set of different *robots*. They can be of same capabilities (homogeneous robots) or of different capabilities (heterogeneous robots). The whole team is involved in accomplish the global goal;
- A *task* is the action to be performed, also referred as a work unit. In some cases, the task can be split in simpler tasks, that can themselves be assigned to different robots. In other cases, the tasks are part of a more general task, or goal that the team has to accomplish as a whole;
- A *time window* is a time interval related to the execution of the task. It starts at the earliest time a task can start, and it ends at the latest time the task can end. A time window is said to be closed when both the start and end times are given;
- *Synchronization constraints* are restrictions on the way more tasks are related in time to each other. For example, some tasks must start or end at the same time;
- *Precedence constraints* specify ordering in time between tasks. For instance, a task must start when another task already ended. It differs from synchronization by the fact that for synchronization, a specific time is involved.
- A *schedule* is a timetable in which each task has a specific time to start, end or both. In some cases each robot has its own individual schedule, while in others all robots share a single schedule;
- The *makespan* is the time difference between the end of the last task and the start of the first task;
- A *route* is a sequence of location to visit to accomplish one or different tasks. Routes concern spatial distribution of the task or of a set of tasks;
- A *task release* (or *request*) refers to a task becoming available for execution.

1.2 Generic problem formulation

1.2.1 Robots and tasks

We assume there are finite sets of robots and tasks in our problem. Robots (or vehicles) may have a state represented by location, speed, schedule, capabilities specifications.
A task may have states as location, earliest start, latest finish time, duration, cost, size, reward and other possible specifications depending on the problem.
These states should represent entirely the robots and tasks for the specific problem, so that to have the necessary information to solve the assignment problem.

1.2.2 Constraints

Constraints are potentially arbitrary functions that restrict the space of feasible solutions of the problem. They can be present as time windows, so they require the start and the end of the task to lay on a specified time interval.
Ordering constraints specify a dependency between tasks or between different actions of a single tasks (for instance, for pick-up and delivery tasks, the pick-up action must happen before the drop-off action).
In the literature, a general approach to represent tasks is to have them as nodes of a graph, where edges instead, represent relation or precedence constraints between tasks.
Constraints can also be related to the capabilities of the robots, that define which robots are capable of performing which tasks. For instance, the fleet can be characterized by a certain fleet heterogeneity, where some robots have facilities that others have not, and this will end up in them being able to operate in conditions the other robots cannot [57].
Capacity constraints instead, define how many tasks a given robot can perform at the same time. When robots are cars, for example, they have a limited seats capacity so no more than a certain number of passengers can fit in the vehicle.
Temporal constraints are of huge importance in the field of task allocation, because of implications due to the nature of the problem (like the sequence of actions that needs to be observed for the fulfillment of the goal) and also because of requirements needed so that to guarantee a good quality of the assignments in terms of timing of task execution and possible delays with respect to the tasks' deadlines.

Many temporal relationships are possible between a couple of tasks [4], as shown in Figure 1.1. In this figure we can see all the possible temporal combinations between two tasks.
These relationships can be used to define ordering and synchronization constraints between the tasks and can also be used as criteria for the combination of tasks when tasks are grouped in sets.

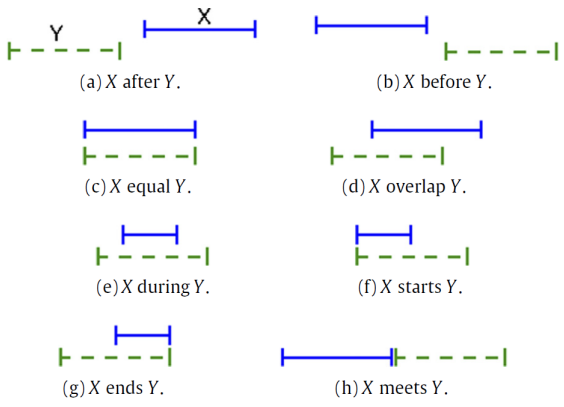


Figure 1.1: Possible time order and synchronizations between two tasks.

Time windows

A time window is a temporal interval constraint on the start and the finish time of a task. The task has usually a lower bound that is the earliest starting time, and an upper bound that is the latest finish time. For more complex formulations we can have latest

start time and earliest finish time. We can therefore define:

- Start time: is the time from when the task can be executed, the beginning of its time window. Before this time, the task does not exist;
- End time: is the last moment in time an agent can work on the task. It marks the end of the time window of the task. After this time, the task does no longer exist;
- Duration: is the time it takes an agent to do the task, once it is in the task location. Its length is at most the length of the task time window;

A task is doable if the agent can reach it no later than the interval given by the *end time* minus *duration*: otherwise the task cannot be completed before its end time [41].
Task allocation problems with time windows are generally $\mathcal{NP}$-hard [88] and finding feasible solutions is $\mathcal{NP}$-complete [84].
Precedence and synchronization constraints impose partial ordering between tasks, which can be used to eliminate candidate solutions that violate the ordering.
Instead, tasks with time windows are independent of each other and can be performed in any order, as long as the robots can reach the tasks and execute them within the time windows of the tasks.
Time windows can be used to describe many different types of temporal constraints between the tasks, so that in the scheduling, tasks are linked to each other depending on their own time windows.

Hard and soft temporal constraints

When speaking about temporal constraints, there could be other additional information we can give on the kind of temporal constraints the robots have to deal with. For instance, we can have hard and soft temporal constraints. Soft temporal constraints allow the assignment to violate the temporal constraints, knowing though that the quality of the assignment will decrease as the constraint is violated. When the outcome of a task depends softly on its time window, the later the task is fulfilled, the less useful the result of the operation will be.
Instead, when we have hard temporal constraints, the accomplishment of the task has no longer any value if the task is completed after its deadline.
Sometimes, the unfeasibility of a problem can be given by constraints that are too tight so that we have no space to find for feasible solutions. In these cases, some applications allow to soften some temporal constraints (especially when dealing with hard temporal constraints), in order to define a set in which feasible solutions exist. Soften the temporal constraints can also speed up the computation of the assignment and improve the quality of heuristic search.

1.2.3 Optimization objectives

When tasks have been defined and the relationships between the tasks have been analyzed, an assignment to vehicles has to take place. Applications of MRTA/TOC (Multi-Robot Task Allocation with Temporal and Ordering Constraints) at the end require the robots to achieve the goal minimizing (or maximizing) a certain cost function. To this purpose

an optimization problem for the minimization or maximization of a certain cost function is solved. Costs can be in the form of temporal measures (time spent, or delay on task execution) or spatial measures (travelled distance) or related to the number of tasks executed, depending on the assignment quality criteria involved in the application. There can be cases where we have single or multiple objectives. Multi-objective is used when we want to find a trade-off between many criteria with which we want to evaluate the quality of our assignment. For example, it could be that we want to minimize the total distance travelled by the robot fleet while also minimize the time from the end time required by the task and the actual time the task is completed by the robot (in case of soft temporal constraints).
Objective functions are functions that related the variables of our problem to the cost of the assignment. They may describe, as previously mentioned, the total cost as the time delay with respect to tasks deadlines, the total travelled distance, the total time distance, or to maximize a certain quantity that could be a score, a reward or the number of satisfied tasks in the assignment.
The general framework will generate a solution in which the given robots are efficiently allocated to the given tasks in such a way to maximize the overall performance of the whole assignment problem.

We want to show now a very simple example of task allocation, referring to the case of ST-SR-IA that is Single-Task robots, Single-Robots tasks and Instantaneous Assignment (no schedule for future assignments).
This problem can be represented by a linear assignment problem, where N is the number of robots and M is the number of tasks to be assigned [95].
Following, the mathematical model for such case from the combinatorial optimization literature:

$$\max_{x} \quad \sum_{i \in N} \sum_{j \in M} u_{i,j} x_{i,j} \tag{1.1}$$

$$\begin{aligned} \text{subject to:} \quad & \sum_{i \in N} x_{i,j} = 1, \ \forall i \in N \\ & \sum_{j \in M} x_{i,j} = 1, \ \forall j \in M \\ & x_{i,j} \in \{0, 1\} \end{aligned} \tag{1.2}$$

The objective function in this case, that is the most general case, is a matrix of dimensions $N \times M$ that associates to every possible robot-task assignment x a utility (or score) u. The larger the score, the greater is the productivity.
In this case a utility function u is used, but it is common that a cost function c is used instead. In that case the problem is formulated so that to minimize the total cost of the assignment.
An example of the minimization case is when $c_{i,j}$ represents the distances in time or space between each robot and each task. If this is the case, we want that the optimal assignment is such that the total time spent by the robot is the less possible, so that to save energy and have an efficient system.

For a feasible solution to this problem, the number of agents N must be equal the number of tasks M. In the case they are not equal we can add as many "dummy" agents or tasks as needed. These dummy agent/tasks must have very low utility values with respect to all tasks/agents in the system [57].
The linear assignment problem can be solved in polynomial time with algorithm such as the Hungarian algorithm [58].

Utility

As said before, being this an optimization problem, we are trying to define a feasible asignment of tasks to the robots that optimizes some objective. This objective can be described as an utility function. To better understand the concept of utility we try to give a definition that is taken from [40].

Given a robot r and a task t, if r is capable of executing t, then one can define, on some stadardized scale, Q_{rt} and C_{rt} as the quality and cost, respectively, expected to result from the execution of t by r. This results in a combined, utility measure:

$$U_{rt} = \begin{cases} Q_{rt} - C_{rt} & \text{if } r \text{ is capable of executing } t \\ -\infty & \text{otherwise} \end{cases}$$

There are problems, where the agent-task utility is independent of the other utilities between the same robot and the other tasks. This happens for example in the case a fleet of robots is required to pick-up some objects scattered on a map, and bring them to their starting location. Every robot will departure from a different location and it is required to bring the object to that location after having picked it up.
Therefore, we can define a value Q_{rt} of the object and a cost C_{rt} for that object t relative to a specific robot r. The cost can be representative of the time required to complete the task or of the energy consumption of the robot.
In this case we can say that the utilities of the different objects relative to robot r are independent of each other, since the robot has to go back to its starting location after the object is picked-up.
The global optimal solution to this task allocation problem for the entire fleet would be to allocate each robot to the most convenient object.
Now, if we consider a different scenario in which the robots of the fleet have the capability to carry more than one object at a time, they have the possibility of taking one object first and then heading to other objects to pick them up too.
For example, if we consider robot R_1 and objcets T_1 and T_2, and the algorithm knows that the distance to travel between the two objects is smaller than the distance to go from object T_1 to the starting location of R_1 and again going from starting location of R_1 to object T_2, then the robot will travel directly form T_1 to T_2 since it is more convenient.
This is the case where the two utilities $U_{R_1T_1}$ and $U_{R_1T_2}$ are related to each other, since there is the possibility of combining the two tasks for robot R_1.
We can formalize this by extending the previous definiton of utility function, as suggested in [57]. The utility function will now not only refer to single robots and single tasks but to subsets of robots and subsets of tasks. $\mathcal{R}$ represents a subset of robots while $\mathcal{T}$ represents a subset of tasks.
The utility function will be:

$$U_{\mathcal{RT}} = \begin{cases} Q_{\mathcal{RT}} - C_{\mathcal{RT}} & \text{if } \mathcal{R} \text{ is capable of executing } \mathcal{T} \\ -\infty & \text{otherwise} \end{cases}$$

For each subteam of agents we can define an *effective utility*, ${}^{e}U_{rt}^{\mathcal{RT}}$, for an agent $r \in \mathcal{R}$ and task $t \in \mathcal{T}$ such that:

$$U_{\mathcal{RT}} = \sum_{r \in \mathcal{R}} \sum_{t \in \mathcal{T}} {}^{e}U_{rt}^{\mathcal{RT}} \tag{1.3}$$

Depending if the agent-task utilities are independent or dependent, we can define the *effective utility* as:

- ${}^{e}U_{rt}^{\mathcal{RT}} = U_{rt}$ in the case of independent utilities;
- ${}^{e}U_{rt}^{\mathcal{RT}} \neq U_{rt}$ in the case of interrelated utilities;

An example of a problem with in-schedule dependencies, is the case of a fleet of vehicles that has to pick-up different customers and there is the possibility that more customers are on board the vehicle at the same time. This is called *ride-sharing* and it is a problem where the utility function is strictly related to routing costs.
When it is possible to travel from every location to every other location in the map, we can represent the problem as a Multiple Traveling Salesman Problem (m-TSP) that is a generalization of the Traveling Salesman Problem (TSP) [12].
m-TSP requires that all salesmen (robots) start from the same location, but in the literature we have generalizations called Multi-Depot Multiple Traveling Salesman Problem, where robots can depart form different locations.
If the robots have to pick-up objects, and can collect more than one object but they have a limited capacity, which means that they can carry a maximum number of objects, we call the problem a Capacitated Vehicle Routing Problem [92].
One application of the vehicle routing problems is the transportation of passengers or items.
The assignments have to be made taking into account the resource availability of each agent. These problems are derived from the classical Assignment Problem and by taking into account these capacity constraints they are also called Generalized Assignment Problems (GAP) [3].
Usually, solving a vehicle routing problem involes that a feasible solution is found that meets all the requirements given by customers and vehicles, while a global transportation cost is minimized. The majority of this problems are solved through an integer or mixed integer programming problems with many constraints depending on the requirements of the system.

A simple formulation of the problem with capacity constraints is the following:

$$\min_{x} \quad \sum_{i \in N} \sum_{j \in M} c_{i,j} x_{i,j} \tag{1.4}$$

$$\text{subject to:} \quad \sum_{i \in N} x_{i,j} = 1, \ \forall i \in N$$
$$\sum_{j \in M} d_{i,j} x_{i,j} \leqslant b_i, \ \forall j \in M \tag{1.5}$$
$$x_{i,j} \in \{0, 1\}$$

Where $c_{i,j}$ is the cost of assigning task j to agent i, $d_{i,j}$ is the resources needed by robot i to perform task j, b_i is the resource capacity of agent i (with $i \in \mathcal{I}$). x_{ij} takes value 1 if job j is assigned to agent i, and 0 otherwise.

We can notice that the constraint from 1.5 that force every robot to be assigned only one job is not present anymore. Instead, there is a set capacity constraints that limits the robots to do only a certain number of jobs depending on the capacities of the jobs (energy or number of items that are taking) that are specified by b_i.
In this case we are no longer speaking of utilities but the objective function is characterized by a cost function to be minimized.

1.3 Categorization (taxonomy)

To begin with a first presentation of the different task assignment problems, in the literature we find a basic classification that works by categorizing robots, tasks and scheduling along three axis [40].
The first axis distinguishes between single-task robots (ST) and multi-task robots (MT), that is, some robots are capable of executing only one task at a time while other robots can execute multiple tasks at the same time.
The second axis states a distinction between single-robot tasks (SR) and multi-robot tasks (MR), that is, some tasks need only one robot to be executed, while other tasks need more robots to be accomplished.
The third axis instead depict the difference between instantaneous assignment (IA) and time-extended assignment (TA). The former says that the assignment is made at a moment in time and no scheduling for future assignments is of interest, while the TA says that a schedule also for future assignments of tasks to robots is made. With TA we will have robots with queues of tasks that have been assigned. Otherwise, when working with IA, we will know only about the current allocation, with no reference to what is next for each of the robots. In this case, when one or more robots have finished their jobs, there will be the need to run another optimization problem to assign the tasks that are left.

ST-SR-IA problem is an instance of the optimal assignment problem in combinatorial optimization and is the only problem in the space depicted above that can be solved in polynomial time.
MT-SR-IA problem is significantly harder, since robots can execute more tasks simultaneously and therefore many combinations arise [40]. In this case, robots will collaborate to execute tasks. Some tasks may need robots to work in a certain order and robots can also have different capabilities. These problems push the problem to a very high complexity level.

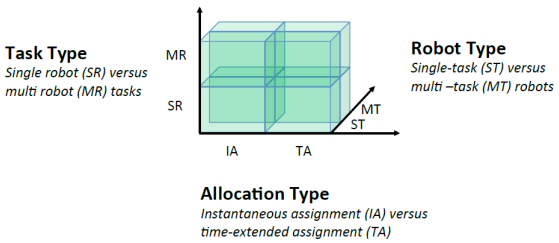


Figure 1.2: Representation of the three-axes classification for multi-robot task assingment typologies.

To focus on what we just mentioned, we can try to distinguish between complexities of tasks. Tasks can be simple tasks, where the task is a basic task. On the other hand, tasks can be seen as compound tasks [57], that are tasks that can be decomposed into a set of simpler or compound subtasks, with the requirement that there is exactly one fixed full decomposition for the task (like pick-up and delivery can be seen as two separate action while the whole task can be seen as the route form the pick-up point to the drop-off point). Using compound tasks can help in the assignment problem when different robots have to work on the different small tasks or when we want some tasks to be executed in a certain order.
For compound tasks, task allocation can be preceded by task decomposition, during which a compound task is broken up into several simple tasks. On the other hand, multiple elemental actions can be compounded into more complex tasks, so that the sequence of actions is seen as a single task.

1.3.1 Different problems categorization

We want now to go deeper into the categorization of the problems, trying to classify them for what are the constraints of the problem and for what is the goal of the task assignment to the robots.
We especially focus on highlighting the importance of temporal and ordering constraints among the tasks and to how the inclusion of such constraints can increase the complexity of the task allocation problem [72].
This formulation, which we have already mentioned, is called MRTA/TOC (Multi-Robot Task Allocation with Temporal and Ordering Constraints) and comes as a general case of more specific problems that we are going to briefly introduce.

- *VRPTW (Vehicle Routing Problem with Time Windows)*: it requires solving allocation, routing and scheduling simultaneously. An example is the online pickup and delivery problem with transfers, where a team of vehicles has to pick-up a set of items at certain locations and deliver them to other locations. Usually, VRPTW problems assume that all vehicles start from a single depot point and after the job is successfully completed, they return at their original location.
  VRPTW often assume homogeneous vehicles within the fleet, with respect to their capabilities and capacities.
  This problem formulation, as almost every other vehicle routing problem, is NP-hard and unlikely to be solved in polynomial time [60]. Thus, typically these problems

are too complex to be solved within a reasonable amount of time and heuristic or approximation algorithms become the methods of choice [36];

- *TOPTW (Team Orienteering Problem with Time Windows)*: the goal is to search for control points to visit between the starting point and the destination. One instance of this problem formulation is the Dial-a-Ride problem, that is an over-constrained problem, which means not all the tasks can be satisfied. In this case the goal is to find the subset of tasks that maximizes the total profit.
  The most common example of Dial-a-Ride problem is the door-to-door transportation for elderly people. Many heuristics have been developed during the years and now we can solve Dial-a-Ride problem efficiently also when the number of customers is high [30];

- *JSP (Job-Shop Scheduling Problem)*: the problem is to assign groups of activities, called jobs, to a set of machines with the goal of minimizing the cost of completing the jobs, alone or in combination with other objectives.

1.4 Typical approaches for solving the MRTA/TOC

- *Market-Based Approach*: here the principles of market economy are applied to multi-robot coordination [9] [35].
  Market based approaches are focused on the concept of utility functions. Which can represent the ability of the agents (the robots) to measure their interest in specific tasks for trading.
  Auctions, used to then assign the tasks to a winner robot, can be performed in a centralized or distributed manner. In general auctions are computationally cheap and have reduced communication requirements [9].
  Decentralization of the auction system can help in scenarios where no global state information are available (missing communication for example).
  With this approach, to design vehicle utilities it essential to obtaining desirable collective behavior through self-interested vehicles [6]. The vehicle utility should be aligned with the global utility function in the sense that agreeable assignments between robots should lead to high, ideally maximal, global utility.
  A negotiation mechanism then is needed, so that vehicles reach an equilibrium in the task assignment.

- *Optimization-Based approach*: optimization techniques are applied in order to maximize or minimize certain cost functions. The set of available solutions is restricted by a set of constraints and the optimal solution is chosen within these constrained solutions according to certain criteria.
  Usually optimization approach is used when good knowledge of the environment and good communication between agents is available. Most of the optimization-based approaches make use of centralized algorithms to solve the task-assignment.
  Sometimes heuristic approaches are used to speed up the algorithms when the size of the problem is huge and to solve unfeasibility, because they are proved to provide satisfying solutions.

Chapter 2

Problem definition

2.1 Evolution of pick-up and delivery services and enabling technologies

2.1.1 Pick-up and delivery sevices

Our scenario is composed of a fleet of robots that is thought to be deployed to satisfy a delivery service on a university campus, where customers can specify pick-up and drop-off locations using the delivery service.
This is a typical pick-up and delivery problem, where vehicles have tasks composed of a pick-up activity and a drop-off activity and those activities have to be executed in a specific order, otherwise the task is not realized.
One of the most important realization of the pick-up and delivery problem is the taxi service in big cities like New York pr London where the taxi demand is extremely high (not only on peak hours) and there is the need for a centralized management of the vehicles so that to try to satisfy the incoming demand for service.
The urge for an optimal assignment of vehicles to customers meets the needs of both customers and service providers. Both parties are interested in getting the most valuable result with the less expense, that is: customers want to go from a specific place to another paying the less amount of money possible, while the provider wants to service an high number of requests by making its fleet to travel the less miles possible, so that to save on fuel and time.
Having a service that can optimally manage the many requests that come from the thousands of customers, is a big step ahead with respect to the traditional taxi system, where customers look for free taxis on the street.
Since 1950 the taxi experience has changed little, where people have not imagined there could have been a better way to manage taxi rides and booking them [34].
During the automobility era, private vehicle ownership has been encouraged and welcomed as facilitating the opportunities of those who were forced to take public transports walking from home to the nearest stop.
Lot of effort and investments have been put on the highway infrastructure, so that to create a system ready to accommodate a very large number of vehicles on the roads.
Taxi operators themselves, operated in monopolistic and highly regulated environments, had little interest in investing money to innovate the system, where system drivers are asked to drive around in search for potential customers. In this situation even experienced drivers, who know which areas to go at what time of the day to have the higher chance of incurring in a customer, will consume high amount of fuel in idle travels looking for a

rider. Also, from the rider point of view, it can happen that no taxi is in sight and that waiting times extend to a point that the service is devalued.
All these factors influence the efficiency and equilibrium of the system and can lead to unbalanced taxi distribution and lot of time and energy wasted.
Some of the solutions for crowded environments have been to put customers in queues or to set certain pickup points where customers can look for taxis in an easier way.
Other solutions include to book rides by phone via a customer service, but also this often results in being unreliable. Taxis may arrive early or on late and there is little communication between the rider and the driver until they meet at the pick-up point.
Other issues can be related to language barriers between the customer and the customer service or between the customer and the driver. Pick-up and drop-off locations may be misunderstood and this would lead to low service effectiveness.

2.1.2 The arrival of the app-based tranportation systems

Companies like Uber and Lyft have challenged all the issues presented in the previous section giving customers an easier way to book rides and giving drivers a more efficient way to move riders around, decreasing the idle times between two rides and thus saving time and fuel. This has been possible because of the availability of highly connected devices such as smartphones and the availability of better connections with Internet offering real-time delivery of information and higher connection coverage. Whithin technologies that allowed these changes, there are software that rely on location-aware capabilities such as Global Positioning System (GPS) that is common in every smartphone nowadays [2]. The platforms are designed to provide ride-matching services via smartphone applications differing from early systems that used non-real time services such as internet forums, or telecommunications, where responses were not immediate [34].
Nowadays, great part of the research to try to reduce carbon emissions is focused on looking for solutions at the vehicle level, therefore trying to move from typical ICE (Internal Combustion Engines) to more efficient combustion engine systems (improving fuel economy), hybrid systems where electric motors and ICE collaborate in providing the power to the traction system or fully electric vehicles. Other solutions are trying to move to alternative fuels such as bio-diesel, hydrogen, non-fossil natural gas and others.

Solutions at the system levels are needed also. Some rudimentary solutions are decisions made regarding when and how to reach certain places so that to regulate the traffic and try to decrease pollution.
But one of the main factors influencing the impacts on the environment of the transportation sector is its efficiency. According to [24] the average occupancy of personal vehicles is 1.7 passengers per vehicle. This means that most of the trips made by car have only one passenger (they are the so called SOVs: single-occupant vehicles). This is a problem both for the number of vehicles that are on the street at the same moment, increasing the total amount of emission per person, and because the high number of vehicles can generate congestions, slowing the vehicles and therefore increasing the trip time and so also emissions.
Incentives that have been given to try to push against SOVs and facilitate traffic flow for high-occupancy vehicles includes the introduction of Carpool lanes for example, so to reduce fuel consumption per person [30].
The increased communication between customers has allowed to investigate solutions that

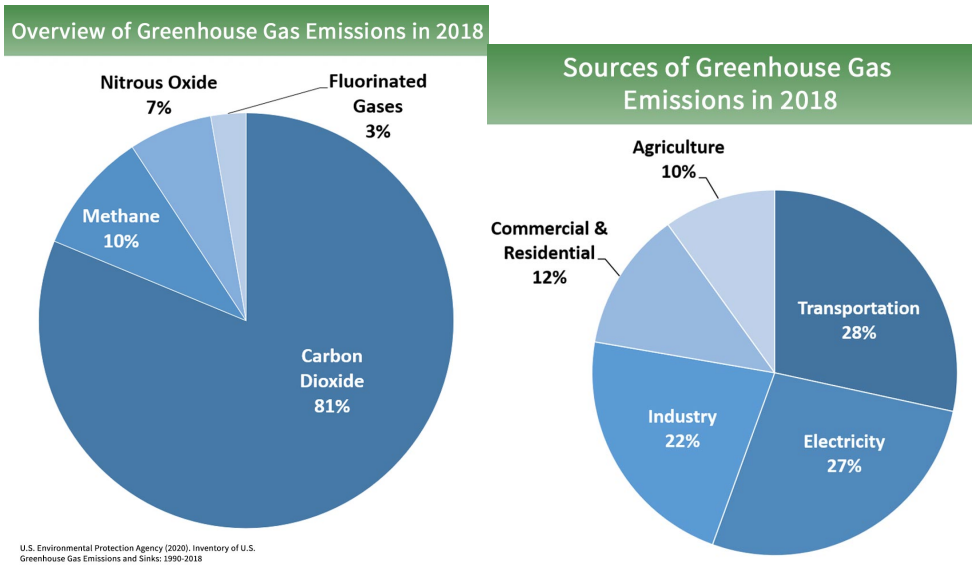
wants more persons to get on board the same vehicle, thus increasing the vehicle occupancy rate in private and public vehicles. This is the reason why ride-sharing methodologies are considered to be potential techniques to highly increase the transportation sector efficiency. The sharing of the ride between customers will match rides between two customers that are strangers to each other and that are eventually going in different places. The spatiotemporal similarity of their routes will determine if they can be merged in the same route. The users have the choice whether to participate or not to the ride-sharing option and what will be the delay interval to which the desired trip will be affected. The user can usually also decide what level of delay to have on its original route.

2.2 Dynamic ride-sharing

2.2.1 Introduction

Before saying what dynamic ride-sharing is, we want to highlight the main assumption of static ride-sharing, so that to be able to catch the difference between the two services.
Static Ride-Sharing assumes that all the information regarding the taxis (locations and capacities) and the customers (pick-up and drop-off locations, time windows) are available before the global assignment problem. In this case the optimal assignment is run once, when all information are collected.
In a dynamic environment taxi requests arrive in real-time and therefore the assignment is usually performed every time a new request arrives.
Dynamic ride-sahring (DRS) is allowing customers to be added to existing trip when the trip is already on its way, so when another customer is already on board of the taxi. DRS differs from regular ride-sharing, known also as carpooling, in that it is an ad-hoc service arranging single shared rides on short notice, or even on the way.
This gives even more flexibility to the service and contribute in maximizing the system efficiency and the benefits of the service. The dynamic ride-sharing system notifies the driver of the possibility to add a new client along the way and makes aware the driver of the additional delay on the current route. Then the driver will have to ask the rider if he/she is okay with picking-up a new rider, to share the ride.
The additional costs regarding the longer route will be charged to the new rider and the customer of the original trip will be refunded of part of what he is supposed to pay, for having accepted a delay on his/her arrival time.
Evidence indicates that specific demographic subgroups are inclined to use Dynamic Ride-Sharing. In particular, subgroups of 18-34 years-olds, have negative attitudes towards private car ownership [70].
These same subgroups represent about half of the dynamic ride-sharing demand.
Some Automobile companies have already indicated the potential economic harm that DRS presents to the car industry.
Dynamic Ride-Sharing is an emerging method of personal mobility based on the traditional concept of ridesharing. DRS algorithms want to match the vehicle drivers with riders in real-time using online technology platforms [2].
In 2016, Uber and Lyft launched for their first time their ride-sharing services that are UberPool and Lyft Line respectively, giving customers the possibility of sharing the ride with other customers that are picked-up and are heading to similar places. Uber states that around 20% of its rides are UberPool rides.
Studies show that switching form traditional taxis to shared autonomous taxis can potentially reduce the fleet size by 59% while maintaining the service level and without

significant increase in wait time for the riders [62].
Also, increasing the average occupancy rate of vehicles, the total travel distance is decreased (up to 55%) and carbon emissions are reduced. In 2018, the transportation sector accounted for 28% of all greenhouses gas (GHG) emissions seen the United States, which is just in front the electric power sector [46].



(a) (b)

Figure 2.1: a. Shows the gases by source for year 2018, b. Shows what are the sources of greenhouse gas emissions for the year 2018 [46].

The more seats are filled during a trip, the less the traffic congestions and therefore the more the environmental benefits. Reduction of traffic congestion will allow vehicle flow to move faster and on average the travel time will become shorter. This will allow customers to spend less time stuck in traffic lines and so to compensate for the delay accepted when joining the ride-sharing ride.
Among the different challenges that have been identified when trying to solve the dynamic ride-sharing problem, the service pricing and passenger matching (to form ridesharing instantaneously) are some who also are taking the great attention from the researchers around the world [39]. Even though, there has been less attention to the pricing problem compared to the ride-matching optimization problem.

The campus pick-up and delivery scenario as a ride-shaing application

Our original scenario is an implementation of the dynamic ride-sharing service, where instead of passengers we deal with items, food and other goods to be delivered in specific locations.
Robots can satisfy more than one task at a time, by picking up different items and delivering them to the corresponding drop-off points.
The primal objective of our problem is to minimize customer's waiting time [24], but it also has enormous potentials for possible impacts with respect to pollution, energy consumption, congestions and costs of the taxi service both for customers and the service provider.
In our specific case, robots are powered by battery packs, so they need to be charged at

charging stations located in specific locations and no direct carbon emissions are of interest. Anyway, it is of high interest to reduce the costs related to the energy consumption of those vehicles, also because robots that are at the charging stations cannot join the fleet and cannot be assigned to trips, thus reducing the availability for robots and increasing the waiting time of customers.
This means that the more energy is saved, the more efficient the system is, both from an energy point of view and from the perspective of the service outcomes.
The advantages of applying vehicle dispatching and ride-sharing strategies to the delivery system are that the average waiting time of users is reduced and the utilization rate of the robots is increased.
When the sharing rate of rides is high (this means the relative number of shared rides versus the total number of rides) the user of the delivery service will face less average delay on his/her delivery. This because the delivery has not to wait for a private dedicated robot to become available but can dynamically be assigned to pre-existing rides if the time windows of the two deliveries are satisfied. Otherwise, the new service request will simply wait for a robot to become available, as in any traditional delivery service.

The goal is to deliver as many items as possible at the lowest cost while obeying a set of constraints, such as time windows and capacities of the vehicles.
Transfers of carried objects can also be introduced to the problem but the complexity grows as the search space of solutions expands exponentially, for a problem that is already NP-hard [26].
The problem of pick-up and delivery task assignment has been solved in literature with many heuristic and metaheuristic algorithms. Also, optimal approaches have been studied but then, when the size of the problem is increased, the algorithm is not able to work dynamically in real-time. Threfore, we need quick assignment strategies to avoid customers to wait long time for their routes to be assigned [48].
The problem is strictly related to vehicle-routing problems [42] [13], whose solution is computationally demanding. Ride-sharing strategies has typically fundamental limitations on scalability as its underlying problem can be reformulated as the travelling salesman problem, which is NP-complete complexity.

2.2.2 Inputs and outputs of the assignment algorithm

Dynamic ride-sharing services should satisfy the requests for rides from customers by trying to merge, when possible, those rides. Also, the vehicles of the fleet have to be assigned following certain criteria. Therefore we can define what are the inputs to the dynamic ride-sharing algorithm and what are the outputs it has to deliver to constitute a usable service.

Inputs

The data that are being fed as inputs to our program are:

- *Tasks*: these are pick-up and delivery tasks, characterized by pick-up location, drop-off location, starting time, estimated arrival time (computed on the route that the single task is requiring), maximum delay on pick-up and drop-off, size of the request (number of passenger or weight of the item), priority level and status (whether it is pending, assigned or picked up). We will call the tasks also *rides*, referring to the dynamic ride-sharing application;

- *Vehicles*: a vehicle is the robot that is assigned to the tasks and it is characterized by: location in space at the algorithm call, capacity and its status (busy or idle);

- *Already existing assignments*: a file containing information about already assigned trips is provided to the algorithm. This list tells what are the vehicle-customer couplings so that is clear what tasks and vehicles are already. This list contains only trips of size one, that are trips composed only of one ride. In this way, if reconsidered in the optimal assignment, it can be assigned other one or two rides.
  This is the core of Dynamic Ride-Sharing, which allows to add customers to routes when routes are already started and some passenger has already been picked-up. In fact, while having one passenger on board, the route can still be modified if the time requirements of an incoming task meet the time windows of the customer on board of the vehicle. This means that until a rider has been dropped-off, there will be always the possibility that his/her driver is going to service other requests.

Outputs

The output of our program will be an optimal assignment of the vehicles to routes, where the routes can contain up to three tasks.
The assignment is of IT type (instantaneous assignment), so every time it generates an assignment that regards only customers that are asking for immediate/short term service and vehicles that are idle at the moment the algorithm is called.
No schedule is produced for future assignments. The algorithm can work in an online manner at the arrival of new tasks or when there are pending tasks and any vehicles becomes idle.
This means that for a vehicle there will not be subsequent assignments, but we will assign to the vehicle only one route (the route can change if other customers are added to the route).
When any of the vehicles drop-off the last rider of a route, the rider (or the robot if it is an autonomous vehicle) will notify the system that its state has changed to *available* and it will enter a list of vehicles that can be considered during the optimization problem.

2.2.3 Clustering the rides

Applications of spatial clustering in transportation have received more attention in the last years.
New strategies are being developed that want the set of tasks to be divided in clusters so that later vehicles can be assigned to the subsets in an optimal way, trying to satisfy the greatest number possible of requests.
After the clustering phase, the different trips will be assigned to vehicles following the criteria that suits the best for the specific application.
A number of clustering algorithms have been proposed and applied in a variety of fields. Some of the typical models that are used for clustering are [38]:

- *Connectivity-based models* or *hierarchical models*: they are based on the idea that objects are more related to nearby objects than those far away. Clusters are therefore created based on the distance between objects in the data space. They should be used carefully when dealing with outliers to avoid cluster merging. To define clusters, we have to decide which linkage criterion to employ to link together data. Then, a distance function will define the size and what partition is best suitable for our application;

- *Centroid-based models*: the centroid of the cluster is representative of a cluster. One of the most popular is the k-means clustering (Lloyd's algorithm), which forms the basis for centroid clustering techniques. The number k of clusters has to be defined in advance and then the k-means clustering will use optimization techniques to find the k centers of the dataset, and will assign data to the k-th centers as belonging to the k-th cluster. When the dataset is large, k-means algorithm is computationally demanding, that is why researchers have proposed algorithms such as mini batch k-means algorithm [11];

- *Distribution-based models*: these models are closely related to statistics and they are used to filter objects most likely belonging to the same distribution. For the generation of clusters, distribution models (Gaussian/Normal) are used. Clusters are defined on how likely the objects included are likely to belong to the same distribution;

- *Density-based models*: In this group of models, clusters are defined based on identifying areas of higher density that can be found in the remainder of the data space. The resulting clusters can have an arbitrary shape, and the points within can be arbitrarily distributed. Density clustering is able to handle noise when noisy objects are sparse in the dataspace. Some algorithms are DBSAN, OPTICS, DENCLUE, CLIQUE;

Clustering of "similar" objects is an inherently ambiguous task unless a measure of similarity is well defined. This also makes difficult to validate clustering algorithms, since the quality of the clustering highly depends on the similarity measure used and by its implementation. Moreover, if the number of clusters is not given in the clustering algorithm (like the case of k-means algorithm), then it is not trivial to find the optimal number of clusters and some criteria have to be involved.

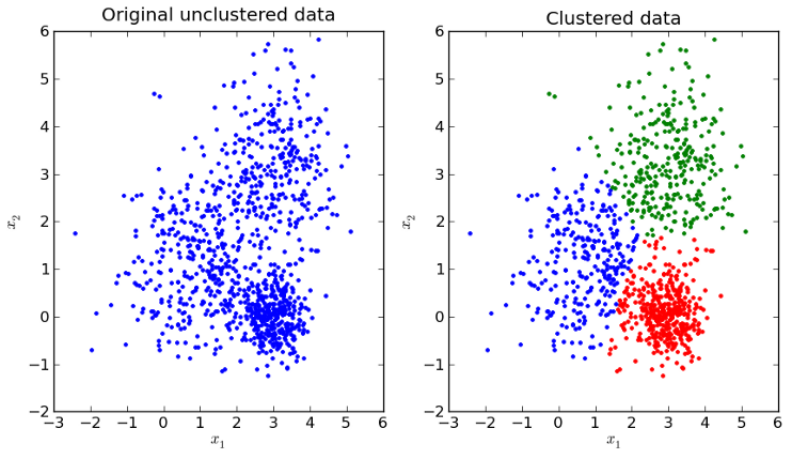


Figure 2.2: Elementary example of the clustering of a two-dimensional dataset(on the left and the same dataset divided in clusters (on the right).

Trajectory-based clustering

Some algorithms, inspired by clustering theory, consider not only the pick-up and drop-off locations but want to consider the spatiotemporal occupation of trajectories to try to determine if two trips are similar to each other. Trajectories are sequences of points

distributed in time and space [47].
These trajectories are clustered with different density-based clustering algorithms and criteria that could be:

- *DBSCAN-based algorithms (Density-Based Spatial Clustering of Applications with Noise)* that are unsupervised learning algorithms which group together points that are closely packed together. It means that it separates high density clusters form clusters of low density. Usually DBSCAN fails when dividing clusters of similar densities [34]. Such algorithms are for example TRACLUS [20], NETSCAN [52];
- *NEAT algorithm*, which works well for a road network model, while DBSCAN algorithms were considering only Euclidean distances between points for objects that can freely move in a two-dimensional space;
- *TOPOSCAN* [34], which still makes use of DBSCAN for the path clustering phase, the one that reveal potential carpool commuting routes. This algorithm consider the possibility of network-constrained movements for the vehicles.

A global decision maker, that has real-time information about vehicles and requests is needed. It then will be responsible for managing the acquired data and assign vehicles to tasks or sets of tasks. When a set of tasks is served by only one vehicle, it is needed that pick-up and drop-off of each of the tasks of the group satisfy the timing constraints of the single tasks, therefore minimizing the delay of the arrival time of the tasks.
The service has to provide transportation options in real-time, so that the service can be deployed for applications on site to offer the delivery service.

2.2.4 Trips generation from clustered rides

Taking inspiration from connectivity-based clustering techniques, we want an algorithm that is able to couple different routes coming from different customers. The algorithm should be able to recognize when two requests are similar in space and time in the two-dimensional space over which the pick-up and drop-off locations are spread.
To this aim, we need to define the similarity criteria that our algorithm will use to determine couplings between different rides.
We know that ride-sharing occurs frequently when the origins and/or destinations of two groups of passengers are close in location and time, that is:

If a vehicle can travel through all the pick-up and drop-off locations of two tasks (while respecting the pick-up/drop-off order of the single tasks), then the two tasks can be coupled and clustered together in one single route.

For the vehicle to be able to travel through all the pick-up and drop-off locations of the tasks, it is needed that there exists an order of pick-up and drop-off locations which allows the vehicle to move from one destination to the other within the time windows of the two tasks.
After all the possible matching between the different trips have been found and we have a set of possible trips (the different clusters) that are satisfying the customers requesting for a vehicle, we want to assign the available vehicles to the trips, so that to start the ride.

With the methodology we used, we will not find only a single configuration for the cluster

separation, but we will define a huge number of combinations between the tasks. Therefore, every task eventually belongs to more clusters. Then, we need to adopt a decision process able to assign the different rides (or clusters) to the vehicles.
A special attention has to be given to the fact that two different clusters containing the same task, cannot be assigned at the same time.
From now on, the clusters will be called trips and identified by the T.

2.2.5 Optimal assignment of the trips to vehicles

The assignment problem is addressed by solving an optimization problem for the assignment of vehicles to routes. The constraints of the optimization problem will define the feasible set on which to search for optimal solutions. The feasibility set defines that if a trip T_i containing task r_k is assigned, then another trip T_j, containing the same task r_ik cannot be assigned. Otherwise, task r_k would be assigned to two or more different vehicles.
The choice whether to assign between trip T_i, trip T_j and all the other trips, will depend on the optimization of a certain cost function.
The maximum number of trips that can be assigned when the optimization is run, is at most equal to the number of vehicles that have notified to be available for assignment. They could be idle vehicles (vehicles with no passengers on board) or vehicles that already have assigned one ride.

2.2.6 Rebalancing of the vehicle fleet

After the optimal assignment problem is run, it could happen that some rides have not been assigned or that some vehicles remained not assigned. This happens because of imbalances in the fleet of vehicles with respect to the incoming requests or because some tasks where far away from areas of high demand density and for them to be assigned resulted impractical in the optimal assignment.
If new tasks continue to arrive in areas of interest (that are areas with lot of pick-up demand), it is most likely that those not assigned task will not be assigned also the next time the assignment algorithm is called. This because it will always result more convenient (lower total cost) to service tasks in dense demand areas.
Therefore, these tasks run into the risk of remaining in a pending status for a long time unless we take care of them separately. The longer the ride will remain pending, the less will be the satisfaction of the customer with the service provided.
This is the reason why an add an additional feature is added to the algorithm in order to manage those unassigned requests, when possible. Indeed, not always it will be possible to match vehicles with unassigned requests. For example, if no vehicle is idle, it would be impossible to assign those requests, unless we accept to relocate an already busy vehicle to that task. For reasons regarding the satisfaction of the customers, though, we do not accept already assigned vehicles to be relocated when they are already heading to a pick-up point. We do not want buzzy assignments that jumps from one customer to the other. It would mean to have the customer seeing his ride always relocated to other vehicles. Moreover, given the frequency of the assignment algorithm call, it would not be feasible in certain situations for vehicles to continuously change the heading location and this may also lead to inefficiency and the system could be severely affected.
Therefore, if idle vehicles are available, we will assign them to unassigned requests, so that to head the vehicles towards areas where requests were not considered.

2.2.7 Additional observations on the choosen method

Many other approaches found in literature are based on heuristic solutions to the task assignment problem [64] [29], and many others solve it optimally. Solving the assignment in an optimal way, when the size of the problem is big, for example with dozens of tasks and dozens of vehicles, could be computationally impractical.
Our goal is to study an optimal assignment that is solved through the solution of a Mixed-Integer Linear Programming (MILP) after the matches between tasks and between tasks and vehicles have defined the feasible set of the problem.
The matching phase is taking the most part of the effort of the algorithm, since it has to analyze the possibility of sharing every ride with every other ride. Therefore, the space of possible combinations is large and to check for every possible combination is computationally inefficient.
This is the reason why a random factor is added in the generation of the shared routes, so that to limit the time spent in finding for combinations.
In fact, the possible combinations are so many that this limiting factor is highly influencing the computational performance of the algorithm.
Therefore, we want to study the influence of this added randomness to the quality of the assignment and evaluate in what measure the random factor can operate depending on the size of the problem, keeping in mind that our first goal is to satisfy as many requests as possible while minimizing the cost of doing so.
The algorithm has been tested with large size instances with hundreds of vehicles and hundreds of requests received simultaneously.
The algorithm can be called once every fixed time interval or at a task arrival, depending on the environment in which it will be deployed.
When new tasks arrive, and vehicles become available, the algorithm will be able to recompute a new assignment, fulfilling the requirement that already assigned trips will remain assigned. For instance, when a vehicle is assigned to pick-up one task, if a new task arrives, the algorithm will take into consideration of assigning also that task to the same vehicle, if that vehicle is still capable of satisfying the first task within its time window.
Moreover, priority can be assigned to task, for instance, if the customer is willing to pay an extra price for a "as soon as possible" delivery option. In that case, if some vehicle is free, we can guarantee that those priority tasks will be assigned prior to the other tasks. Otherwise, the chances of a priority tasks to be assigned will be increased.

Chapter 3

Method of solution

3.1 Algorithm workflow

As we briefly introduced before, the approach decouples the problem in three main steps:

- *Feasible trips generation*: At this stage, we are only interested in defining a feasible set of possible routes that the algorithm can assign. A trip exists and is said feasible only when the tasks composing it can share the same route and when at least one of the idle vehicles can satisfy the time windows of all the tasks composing the trip.
- *Optimal assignment*: At this step, we find an optimal assignment within all the possible feasible trips (that constitute the feasible set). This means that only part of all the possible combinations we found will be executed, seeking the best possible solution with respect to the cost function. The optimization problem is solved as a Mixed-Integer Linear Programming. Since one ride might belong to more than one trip and one trip might be feasible for more than one vehicle, a set of constraints will make sure that tasks are assigned to only one trip and that vehicles are assigned to only one trip.
- *Rebalancing*: At this stage, we run a Linear Program (LP) to match unassigned tasks to idle vehicles, if any is present. With the rebalancing phase, we are trying to compensate the imbalance that may arise when there are areas with high density of requests while other requests are therefore considered too expensive to be satisfied, because of their distance in the two-dimensional space.
  Using the rebalancing strategy, we are forcing those tasks to be assigned, to avoid that they remain pending for a long time.

These three phases together constitute the core of the algorithm for the assignment of the vehicles to the different rides. Each of the three phases contains additional details and features that will be treated deeply in the next sections.

3.2 The core idea for the generation of shared rides

The generation of feasible trips is based on the idea of looking at the links existing between rides (if they can be shared) and between the rides and the vehicles.
The high-level idea is to cast the problem of identifying the best trip sharing as a network problem.
A common use in the literature to describe the space of rides and vehicles and to define

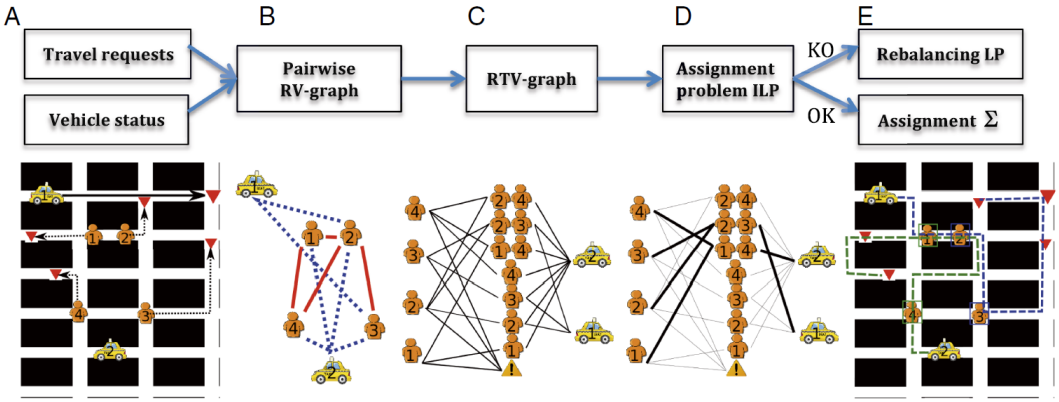


Figure 3.1: Representation of the generic flow of the algorithm from the inputs to the outputs.

the relations in between, is to use graph representation. In this way we can manipulate the information contained in the matrix representing the links between objects (rides and vehicles) in our space.
After comparing and analyzing possible matches between rides and robots, we obtain a network called shareability network [82]. Every task is defined by a starting time and an ending time, and a maximum allowed delay Δ that we assume equal for all the tasks. If we wanted to consider a different Δ for each task we could have simply add this information to the task data structure and use it every time the comparison between the task is done. We think that a unique Δ is more representative of what the service can offer instead allowing users to customize their maximum allowed waiting time would lead to make shareability very difficult since every customer would like to have to wait the less amount of time possible. This come in contrast with the basic idea of ride-shareability where customers accept the service to not be exclusive by paying less.
The parameter Δ represents the service quality, which defines the maximum time the assigned customers will have to wait for their vehicle to pick them up and what will be the maximum delay at the arrival.

The shareability network we want to build is an undirected graph G(N, E) that is build on the idea of using the maximum delay Δ to define if two tasks are likely to be shared in a single trip.
If the nodes of the network represent the tasks and the robots of our problem, then the links will be representative of the possibility to share the two tasks or, in the case of a link between a task and a vehicle the link will represent the possibility for that vehicle to serve the task.
Thus, we can say:

- An edge exists between two tasks if they can by combined under some shareability criteria, given by the time windows of the tasks.
- An edge exists between a request and a vehicle if the vehicle could travel to that request and satisfy its pick-up and drop-off within the time window of the request.

The objective now is to compute the graph adjacency matrix for the graph containing the requests and vehicles nodes, that from now on we will call *RV-Graph*.
Tasks can also be considered as trips of size one.
From now on, the *tasks* will be also called *requests* and we will refer to single tasks as r

or to subsets of tasks as $\mathcal{R}$. The word *request* is referred to the action of the customer requesting the pick-up and delivery task. *Requests* and *Tasks* will be interchangibly used.
Whether two tasks r_i and r_j can be combined, depends on the spatial/temporal properties of the two trips and on the upper bound Δ which is the maximum delivery delay.
The temporal dependence is related to the parameter Δ while the spatial dependence is due to the location of the pick-ups and drop-offs of the two tasks.
Another constraint regards the maximum capacity of the vehicles participating the ride-sharing service. Thus, if the sum of the sizes (number of passengers of the single ride) of the two tasks is greater than the maximum size of the available vehicles, then the two tasks cannot be combined.
The adjacency matrix will describe the links set that defines the relationship between all the tasks and vehicles.

This new approach allows polynomial-time, i.e. feasible, computation of the optimal ride-sharing strategy when at most two trips can be combined, and polynomial-time computation of a constant-factor approximation of the optimal solution when $k > 2$ trips can be shared [82]. The degree of the involved polynomials increases with k.
The shareability matrix is somewhat sparse which means that the average node degree is not high and not depend on n.

3.3 RV-Graph (shareability graph)

To build the shareability graph (*RV-Graph*) we need to understand if two tasks are compatible when we consider the delay parameter Δ and the location of pick-ups and drop-offs.
Nodes of the *RV-Graph* are requests and vehicles, *links* can exist between two different requests (if shareable) or between a request and a vehicle (if the vehicle can serve the request).
We want to look at the two pick-up and delivery tasks as tasks made of two separate actions. One is to pick-up the item/customer the other one is to drop it off.
We assume the actions of picking-up and dropping-off to be instantaneous. In real applications, these actions not only require the physical time to enter/exit the vehicle or to take/release the object, but we would encounter many other problems, like delays of the customer, difficulty in recognizing the passenger, time spent for approaching the vehicle when there are no parking spots right in front of the customer, time to grab the object, obstacles that hinder the passage and so on.
We can think our problem to be on a very high level of hierarchy, we think that those delays and inefficiencies could be modeled by a *waiting time* slot that if not satisfied, simply gives the permission to the vehicle/robot to abort the request and ignore the passenger/task when this is no more feasible inside the allowed time window. This has not been implemented.

3.3.1 Request - request couplings

To satisfy all the pick-ups and drop-offs we want to think of an imaginary vehicle that starting from the first location, will visit all the other three locations travelling from one to another until it reaches the last location.
Building the shareability graph for shared rides of size two starting from the task set $\mathcal{R} = \{r_1, \ldots, r_n\}$ requires $O(n^2)$ where n is the number of rides (tasks).
This is true in the worst case, that is when all the possible combinations between pick-up

and delivery events of the two tasks are verified. But in the real case, the precedence constraint between the pick-up and the drop-off events is constraining that only four of the combinations are possible between the two tasks.
Therefore, the possible combinations are:

1. $o_i \rightarrow o_j \rightarrow d_i \rightarrow d_j$;
2. $o_i \rightarrow o_j \rightarrow d_j \rightarrow d_i$;
3. $o_j \rightarrow o_i \rightarrow d_j \rightarrow d_i$;
4. $o_j \rightarrow o_i \rightarrow d_i \rightarrow d_j$;

where o_x and d_x are the origin and destination of task x.

We already have a formal order of the possible pick-ups and drop-offs of two tasks when they belong to a shared ride. Now we give a definition of what we mean by shared ride, that is a ride composed of two different tasks. A coordinated trip $T_{i,j}$ (or *shared ride*) is a trip composed of two requests r_i and r_j.

Definition [82]: The coordinated trip $T_{i,j}$ is feasible if and only if a trip route can be found such that the following conditions are satisfied:

1. $st_i \leq pt_i \leq st_i + \Delta$;
2. $st_j \leq pt_j \leq st_j + \Delta$;
3. $dt_i \leq at_i + \Delta$;
4. $dt_j \leq at_j + \Delta$;

with pt_x the pick-up time at o_x in the combined trip and dt_x the delivery time at d_x in the combined trip.

Conditions 1 and 2 are not allowing the pick-up to happen earlier than expected, but it is required that the rider is there from the time the pickup should start until the time the pickup has to be completed. Instead, conditions 3 and 4 allow the ride to arrive earlier than expected, considering it as a point in favor for the service. These conditions both consider that a customer is willing to wait at most some extra time Δ.

From [82]: **Theorem.** *Building the shareability network S = (R, L) starting from the trip set $\mathcal{R} = \{r_1, ..., r_n\}$ requires $O(n^2)$ time.*

Proof. In the worst-case, we have to consider all $O(n^2)$ possible pairs of requests r_i, r_j. For each pair, the feasibility condition for the combined trip $T_{i,j}$ can be verified in $O(1)$ as follows.
The following is the procedure used to verify if two trips can be matched or if they cannot. It makes use of combinations 1-4 and of conditons *a-d* to build constraints that must be verified so that the two requests r_i and r_j are shareable.
We take as an example combination 1 and first define the time distances between points so that to define, with respect to the first pick-up, all the pick-ups and drop-offs in time domain.

We call $tt(x, y)$ the function that computes the travel time between two any points on our map, and we will use it to compute the time between the different points of the route o_i, o_j, d_i, d_j. This function makes simple use of a constant to relate the so called Manhattan distance (L_1 distance) between two points in the 2D map and the travel time. We are assuming the most possible ideal conditions for the computation of the travel time. In a real evnironment, when a complete application has been developed and therefore a network representation of the map is available, standard techniques for efficiently computing shortest paths can be used. The best methods can compute shortest paths on networks with 70 millions edges in less than a millisecond.
At the end, we will have:

$$\begin{aligned} pt_j &= pt_i + tt\,(o_i, o_j) \\ dt_i &= pt_j + tt\,(o_j, d_i) \\ dt_j &= dt_i + tt\,(d_i, d_j) \end{aligned} \tag{3.1}$$

Where pt_i is trivially verified to be inside or outside the boundary imposed in condition a.
We can now combine equations 3.1 with conditions $a - d$ and obtain the new conditions that will be used to find out if request r_i and request r_j can be combined in one trip.
These conditions are, for pick-ups/drop-offs combination 1:

$$\begin{aligned} st_i \leqslant pt_i \leqslant st_i + \Delta \quad &\text{(A1)} \\ st_j \leqslant pt_i + tt\,(o_i, o_j) \leqslant st_j + \Delta \quad &\text{(A2)} \\ pt_i + tt\,(o_i, o_j) + tt\,(o_j, d_i) \leqslant at_i + \Delta \quad &\text{(A3)} \\ pt_i + tt\,(o_i, o_j) + tt\,(o_j, d_i) + tt\,(d_i, d_j) \leqslant at_j + \Delta \quad &\text{(A4)} \end{aligned} \tag{3.2}$$

Conditions $A1 - A4$ refers to the case where tasks r_i and r_j are related by $o_i \to o_j \to d_i \to d_j$. For this route, the feasibility can be verified by checking whether a value of the variable pt_i that simultaneously satisfies the four conditions $A1 - A4$ exists.
For the other combinations of pick-ups and drop-offs, the feasibility condition is verified in a similar way. Following, we will show what are the conditions to be satisfied for tasks r_i and r_j in the other cases (combinations $2 - 4$):

- In the case the combination is $o_i \to o_j \to d_j \to d_i$, we have:

$$\begin{aligned} st_i \leqslant pt_i \leqslant st_i + \Delta \quad &\text{(B1)} \\ st_j \leqslant pt_i + tt\,(o_i, o_j) \leqslant st_j + \Delta \quad &\text{(B2)} \\ pt_i + tt\,(o_i, o_j) + tt\,(o_j, d_j) \leqslant at_j + \Delta \quad &\text{(B3)} \\ pt_i + tt\,(o_i, o_j) + tt\,(o_j, d_j) + tt\,(d_j, d_i) \leqslant at_i + \Delta \quad &\text{(B4)} \end{aligned} \tag{3.3}$$

- In the case the combination is $o_j \rightarrow o_i \rightarrow d_i \rightarrow d_j$, we have:

$$\begin{aligned} st_j \leqslant pt_j \leqslant st_j + \Delta \quad & \text{(C1)} \\ st_i \leqslant pt_j + tt\,(o_j, o_i) \leqslant st_i + \Delta \quad & \text{(C2)} \\ pt_j + tt\,(o_j, o_i) + tt\,(o_i, d_i) \leqslant at_i + \Delta \quad & \text{(C3)} \\ pt_j + tt\,(o_j, o_i) + tt\,(o_i, d_i) + tt\,(d_i, d_j) \leqslant at_j + \Delta \quad & \text{(C4)} \end{aligned} \tag{3.4}$$

- In the case the combination is $o_j \rightarrow o_i \rightarrow d_j \rightarrow d_i$, we have:

$$\begin{aligned} st_j \leqslant pt_j \leqslant st_j + \Delta \quad & \text{(D1)} \\ st_i \leqslant pt_j + tt\,(o_j, o_i) \leqslant st_i + \Delta \quad & \text{(D2)} \\ pt_j + tt\,(o_j, o_i) + tt\,(o_i, d_j) \leqslant at_j + \Delta \quad & \text{(D3)} \\ pt_j + tt\,(o_j, o_i) + tt\,(o_i, d_j) + tt\,(d_i, d_j) \leqslant at_i + \Delta \quad & \text{(D4)} \end{aligned} \tag{3.5}$$

If one of the systems of equations described by the sets of equations A, B, C, D, is defining a non empty set on the variable pt_i (when task i is the first to be picked up) or on the variable pt_j (when task j is the first to be picked up), then a solution exists and the two tasks can be combined.
Depending on what is the set of equation for which a non-empty set of solutions exists, then we will have the corresponding feasible matching order of pick-up and drop-off actions between the two tasks.
When two tasks r_i and r_j can be shared, then the *link* $e\,(r_i, r_j)$ between the two nodes representing the two tasks in the *RV-Graph* will be generated.
Notice that two tasks can be shared in more than one order. This means, for instance, that both orders $o_i \rightarrow o_j \rightarrow d_j \rightarrow d_i$ and $o_j \rightarrow o_i \rightarrow d_i \rightarrow d_j$ can result feasible for two tasks i and j. In this case we have many choices on how to procede. We will treat this topic in more details later.

3.3.2 Vehicle - request couplings

The next stage is to verify if any of the available vehicles (that are nodes in the *RV-Graph*) can serve any of the pending requests. If this is true, a link will exist between the vehicle node and the request node.
To find out if the vehicle is able to serve the request, we will use function $travel(v, r)$ where v and r are a specific vehicle and a specific request for which we are computing the matching.
Vehicle v is characterized by location coordinates v_x and v_y, by a status that tells if it has any request already assigned (and if yes, what requests), and by a capacity c_v, that is representative of the number of passengers the vehicle can accomodate at the same time. In the case of robots carrying objects, the capacity will represent the number of items that the robot can collect at the same time.

Function $travel(v, r)$ will make sure that the vehicle respects all the requirements of task r.
The conditions to be verified so that the vehicle is able to serve the task are the following:

1. $T_{clock} + tt\,(v_{xy}, o_r) \leqslant st_r + \Delta$;
2. $T_{clock} + tt\,(v_{xy}, o_r) + tt\,(o_r, d_r) \leqslant at_r + \Delta$;
3. $r_{pass} \leqslant c_v$;

Where T_{clock} is the time at which function $travel()$ is called, v_{xy} is the location of vehicle v, o_r and d_r are the origin and destination location coordinates of the request r, st_r and at_r are the starting and arrival time of the request and Δ is the maximum allowed delay.

When all the conditions 1, 2 and 3 are satisfied, then the link $e\,(v_v, r_r)$ is generated between vehicle v and task r, otherwise the function $travel(v, r)$ will return *false*.
Condition 2 is automatically verified when condition 1 is true. This becasue the arrival time at of a task is computed on the base of the minimum time required to reach the destination from the starting point of the task, knowing its starting time st.
If the conditions are satisfied it means that the vehicle can successfully serve the task.
Function $travel(v, r)$ will be called for each $v \in \mathcal{V}$, and for each $r \in \mathcal{R}$, where $\mathcal{V}$ is the set of available vehicles and $\mathcal{R}$ is the set of pending requests at the time of the assignment algorithm call.
When the number of tasks and vehicles is high, and especially if the density of tasks and vehicles in an area is high, then the possibility of matching requests and vehicles will be high. This means that many links will be formed between tasks and between tasks and vehicles. A maximum number of edges (links) per node can be fixed, so that to decrease the average computation time for the matching phase.

Example. Representation of the adjacency matrix for the *RV-Graph*:

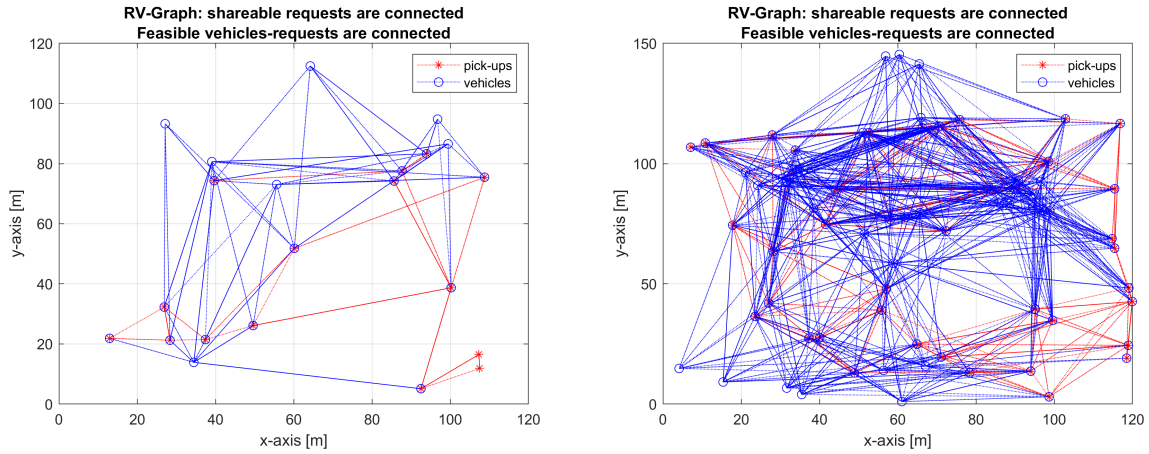
$$\begin{array}{c} \\ r_1 \\ r_2 \\ \vdots \\ r_n \\ v_1 \\ v_2 \\ \vdots \\ v_m \end{array} \begin{array}{c} \begin{array}{cccccccc} r_1 & r_2 & \cdots & r_n & v_1 & v_2 & \cdots & v_m \end{array} \\ \begin{pmatrix} 0 & 1 & \ldots & 0 & 0 & 1 & \cdots & 0 \\ 1 & 0 & \ldots & 1 & 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \ldots & 0 & 1 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & \cdots & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \ldots & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{3.6}$$

The matrix represented in Equation 3.6 is the adjacency matrix for the *RV-Graph*.
The *RV-Graph* is a simple graph with vertex set $\mathcal{N} = \{N_1, N_2, ..., N_{n+m}\}$ and its adjacency matrix Adj is a square $|\mathcal{N}| \times |\mathcal{N}|$ matrix.
The rows and columns of the matrix represent the vertexes of the *RV-Graph* network, that are nodes $\mathcal{N} = \mathcal{R} \cup \mathcal{V}$ with $\mathcal{R} = \{r_1, r_2, ...r_n\}$ that is the set of tasks and $\mathcal{V} = \{v_1, v_2, ..., v_m\}$ that is the set of vehicles.

$$Adj_{ij} = \begin{cases} 1 & \text{if } e(N_i, N_j) \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

The entries of the adjacency matrix are *one* when there is an edge from vertex N_i to vertex N_j, and *zero* when there is no edge [15]. The network is an undirected graph therefore its adjacency matrix is symmetric.
The diagonal elements of the matrix are all zero, since edges from a vertex to itself (*loops*) are not allowed in simple graphs.

Once the sets of requests and vehicles have been analyzed to find the possible matchings, we want to build a set of trips $\mathcal{T}$ that represent tasks that can be matched with vehicles and the matchings between tasks that can be served by vehicles.
This set of trips will be represented by another network, that is called *RTV-Graph* [5]. The next chapter will treat in detail the *RTV-Graph*.



(a) (b)

Figure 3.2: *a*) RV-Graph for a small-size instance with 15 requests and 7 vehicles. *b*) RV-Graph for a medium-size instance with 40 requests and 30 vehicles.

3.4 RTV-Graph

The *RTV-Graph* is a new network generated in order to identify all the possible *feasible* trips. By *feasible* we mean that a vehicle is actually able to reach the first task, pick it up and heading to its destination within the task time window (defined by delay Δ). Then, if any other task belongs to the *feasible* trip, the vehicle will also be able to pick-up and drop-off all the other tasks within the respective time windows.
We call 'trips of size k' the trips that are composed of k (two or more) requests that can be matched. In particular, we will build trips of size two and trips of size three.
First, we want to introduce how trips of size one are built. Then, we will explain how trips of size two are built and finally how trips of size three are generated. For both sections we will follow the procedures explained in [5], making some changes where necessary because of choices due to computation time required for finding a solution to the *RTV-Graph* generation.

3.4.1 Trips of size one

Trips of size *one* are the simplest in our set of trips, because they are simply composed by one single task and linked to a vehicle. They are the first of the trips that we will find vehicle after vehicle, in such a way to ease later the generation of trips of greater sizes. Indeed, trips are added to the *RTV-Graph* by increasing size, for each vehicle.
All the information we need to generate the trips of size *one* can be found into the *RV-Graph* of our set of tasks and vehicles. In fact, when we used function *travel()* to see if the vehicles were able to satisfy single requests, we were actually verifying whether that candidate trip of size *one* was feasible or not. Therefore, what we need to do is to look back at the *RV-Graph adjacency* matrix and gather the information we need. Indeed, we know that every task r that is linked to vehicle v in the *adjacency* matrix could be satisfied by that vehicle. This means that the task is part of a trip of size *one* T_1. Therefore, everytime a link between a vehicle v_j and a task r_k exists in the *RV-Graph*, we will add a trip T_i to the *RTV-Graph* as a new node and add the edge $e(r_k, T_i)$ between the task and the trip and edge $e(v_j, T_i)$ between the vehicle and the trip. We can have a look at the simple Algorithm 1 that add the trips to a list of trips of size *one* for each vehicle $v \in \mathcal{V}$. We will discuss later about the sampling phase at line 3 and 4 of the pseudocode.

Algorithm 1 Size one trips generation

```
1:  for each vehicle v ∈ V do
2:      Generate trips of size one:
3:      if RV-edges[v].size > n_1 then
4:          RV-edges[v] ← sample(RV-edges[v], n_1)
5:      for each e(r_i, v) ∈ RV-edges[v] do
6:          if ( then T_1(= {r_1}) ∈ T_1)
7:              Generate e(T_1, v) in RTV-Graph
8:          else
9:              T_1 ← T_1 = r_1
10:             Generate e(r_i, T_1), r_i ∈ T_1, and e(T_1, v) in RTV-Graph
```

3.4.2 Trips of size two

Even if the trips of size *two* are generated on top of the trips of size *one*, we still need to go back to the information stored in the *RV-Graph* to be able to build trips of size *two*. Indeed, we will analyze the regions of the *RV-Graph* to find what of its induced subgraphs are complete. The complete subgraph of a network are also called cliques. Cliques are subset of vertices of a graph (induced graph) such that any two vertices of the subset or adjacent (connected). As explained in Figure 3.4 a clique is a fully connected induced subgraph. Therefore, what we are looking for is a clique of the *RV-Graph* composed by one vehicle vertex and at most two request vertexes.
If we can find such a clique, it means that the request vertexes composing that clique will consitute a *candidate* feasible trip. This because the two request nodes are adjacent (linked to each other) and a vehicle vertex is connected to both of them, which means that vehicle can serve the requests if served singularly.

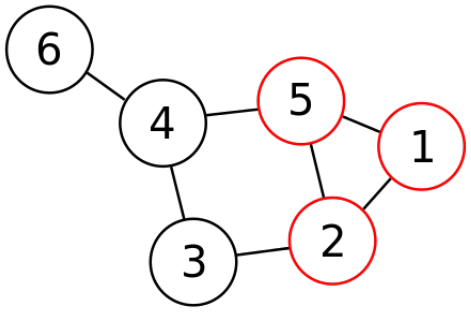


Figure 3.3: Example of a graph made of six vertices with one maximum clique. The maximum clique is the subset of vertices 1, 2, 5. They are all connected with each other, so it is a complete induced graph of the original network. Other maximal cliques are the pairs {2, 3}, {3, 4}, {4, 5} and {4, 6}. Maximal cliques are the ones to which no more vertexes can be added. So for example cliques {1, 2}, {1, 5} and {2, 5} are not maximal beacuse other vertices can be added to form the bigger clique {1, 2, 5}.

We want to state a necessary condtion for the existence of a trip:

A trip $T_{i,j}$, composed by tasks r_i and r_j, is feasible only if both tasks r_i and r_j can be served by the same vehicle v_v when considered singularly.

This can be formalized with the *lemma* proposed in [5]:

Lemma (Cliques). *A trip T can be feasible only if a clique in the RV-Graph exists for all the requests in T and some vehicle v. Namely, if T is valid, then,*

$$\exists\, v \in \mathcal{V} \text{ such that } \forall r_1, r_2 \in T,\ e(r_1, r_2) \text{ and } e(r_1, v) \text{ exist} \tag{3.7}$$

The reason why both the tasks has to be connected is because if one of the two tasks was not linked (adjacent in the graph) to the considered vehicle, that task could not be served by that vehicle. Therefore, it is trivial to state that this task could not be served in a shared route if it cannot even be served by itself.
This is why we look for cliques of the *RV-Graph* to find candidate feasible trips.

After having found a candidate feasible trip we need to call a function that can verify that the two tasks can actually be served by that vehicle (for cliques with one vehicle and one request, we already know that, if a link exist, they can be considered feasible trips).
We need to define a function $travel\_size\_two(v, r_i, r_j)$ that computes the feasibility of the candidate trip $T_{i,j}$ based on some conditions, so that to satisfy constraints like the vehicle capacity c_v, the starting time st and arrival time at of the two tasks, knowing that a maximum delay Δ is allowed.

One important thing is that function $travel\_size\_two()$ will also get as argument a number that tells what is the order linking the two tasks, considering that the first of the two tasks is always the first task passed as argument.
This means that we will pass as argument a state 1 if the order linking the two tasks is

$o_i \to o_j \to d_i \to d_j$, state 2 if $o_i \to o_j \to d_j \to d_i$ is the possible combination and state 3 if both the combinations are possible.
The infomration about the most convenient (or the only possible) order with which the tasks are combined is contained in the matrix *kind_of_link*, that is a square matrix of dimension $n \times n$ where n is the total number of tasks.

Example. An example of *kind_of_link* matrix is the following:

$$\begin{array}{c} \\ r_1 \\ r_2 \\ \vdots \\ r_n \end{array} \begin{array}{c} \begin{array}{cccc} r_1 & r_2 & \cdots & r_n \end{array} \\ \begin{pmatrix} 0 & 1 & \dots & 0 \\ 1 & 0 & \dots & 2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 2 & \dots & 0 \end{pmatrix} \end{array} \tag{3.8}$$

In this example, since the entry of the *kind_of_link* matrix for tasks r_1 and r_2 is 1, it means that the two tasks are linked by the first of the possible orders, that is $o_i \to o_j \to d_i \to d_j$. Instead, tasks r_2 and r_n can be shared following the pick-ups and drop-offs order 2, that is $o_i \to o_j \to d_j \to d_i$.

In the case in which both the combinations are possible, and both the combinations result feasible, we will take as feasible trip the one with the lower cost, but we will save the information that the trip could be done in both ways.
The reason why we choose only one of the two trips is that there is not reason in considering as trip the one with the higher cost, if then the choice on the assignment of the trips is based on minimizing the total cost of the assignment.
It is important, though, to save the information about the possibility of doing the trip with another combination of pick-ups and drop-offs, because this information will be used when looking for trips of size 3.
The following are the conditions that are verified by $travel\_size\_two(v, r_i, r_j)$ in order to say if the size two trip is feasible:

- In the case the two tasks can be shared by $o_i \to o_j \to d_i \to d_j$:
  1. $T_{clock} + tt\,(v_{xy}, o_{r_1}) + tt\,(o_{r_1}, o_{r_2}) \leqslant st_{r_2} + \Delta$;
  2. $T_{clock} + tt\,(v_{xy}, o_{r_1}) + tt\,(o_{r_1}, o_{r_2}) + tt\,(o_{r_2}, d_{r_1}) \leqslant at_{r_1} + \Delta$;
  3. $T_{clock} + tt\,(v_{xy}, o_{r_1}) + tt\,(o_{r_1}, o_{r_2}) + tt\,(o_{r_2}, d_{r_1}) + tt\,(d_{r_1}, d_{r_2}) \leqslant at_{r_2} + \Delta$;
  4. $r_{1.pass} + r_{2.pass} \leqslant c_v$;

- In the case the two tasks can be shared by $o_i \to o_j \to d_j \to d_i$:
  1. $T_{clock} + tt\,(v_{xy}, o_{r_1}) + tt\,(o_{r_1}, o_{r_2}) \leqslant st_{r_2} + \Delta$;
  2. $T_{clock} + tt\,(v_{xy}, o_{r_1}) + tt\,(o_{r_1}, o_{r_2}) + tt\,(o_{r_2}, d_{r_2}) \leqslant at_{r_2} + \Delta$;
  3. $T_{clock} + tt\,(v_{xy}, o_{r_1}) + tt\,(o_{r_1}, o_{r_2}) + tt\,(o_{r_2}, d_{r_2}) + tt\,(d_{r_2}, d_{r_1}) \leqslant at_{r_1} + \Delta$;

4. $r_{1.pass} + r_{2.pass} \leqslant c_v$;

If the conditions are not satisfied, the function will return *false* and the two tasks will be considered only as size one trips.

After the the analysis of the *RV-Graph* and generation of the *RTV-Graph* we will end up with a set of trips $\mathcal{T} = \{T_1, ..., T_w\}$ each of which will be of the type $T = \{r_1, ..., r_{n_T}\}$. A request r_i may be part of several feasible trips of varying sizes, and a trip might admit several different vehicles for execution. The request-trip-vehicle *RTV-Graph* contains edges $e(r, T)$, between a request r and a trip T and edges $e(T, v)$, between a trip T and a vehicle v.
We can formalize as [5]:

$$\begin{aligned} \exists\, e(r,T) &\Leftrightarrow r \in T \\ \exists\, e(T,v) &\Leftrightarrow travel(v,T) = \text{"valid"} \end{aligned} \tag{3.9}$$

All the trips that are added in the *RTV-Graph* are linked to vehicles nodes and so they are feasible. All of these trips will be taken into account during the optimization process.

Algorithm 2 Size two trips generation

```
1: for each vehicle v ∈ V do
2:     Generate trips of size one:...
3:     Generate trips of size two:
4:     if T1_list[v] ≠ ∅ then
5:         for each r1 ∈ T1_list[v] do
6:             for each r2 ∈ T1_list[v] do
7:                 if r1 ≠ r2 then
8:                     if travel_size_two(v, {r1, r2}) = valid then
9:                         if T2(= {r1, r2}) ∈ T∈ then
10:                            Generate e(T2, v) in RTV-Graph
11:                        else
12:                            T∈ ← T2 = {r1, r2}
13:                            Generate e(ri, T2), ∀ri ∈ T2, and e(T2, v) in RTV-Graph
```

3.4.3 Trips of size three

We want now to treat the generation of trips of size three, which are those feasible trips composed by three tasks r_i, r_j and r_q and that have at least one vehicle that can pick-up and drop-off all of the tasks within the respective deadlines imposed by the time windows. Looking to all the possible combinations of trips of size one and trying to prove on these combinations the feasibility with functions similar to $travel()$ and $travel\_size\_two()$ would be computationally infeasible.
We need therefore a way to exclude combinations that we could tell, before calling a feasibility check function, if that specific combination is feasible or not.
To this aim, we recall a *lemma* stated in [5]:

Lemma (Sub-feasibility). *A trip T can be feasible only if there exists a vehicle v for which, for all $r \in T$, the sub-trips $T' = T \setminus r$ are feasible (a sub-trip T' contains all the*

requests of T but one). Namely,

$$T \text{ feasible } \Rightarrow \exists v \in \mathcal{V} \text{ such that } \forall r \in T,\ e(T \setminus r, v) \text{ exists.} \tag{3.10}$$

Therefore, a trip T only needs to be checked for existence if there exists a vehicle v for which all of its sub-trips T' present an edge $e(T', v)$ in the RTV-Graph

Since our objective is to obtain trips of size three, we need to combine trips of size one with trips of size two. There could be the possibility of combining three size one trips only, and then check for the different combinations. This, fortunately, is not needed, becasue for the *lemma* stated before, if a trip of size three is feasible, this means that all of the size two sub-trips $T \setminus r$ need to exist. If this is true, all of the three tasks, if taken two by two, will constitute size two trips.
Therefore, it is more clever, during the analysis, to consider size three trips as composed of one size one trip and one size two trip. This will help us in discarding the combination that we can a-priori consider unfeasible.

Example. Try to consider three tasks r_a, r_b and r_c so that $T_a = \{r_a\}$ and $T_{b,c} = \{r_b, r_c\}$ where T_a is a trip of size one and $T_{a,b}$ is a trip of size two. We have that $T_a \in \mathcal{T}_1^v$ where $\mathcal{T}_1^v$ is the set of all the trips of size one for a certain vehicle v and $T_{b,c} \in \mathcal{T}_2^v$ where $\mathcal{T}_2^v$ is the set of all the trips of size two for a certain vehicle v. Assume we want to find out if the set of the three tasks can be considered a candidate feasible trip of size three. As the *lemma* is suggesting, we start looking for the $T' = T_{a,b,c} \setminus r\ \forall r \in \{r_a, r_b, r_c\}$ where $T' \in \mathcal{T}_2^v$. For instance, we found that $T_{b,a} \in \mathcal{T}_2^v$ but not trip $T_{a,c}$ or $T_{c,a}$ exists within $\mathcal{T}_2^v$.
Therefore, the three tasks cannot be considered a candidate trip of size three.
If at least one between $T_{a,c}$ and $T_{c,a}$ existed in $\mathcal{T}_2^v$, then the three tasks would have been considered as a candidate trip of size three.

Once we have a size one trip combined with a size two trip, we need to take into consideration all the combinations between the trips. This means that having tasks r_i, r_j and r_q we can define *six* combinations for the pick-up order and *six* combinations for the drop-off order.
If we consider that the pick-ups must happen before the drop-offs of the respective tasks, and if we consider that we want all the pick-ups to happen before all the drop-offs we will obtain two sets of possible combinations: one for the pick-ups of the tasks, one for the drop-offs of the tasks.
We want therefore to define what are the possible combination of pick-ups and drop-offs for the three tasks.
The possible orders for the pick-ups of the three tasks are:

1. $p_{r_1} \rightarrow p_{r_2} \rightarrow p_{r_3}$
2. $p_{r_1} \rightarrow p_{r_3} \rightarrow p_{r_2}$
3. $p_{r_2} \rightarrow p_{r_1} \rightarrow p_{r_3}$
4. $p_{r_2} \rightarrow p_{r_3} \rightarrow p_{r_1}$
5. $p_{r_3} \rightarrow p_{r_1} \rightarrow p_{r_2}$

6. $p_{r_3} \rightarrow p_{r_2} \rightarrow p_{r_1}$

where p_{r_x} is the pick-up of task x.

For the drop-offs orders of the three tasks we have:

1. $d_{r_1} \rightarrow d_{r_2} \rightarrow d_{r_3}$
2. $d_{r_1} \rightarrow d_{r_3} \rightarrow d_{r_2}$
3. $d_{r_2} \rightarrow d_{r_1} \rightarrow d_{r_3}$
4. $d_{r_2} \rightarrow d_{r_3} \rightarrow d_{r_1}$
5. $d_{r_3} \rightarrow d_{r_1} \rightarrow d_{r_2}$
6. $d_{r_3} \rightarrow d_{r_2} \rightarrow d_{r_1}$

where p_{r_x} is the pick-up of task x.

Since vehicles that picks-up requests have also to drop them off, then we will have a total of 36 combinations with which the candidate trips of size three can be executed.
These 36 possible combinations derive from the need of defining trips of size three, which means that at a certain time during the trip, all the 3 requests will be present inside the same vehicle. To this aim, all the pick-ups must happen consecutively. If the drop-off of one of the rides on board the vehicle happens before the pick-up of the third task, then the trip will be considered as an original trip of size two that temporally becomes a trip of size one (after dropping off the passenger) and then picks-up the new passengers to become again a trip of size two.
In this case there is the need for the vehicle to notify about the change of its status from having *two* rides on board to having only *one*. This becasue our algorithm is doing istantaneous assignment (IA) and we are not planning future assignments ahead in time.

We now want to define a function *travel_size_three*() which is used to compute the feasibility of *candidate* feasible trips.
This function works in a similar way to *travel*() and *travel_size_two*() functions. It takes as arguments the three tasks that are candidates to be matched in a size three trip, the vehicle that is supposed to serve the candidate trips and the possible orders that are feasible (considering the a-priori knowledge of the shareability of requests).
Following, we show the conditions that function *travel_size_three*() have to verify to return the feasibility of the *candidate* trip:

1. $T_{clock} + tt\,(v_{xy}, o_{r_1}) + tt\,(o_{r_1}, o_{r_2}) + tt\,(o_{r_2}, o_{r_3}) \leqslant st_{r_3} + \Delta$;
2. $T_{clock} + tt\,(v_{xy}, o_{r_1}) + tt\,(o_{r_1}, o_{r_2}) + tt\,(o_{r_2}, o_{r_3}) + tt\,(o_{r_3}, d_{r_1}) \leqslant at_{r_1} + \Delta$;
3. $T_{clock} + tt\,(v_{xy}, o_{r_1}) + tt\,(o_{r_1}, o_{r_2}) + tt\,(o_{r_2}, o_{r_3}) + tt\,(o_{r_3}, d_{r_1}) + tt\,(d_{r_1}, d_{r_2}) \leqslant at_{r_2} + \Delta$;
4. $T_{clock} + tt\,(v_{xy}, o_{r_1}) + tt\,(o_{r_1}, o_{r_2}) + tt\,(o_{r_2}, o_{r_3}) + tt\,(o_{r_3}, d_{r_1}) + tt\,(d_{r_1}, d_{r_2}) + tt\,(d_{r_2}, d_{r_3}) \leqslant at_{r_3} + \Delta$;
5. $r_{1.pass} + r_{2.pass} + r_{3.pass} \leqslant c_v$;

Where the notations o_{r_1}, o_{r_2}, o_{r_3}, d_{r_1}, d_{r_2} and d_{r_3} are generic notations given to the parameters of the function, that indicates what is the order of the pick-ups and what is the order of the drop-offs. This means that o_{r_1} and d_{r_1} are not, in general, related to the same task. Instead, o_{r_1} tells what is the first task to be picked up (for the candidate trip) and d_{r_1} tells what is the first task to be dropped off (for the candidate trip). They will be passed as argument in the specific order of the candidate trips.
Since trying all 36 different combinations would be computationally infeasible, we need a way to discard a-priori some of the combinations. In this way we will try only a limited number of combinations.
The solution consists in focusing on the sub-trips of size two that compose our combination.
When starting the exploration for trips of size three, we start by having a set of three tasks r_1, r_2 and r_3, for which we are going to search for sub-trips of size two. We remember that if all the possible trips of size two between the three tasks exists for at least one vehicle, then the three tasks together with the vehicle will be candidates to become a trip of size three.
We now want to recall the *adjacency matrix* of the *RV-Graph*. At that point, when building the *RV-Graph* we were also paying attention to what was the order of the pick-ups and drop-offs for the two tasks, and saving this information in a matrix named $kind\_of\_link$ that has entries equal to zero when two tasks cannot be shared, and a number specifying the kind of combination when the two tasks can be shared.
Using these information, we can take two of the three tasks at a time, and at the same moment of verifying if they are shareable, verify what is the order of pick-ups and drop-offs that is feasible between the two tasks when they belong to a trip of size two.

Example. For instance, we can consider tasks r_1, r_2 and r_3 and assume that it has been verified that all the sub-trips $T' = T \setminus r$ exist for a certain vehicle v. Now, suppose the following are the feasible trips:

- $T_{1,2}$ such that $\{o_{r_1}, o_{r_2}, d_{r_2}, d_{r_1}\}$ is the order of the pick-ups and drop-offs. We will call this trip $T_{1\to 2\to 2'\to 1'}$
- $T_{1,3}$ such that $\{o_{r_1}, o_{r_3}, d_{r_1}, d_{r_3}\}$ is the order of the pick-ups and drop-offs. We will call this trip $T_{1\to 3\to 1'\to 3'}$
- $T_{3,1}$ such that $\{o_{r_3}, o_{r_1}, d_{r_1}, d_{r_3}\}$ is the order of the pick-ups and drop-offs. We will call this trip $T_{3\to 1\to 1'\to 3'}$
- $T_{3,2}$ such that $\{o_{r_3}, o_{r_2}, d_{r_2}, d_{r_3}\}$ is the order of the pick-ups and drop-offs. We will call this trip $T_{3\to 2\to 2'\to 3'}$

Given these feasible trips of size two for the three tasks, we can define what are the possible combinations for the pick-ups and the possible combinations for the drop-offs. To this aim, starting from the pick-ups, we can define a set of constraints that will tell what are the orders in which the tasks can be picked up. The conditions we have form the pick-ups order are:

- p_{r_1} can precede p_{r_2};
- p_{r_1} can procede p_{r_3};
- p_{r_3} can procede p_{r_1};
- p_{r_3} can procede p_{r_2};

That translate in the constraints:

$$\begin{aligned} tp_{r_1} &\leqslant tp_{r_2}; \\ tp_{r_3} &\leqslant tp_{r_2}; \end{aligned} \tag{3.11}$$

where tp_x is the pick-up time of task x.

This means that the only feasible combinations for the pick-ups for the given tasks, given the constraints 3.11 are:

- $1 \rightarrow 3 \rightarrow 2$;
- $3 \rightarrow 1 \rightarrow 2$;

If we procede in the same way for the drop-offs case, we obtain constraints $tp_{r_2} \leqslant tp_{r_1}$, $tp_{r_1} \leqslant tp_{r_3}$ and $tp_{r_2} \leqslant tp_{r_3}$ that lead to following possible combinations for the drop-off of the three tasks:

- $2' \rightarrow 1' \rightarrow 3'$;

As we can notice we went from the need to check for 36 combinations to being able to check only 2 possible combinations for the *candidate* trip of size three composed by the three tasks.
At this point, the function *travel_size_three*() will only try these two combinations. Since the number of vehicles and of tasks are huge, then also the number of trips of size one and trips of size two will be great. This shows that exploiting the a-priori information contained in the trips definition and on the *RV-Graph* and all the data structures related to it, like the *kind_of_link* matrix, will save a lot of computation time to the algorithm.
The pseudocode contained in the Algorithm 3 regards generation of trips of size *three*. At lines 6 and 7 we use a random sampling function to decrease the number of trips of size *two* that will be used to investigate for trips of size *three*.

Algorithm 3 Size three trips generation

for each vehicle $v \in \mathcal{V}$ **do**
  Generate trips of size *one*:...
  Generate trips of size *two*:...
  Generate trips of size *three*:
  if T2_list[v] $\neq \emptyset$ **then**
    if T2_list[v].size $> n_2$ **then**
      T2_list[v] $\leftarrow$ `sample`(T2_list[v], n_2)
    for each $T_1 \in$ T1_list[v] **do**
      for each $T_2 \in$ T2_list[v] **do**
        if $T_1 \cap T_2 = \emptyset$ **then**
          $\{r_1, r_2, r_3\} = T_1 \cup T_2$
        if $\forall i = 1, \ldots, 3 \ : \ \{r_1, r_2, r_3\} \setminus r_i \in$ T2_list **then**
          pickup_order $= \emptyset$, dropoff_order $= \emptyset$
          if $e(r_3, r_1)$ and $e(r_3, r_2) \in \mathcal{E}_{RV\text{-}Graph}$ **then**
            pickup_order.`push`$(\{r_3, r_1, r_2\})$
          if $e(r_3, r_1)$ and $e(r_3, r_2) \in \mathcal{E}_{RV\text{-}Graph}$ **then**
            pickup_order.`push`$(\{r_1, r_3, r_2\})$
          if $e(r_3, r_1)$ and $e(r_3, r_2) \in \mathcal{E}_{RV\text{-}Graph}$ **then**
            pickup_order.`push`$(\{r_1, r_2, r_3\})$
          if $T_{12} \ : \ 1 \rightarrow 2 \rightarrow 1' \rightarrow 2'$ **then**
            if $\{r_3, r_1, r_2\}$ = valid in *kind_of_link* $\Rightarrow$ dropoff_order.`push`
            if $\{r_1, r_3, r_2\}$ = valid in *kind_of_link* $\Rightarrow$ dropoff_order.`push`
            if $\{r_1, r_2, r_3\}$ = valid in *kind_of_link* $\Rightarrow$ dropoff_order.`push`
          if $T_{12} \ : \ 1 \rightarrow 2 \rightarrow 2' \rightarrow 1'$ **then**
            if $\{r_3, r_2, r_1\}$ = valid in *kind_of_link* $\Rightarrow$ dropoff_order.`push`
            if $\{r_2, r_3, r_1\}$ = valid in *kind_of_link* $\Rightarrow$ dropoff_order.`push`
            if $\{r_2, r_1, r_3\}$ = valid in *kind_of_link* $\Rightarrow$ dropoff_order.`push`
          if *travel_size_three*$(v,$ pickup_order, dropoff_order$)$ = valid **then**
            if $T_3 (= T_1 \cup T_2) \in \mathcal{T}_3$ **then**
              Generate $e(T_3, v)$ in *RTV-Graph*
            else
              $\mathcal{T}_3 \leftarrow T_3 = T_1 \cup T_2$
              Generate $e(r_i, T_3), \forall r_i \in T_3,$ and $e(T_3, v)$ in *RTV-Graph*

Another trick we use to decrease the computation time of function *travel_size_three()* is to quit the function as soon as a size *three* trip is found for the possible combinations. This avoid to check for all the possible combinations within the function. This heuristic will make us save some computation time at the cost of loosing some optimality. After the computation of the trips of size three we end up having a set of feasible trips $\mathcal{T} = \{\mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3\}$ where $\mathcal{T}_1$ is the set of trips of size one, $\mathcal{T}_2$ is the set of trips of size two and $\mathcal{T}_3$ is the set of trips of size three.
It can be that a set $T \in \mathcal{T}$ can be served by more then one vehicle, because the route results feasible for more than a vehicle.
Every different *trip - vehicle* edge $e(T, v)$ will be considered as a *candidate assignment* for our assignment problem.
For instance, if T_1 is linked to both v_1 and v_2, there will be two candidate assignments related to T_1, and these will be represented by the *RTV-Graph* links $e(T_1, v_1)$ and $e(T_1, v_2)$.

The next phase is related to the assignment of the feasible trips to vehicles, while satisfying a set of constraints that tells what trips can be simultaneously assigned or not.
We don not want two different trips that contains the same task to be assigned at the same moment. This would result in a system inefficiency, since some other tasks may have been discarded to accomodate that assignment.
It is important to build carefully the set of constraints of our problem so that to avoid failures in the assignment.

3.5 Optimization problem for the optimal vehicle assignment

3.5.1 Formulation of the ILP optimization problem

The assignment of the vehicles to the trips is obtained through the solution of an optimization problem. The goal of a generic optimization problem is to make the best choice of a vector from a set of candidate choices. Firm requirements or specifications limit the possible choices and the objective value.

Example. The following is the generic formulation of an optimization problem from [17].

$$\begin{aligned} &\underset{x}{\text{minimize}} && f_0(x) \\ &\text{subject to:} && f_i(x) \leq b_i, i = 1, \ldots, m \end{aligned} \tag{3.12}$$

Where the vector $x = (x_1, ..., x_n)$ is the *optimization variable* of the problem, the function f_0 is the *objective* function, the functions $f_i, i = 1, \ldots, m$, are the (inequality) *constraint functions* and the constants $b_1, \ldots, b_m$ are the limits, or bounds, for the constraints. A vector x^* is called *optimal*, or a *solution* of the problem 3.12, if it has the smallest objective value among all vectors that satisfy the constraints.

Due to the integrality requirement on the variables of our problem, we will need to solve an ILP (Integer Linear Programming). We want to introduce now how to build specifically the problem regarding the task assignment from the knowledge of the *RTV-Graph*. Also, we will have an insight on how the ILP is formulated and how it is solved in general.

Problem variables

Our goal is to find the best vector of assignments out of the many possible feasible combinations, so that to minimize the cost related to the global travel time of our application.
To this aim, a vector of binary variables $\epsilon_{i,j} = \{0, 1\}$ is introduced for each edge $e(T_i, v_j)$ between a trip $T_i \in \mathcal{T}$ and a vehicle $v_j \in \mathcal{V}$ in the *RTV-Graph* [5]. We denote with $\mathcal{E}_{TV}$ the set of indexes for which an edge $e(T_i, v_j)$ exists in the *RTV-Graph*.
After the optimization problem is solved, we will have:

$$\epsilon_{i,j} = \begin{cases} 1 & \text{if vehicle } v_j \text{ is assigned to trip } T_i \\ 0 & \text{if the } \textit{candidate} \text{ assignment is not assigned} \end{cases}$$

Ideally speaking, all the requests shall be assigned to a vehicle, but given the constraints, this might not always be the case. Therefore, we define an additional vector of binary variables χ_k that has an element for each request $r_k \in \mathcal{R}$. We will have:

$$\chi_k = \begin{cases} 1 & \text{if request } r_k \text{ is not assigned in any of the trips in } \mathcal{T} \\ 0 & \text{if request } r_k \text{ is assigned} \end{cases}$$

We can now define the set of variables:

$$\mathbf{X} = \{\epsilon_{i,j},\ \chi_k;\ \forall e(T_i, v_j) \text{ edge in } \textit{RTV-Graph},\ \forall r_k \in \mathcal{R}\} \tag{3.13}$$

Because our optimization variable is constrained to the values $\{0, 1\}$ we call this problem a binary linear programming that is a special case of the integer linear programming (ILP) problems.
Integer linear programming are very complex problem and are classified as $\mathcal{NP}$-complete problems. The problem therefore belongs to the hardest problems in the complexity $\mathcal{NP}$ [85].

Problem constraints

In our optimization problem, two types of constraints are present. One regards the fact that the vehicles cannot serve more than one trip at a time (remember that trips are already a combination of the different requests when possible). The other constraints are saying that a request can be assigned to one single trip at most, or ignored.
Solutions where requests are ignored exist, but our goal will be to try to have the most number of requests satisfied.
To formalize the first constraint we say that:

$$\sum_{i \in \mathcal{I}_j^V} \epsilon_{i,j} \leq 1\ \forall v_j \in \mathcal{V} \tag{3.14}$$

where $\mathcal{I}_j^V$ is the set of indexes i for which an edge $e(T_i, v_j)$ exists in the *RTV-Graph*.
We can write the constraints in matrix form, that is the form used in most of the solvers:

$$\begin{bmatrix} A_1 \mid 0_{m \times n} \end{bmatrix} \begin{bmatrix} \epsilon_{i,j} \\ \chi_k \end{bmatrix} \leq 1_{m \times 1} \longrightarrow A_{1,extended} \begin{bmatrix} \epsilon_{i,j} \\ \chi_k \end{bmatrix} \leq 1_{m \times 1} \tag{3.15}$$

where A_1 is a rectangular matrix of dimensions $m \times l$ with m the number of vehicles and l the length of the vector of variables $\epsilon_{i,j}$ that is the number of edges $e(T_i, v_j)$ in the *RTV-Graph* for our instance of the problem. Notice that formulation 3.15 is considering also variables χ_k. This because we need to consider also variables χ_k when building the constraints in matrix form, because the vector of variables is $X = [\epsilon_{i,j}; \chi_k]$, even if the first of the two sets of contraints does not concern those variables. Therefore, we need to consider a matrix $A_{1,extended}$ of size $m \times (l + n)$ that accounts also for additional columns of zeros, to match the size $(l + n)$ of the variables vector X.
We want now to show with an exmaple how matrix A_1 is built so that to represent the constraints of our problem.

Example. Here we show an example of the implementation of matrix A_1 for representing the first set of constraints in a matrix formulation.

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_m \end{array} \begin{array}{c} \begin{array}{cccccccccc} e(T_1,v_1) & e(T_1,v_2) & e(T_2,v_1) & e(T_3,v_2) & e(T_3,v_m) & \cdots & e(T_{10},v_1) & \cdots & e(T_t,v_2) \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & \dots & 1 & \cdots & 0 \\ 0 & 1 & 0 & 1 & 0 & \dots & 0 & \cdots & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 & \dots & 0 & \cdots & 0 \end{pmatrix} \end{array}$$

And the vector of variables ϵ is:

$$\epsilon = \begin{pmatrix} e(T_1,v_1) \\ e(T_1,v_2) \\ e(T_2,v_1) \\ e(T_3,v_2) \\ e(T_3,v_m) \\ \vdots \\ e(T_{10},v_1) \\ \vdots \\ e(T_t,v_2) \end{pmatrix}$$

We can see that matrix A_1 associates row elements (that are vehicles) to column elements (that are candidate solutions). When the vehicle v_j, $i = 1, \dots, m$, is appearing in the candidate solution ϵ_c, $c = 1, \dots, l$ then the entry $A_{1_{ij}}$ will be equal to one.
We will need later to add n columns of zeros to matrix A_1 so that to take into account the n χ variables, which represent whether tasks have been assigned or not. The presence of these columns is due to the fact that sizes of the matrixes have to match. Also the variables vector will be of size $l + n$.
Therefore, we are describing a set of inequalities that tells that the sum of the assignments related to a single vehicle, can be at most 1.
For our specific example we have the following set of inequalities:

$$\begin{aligned} & e(T_1,v_1) + e(T_2,v_1) + \ldots + e(T_{10},v_1) + \ldots \leq 1 \\ & e(T_1,v_2) + e(T_3,v_2) + \ldots + e(T_{10},v_2) \leq 1 \\ & \vdots \\ & e(T_3,v_m) + \ldots \leq 1 \end{aligned} \tag{3.16}$$

This constraint will make sure that none of the vehicles is assigned to more than one trip.

To formalize the second constraint we say:

$$\sum_{i \in \mathcal{I}_k^R} \sum_{j \in \mathcal{I}_i^T} \epsilon_{i,j} + \chi_k = 1 \; \forall r_k \in \mathcal{R} \tag{3.17}$$

where $\mathcal{I}_k^R$ is the set of indexes k for which the edge $e(r_k, T_i)$ exists in the *RTV-Graph*. We remind that an edge $e(r_k, T_i)$ exists between a task r_k and a trip T_i when $r_k \in T_i$.
$\mathcal{I}_i^T$ represents the indexes j for which an edge $e(T_i, v_j)$ exists in the *RTV-Graph*. An edge $e(T_i, v_j)$ exists between the vehicle v_j and the trips it can serve.

For this set of constraints, having two sum operations, we decide to consider a matrix product to be able to represent the constraints in matrix form. The product of the two matrices will be a new matrix that will be used in the matrix representation of the set of constraints.
We can represent the set of constraints in matrix form as:

$$\left[A_{I^R}\right] \left[A_{I^T,\, extended}\right] \begin{bmatrix} \epsilon_{i,j} \\ \chi_k \end{bmatrix} = 1_{n \times 1} \longrightarrow \left[A_{I^R}\right] \left[A_{I^T} \mid 0_{t \times n}\right] \begin{bmatrix} \epsilon_{i,j} \\ \chi_k \end{bmatrix} = 1_{n \times 1} \tag{3.18}$$

where matrix A_{I^R} is used to account for the $\mathcal{I}_k^R$ indexes and matrix A_{I^T} is used to account for the $\mathcal{I}_i^T$ indexes.
Matrix $A_{I^T,\, extended}$ is a block matrix composed of the constraint matrix A_{I^R} and by a $t \times (l + n)$ matrix of *zeros*.
We deal first with the inner summation of the constraint equations and therefore with matrix $A_{I^T,\, extended}$. We have that indeces $\mathcal{I}_i^T$ tells when an edge of the *RTV-Graph* is linked to a trip $T \in \mathcal{T}$. This means that the matrix representing the sum operation will be a rectangular matrix of size $t \times (l + n)$, where t is the total number of trips of the *RTV-Graph*, l is the number of edges in the *RTV-Graph* (which represents the number of candidate assignments) and n is the number of tasks of our problem, that is also the dimension of the vector of additional variables χ. $l + n$ will be the total number of variables of our problem. Therefore, the first l variables represent the assignments and the last n variables represent assigned/unassigned tasks.
Following, we show with an example how matrix $A_{I^T,\, extended}$ is defined.

Example. In this example we have a set of trips $T \in \mathcal{T}$ and a set of edges $e(T_i, v_j) \in \mathcal{E}$. We want to show how the matrix relates the two sets.

$$
\begin{array}{c}
 \\
T_1 \\
T_2 \\
T_3 \\
\vdots \\
T_{10} \\
\vdots \\
T_t
\end{array}
\begin{array}{c}
\begin{array}{cccccccc}
e(T_1,v_1) & e(T_1,v_2) & e(T_2,v_1) & e(T_3,v_m) & \cdots & e(T_t,v_2) & r_1 & \cdots \quad r_n
\end{array} \\
\left(\begin{array}{cccccccc}
1 & 1 & 0 & 0 & \dots & 0 & 0 & \cdots \quad 0 \\
0 & 0 & 1 & 0 & \dots & 0 & 0 & \cdots \quad 0 \\
0 & 0 & 0 & 1 & \dots & 0 & 0 & \cdots \quad 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & 0 & \ddots \quad 0 \\
0 & 0 & 0 & 0 & \dots & 0 & 0 & \cdots \quad 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & 0 & \ddots \quad 0 \\
0 & 0 & 0 & 0 & \dots & 1 & \underbrace{0 \quad \cdots \quad 0}_{0_{t\times n}} &
\end{array}\right)
\end{array}
$$

The columns of A_{I^T} are the edges of the *RTV-Graph*, the rows are all the trips $\mathcal{T}$ of the *RTV-Graph*. The entries of A_{I^T} are one when the trip T_i, $i = 1, \dots, t$, is contained in the candidate assignment represented by columns $c = 1, \cdots, l$. The last n columns of matrix $A_{I^T,extended}$ represent what tasks of our problem have been assigned. There is no direct relation between the trips $\mathcal{T}$ and these variables as we can see from equation 3.19. The only reason why these columns of *zeros* are present is because we need consistency in the matrices sizes when building the matrix formulation of our set of constraints.

The outer summation of the constraint is represented by matrix A_{I^R} that is related to indexes $\mathcal{I}_k^R$. A_{I^R} is a rectangular matrix of size $n \times t$ where n is the total number of the tasks and t is the toal number of trips in the *RTV-Graph*. We show with an example how matrix A_{I^R} is shaped.

Example. In this example we want to show how matrix A_{I^R} that relates trips $\mathcal{T}$ and tasks $\mathcal{R}$ is shaped.

$$
\begin{array}{c}
 \\
r_1 \\
r_2 \\
r_3 \\
\vdots \\
r_n
\end{array}
\begin{array}{c}
\begin{array}{ccccc}
T_1 & T_2 & T_3 & \cdots & T_t
\end{array} \\
\left(\begin{array}{ccccc}
1 & 1 & 0 & \dots & 0 \\
0 & 0 & 1 & \dots & 1 \\
1 & 0 & 1 & \dots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 1 & \dots & 1
\end{array}\right)
\end{array}
$$

The columns of A_{I^R} are all the trips $T_i \in \mathcal{T}$ for $t = 1, \dots, t$ in the *RTV-Graph* while the rows of the matrix are the requests $r_k \in \mathcal{R}$ for $k = 1, \dots, n$. This means that the entries of the matrix are one when the task on the k-th row is part of the trip in the i-th column and they are zero otherwise.
Notice that trip T_1 is at least a trip of size two because from matrix A_{I^R} we know that $T_1 = \{r_1, r_3, \dots\}$, T_2 is at least a trip of size one because $T_2 = \{r_1, \dots\}$ and T_3 is a trip of size three $T_3 = \{r_2, r_3, r_n\}$.

Once we have the two matrices A_{I^T} and $A_{I^R,\,extended}$ we can do the matrix product to find the direct relation between the vector of the variables and the known vector:

$$\left[A_{I^R}\right]\left[A_{I^T,\,extended}\right]\begin{bmatrix}\epsilon_{i,j}\\ \chi_k\end{bmatrix} = 1_{n\times 1} \longrightarrow A_2\begin{bmatrix}\epsilon_{i,j}\\ \chi_k\end{bmatrix} = 1_{n\times 1} \tag{3.19}$$

Now we know how to build the matrixes A_1 and A_2 to define the constraints of our problem.
One set of constraints is left, and it is stating that all the variable can only be either *zero* or *one*. Because of these constraints the problem is an *Integer Linear Programming* Problem (ILP) and more specifically a *binary linear programming* problem. The set of constraints is expressed by:

$$\mathbf{X} = \{x : x \in \{0, 1\}\} \tag{3.20}$$

Problem cost function

The objective of our algorithm is to minimize the cost that is defined by the travel time of all the different assigned vehicles once a trip is assigned. The cost function may be any mix of travel time, distance, toll, energy consumpiton etc. associated with the edges [33]. Another objective that usually is used is to consider the delay on the original planned arrival time before a ride-sharing option was proposed to the rider, for each of the rides composing the trip. This cost would be given by:

$$c_{i,j} = \sum_{r\in T_i}(t_r^d - t_r^*) \tag{3.21}$$

where t_r^d is the drop-off time for task r and t_r^* is the best possible arrival time for that task (that previously we considered being the arrival time at_r). $c_{i,j}$ is computed for each of the candidate assignments $e(T_i, v_j)$. It means that each candidate assignment is associated with a cost by definition. We will generally use that cost to build the cost function.
We decided instead to consider as cost of the candidate assignments the total travel time from the location of the vehicle at the time of the algorithm call until the last passenger of the trip for the specific assignment is dropped-off.
Consider a 2D-vector named *Loc*, composed by the pivot locations defining the route of a trip. These are the vehicle starting location coordinates v_{xy}, the pick-ups coordinates $o_{1,xy}$ and $o_{2,xy}$ and the drop-offs coordinates $d_{1,xy}$ and $d_{2,xy}$ of the two tasks. We can define the cost $c_{i,j}$ as:

$$c_{i,j} = \sum_{i=0}^{|Loc|-1} tt(Loc_{i+1} - Loc_i) \tag{3.22}$$

Where $|Loc|$ is the length of the vector and depends on the size of the trip, since trips composed of more rides have more pivot points. $tt()$ is the function computing the travel time between two location coordinates.

We define now the cost function of our problem, where $c_{i,j}$ and the variables vectors

$\epsilon_{i,j}$ and $\chi_{i,j}$ have already been defined:

$$\mathcal{C}(\mathcal{X}) = \sum_{i,j \in \mathcal{E}_{TV}} c_{i,j}\epsilon_{i,j} + \sum_{k \in \{0,\dots,n\}} c_{ko}\chi_k \tag{3.23}$$

This cost function wants to be a tradeoff between the total time spent on travelling by the vehicles (that has to be minimized) and the fact that we want as many requests satisfied as possible. The second term of the cost function is assigning a cost c_{ko} to each unassigned request (when χ_k is one). By using a large constant as c_{ko} we want that the quantity $\sum_{k \in \{0,\dots,n\}} c_{ko}\chi_k$ is as small as possible so that the number of assigned requests is as big as possible.

The ILP optimization problem

Once we have defined what are the variables of our problem, what are the constraints and how they are built and what is the cost function of our problem, we can formulate the optimization problem as:

$$\begin{aligned}
\min_{x} \quad & \sum_{i,j \in \mathcal{E}_{TV}} c_{i,j}\epsilon_{i,j} + \sum_{k \in \{0,\dots,n\}} c_{ko}\chi_k \\
\text{subject to:} \quad & \sum_{i \in \mathcal{I}_j^V} \epsilon_{i,j} \leq 1, && \forall v_j \in \mathcal{V} \\
& \sum_{i \in \mathcal{I}_k^R} \sum_{j \in \mathcal{I}_i^T} \epsilon_{i,j} + \chi_k = 1 && \forall r_k \in \mathcal{R} \\
& \epsilon_{i,j}, \chi_k \in \{0,1\}
\end{aligned} \tag{3.24}$$

The problem is solved with IBM ILOG CPLEX optimizer [31], that provides a set of libraries for the construction of the problem and provide tools for the solution of the problem. The CPLEX algorithm makes use of one of the most used algorithms for MILP problems which is the *Branch-and-Cut* algorithm. The next paragraph will treat what is the branch-and-cut algorithm, and what is the theory behind its working procedures.

3.5.2 Methods for solving integer linear programs

In this paragraph we want to introduce the currently most successful method for solving *integer linear programming* (ILP) problems, that is the so called *Branch-and-Cut* algorithm. Most of the theoretical aspects are taken form [27]. For convenience, most of the literature assumes the more general case of *mixed-integer linear programming* (MILP) problems, where part of the variables are constrained to be nonnegative integers, while other variables are allowed to take any nonnegative real value.
Solving integer programs is a difficult task in general since, as we anticipated, they are considered $\mathcal{NP}$-complete problems. In the case for example of *binary integer linear programming* problems, where the variables are constrainted to take value 0 or 1, one possible approach could be of trying all the possible combinations of the variables, since they are restricted to a finite number of values. Even if logically simple, this is applicable in theory but is not practical when the number of variables is large. One approach commonly used is to find a relaxation of the problem that is numerically easier to solve and that gives a good approximation of the solution.

Linear programming relaxations are mainly used because of its success in computational mathematics and the high results achieved in solving them. Relaxations can be solved in reasonable time because of the efficiency of the algorithms that exist nowadays. Another reason why to use linear programming relaxations is that we can generate a sequence of linear relaxations of our set that provide increasingly tighter approximations.
We now give a formulation of the generic MILP problem:

$$\begin{aligned} \max_{x} \quad & cx + hy \\ \text{subject to:} \quad & Ax + Gy \leq b \\ & x \geq 0 \text{ integral} \\ & y \geq 0, \end{aligned} \tag{3.25}$$

The feasible set of solutions of 3.25 will be the *mixed integer set*:

$$S = \{(x, y) \in \mathbb{Z}^n_+ \times \mathbb{R}^p_+ \ : \ Ax + Gy \leq b\} \tag{3.26}$$

where n is the number of the x nonnegative integer variabels and p is the number of the y nonnegative real variables.
For the case of *binary integer linear programming* we will have a *mixed* 0,1 *linear set* that is in general:

$$S = \{(x, y) \in \{0, 1\}_n \times \mathbb{R}^p_+ \ : \ Ax + Gy \leq b\} \tag{3.27}$$

The so called *natural linear relaxation*, is the linear relaxation of the set S, and this is:

$$P_0 = \{(x, y) \in \mathbb{R}^n_+ \times \mathbb{R}^p_+ \ : \ Ax + Gy \leq b\} \tag{3.28}$$

that is obtained simply discarding the integrality constraint for the variables x. Thus, it derives also a *natural linear programming relaxation* of the MILP problem 3.25 and this is, in short: $\max\{cx + hy : (x, y) \in P_0\}$.

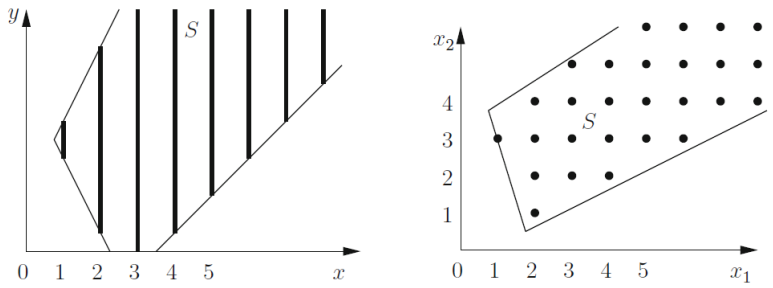


Figure 3.4: Example of mixed integer linear set and of a pure integer linear set, where all the variables are constrained to take integer variables.

Branch-and-cut for solving the MILP

The CPLEX solver uses *Branch-and-Cut* search when solving *mixed integer linear programming* models. The branch-and-cut procedure manages a search tree consisting of

nodes. To speak of the branch-and-cut algorithm is better first to introduce the so called *Branch-and-Bound* algorithm. Branch and cut involves running a branch-and-bound algorithm and using cutting planes to tighten the linear programming relaxations. Branch-and-bound and the cutting plane methods are based on simple ideas but they are at the heart of the state-of-the art software for integer programming.
We now write in short the problem formulated in 3.25 denoting it as MILP:

$$\text{MILP}: \ \max\{cx + hy : (x, y) \in S\} \tag{3.29}$$

where S as before is $S = \{(x, y) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p \ : \ Ax + Gy \leq b\}$. We assume that our problem admits a finite optimum, to ease the explanation. We then call (x^*, y^*) the optimal solution of our problem and z^* the value of the cost function at the optimum.
Let then (x^0, y^0) and z_0 be the optimal solution and the optimal vlaue of the natural linear programming relaxation of our MILP problem. The relaxation is:

$$\text{relaxed MILP}: \ \max\{cx + hy : (x, y) \in P_0\} \tag{3.30}$$

where $P_0 = \{(x, y) \in \mathbb{R}_+^n \times \mathbb{R}_+^p \ : \ Ax + Gy \leq b\}$ is the natural relaxation of the set S. (x^0, y^0) and z_0 are computed with a generic LP solver. Since P_0 is a relaxation of the set of integers S, we have $S \subseteq P_0$ and therefore $z^* \leq z_0$. This beacause the range of possible solutions in the relaxed set P_0 is bigger than the set of solutions in S, therefore the value z_0 of the cost function in P_0 will be at least equal to the value z^* in S, if not greater (in case of maximization).
If solution x^0 is an integral vector, then $(x^0, y^0) \in S$ and therefore $z^* = z_0$ and the MILP would be solved. Instead, if at least one of the components of x^0 is fractional (not an integer), then we need to use one of the algorithm proposed for solving MILP.

We call $x = (x_1, \ldots, x_n)$ the vector of variables of our problem that are constrainted to be integer. Since the vector x^0 of solution we have found for the LP relaxation is fractional (which means that at least one of its elements is fractional) we pick one of its elements of index j, where $1 \leq j \leq n$ such that x_j^0 is fractional. We call this fractional value $f = x_j^0$ and we define the sets:

$$S_1 = S \cap \{(x, y) : x_j \leq \lfloor f \rfloor\} \text{ and } S_2 = S \cap \{(x, y) : x_j \geq \lceil f \rceil\} \tag{3.31}$$

where $\lfloor f \rfloor$ denotes the largest integer $k \leq f$ and $\lceil f \rceil$ denotes the smallest integer $l \geq f$.
We can now define two new mixed integer linear programming problems MILP_1 and MILP_2:

$$\text{MILP}_1: \ \max\{cx + hy : (x, y) \in S_1\} \text{ and } \text{MILP}_2: \ \max\{cx + hy : (x, y) \in S_2\} \tag{3.32}$$

Since the solution of the MILP is the best between the solutions of the two problems MILP_1 and MILP_2, then we will say that the solution of the original MILP is reduced to the solution of two subproblems.
As we have done for the original MILP, we can relax the new problems MILP_1 and MILP_2 by relaxing the sets S_1 and S_2 with the natural linear relaxations P_1 and P_2. These new

relaxed sets will be:

$$P_1 = P_0 \cap \{(x, y) : x_j \leq \lfloor f \rfloor\} \text{ and } P_2 = P_0 \cap \{(x, y) : x_j \geq \lceil f \rceil\} \tag{3.33}$$

And the two corresponding natural linear programming relaxations of the MILP subproblems will be:

$$\text{LP}_1 : \max\{cx + hy : (x, y) \in P_1\} \text{ and } \text{LP}_2 : \max\{cx + hy : (x, y) \in P_2\} \tag{3.34}$$

Now we will procede considering the following criteria as described in [27]:

1. If one of the linear porgrams LP_i is infeasible, i.e., $P_i = \emptyset$, then we also have $S_i = \emptyset$ since $S_i \subseteq P_i$. Thus MILP_i is infeasible and does not need to be considered any further. We say that this problem is *pruned by infeasibility*.
2. Let (x^i, y^i) be an optimal solution of LP_i and z_i its value, $i = 1, 2$.
   (a) If x^i is an integral vector, then (x^i, y^i) is an optimal solution of MILP_i and a feasible solution of MILP. Problem MILP_i is solved, and we say that it is *pruned by integrality*. Since $S_i \subseteq S$, it follows that $z_i \leq z^*$, that is, z_i is a lower bound on the value of MILP. This is true because the set S that contains S_i has more possible solution so the cost function can improve.
   (b) If x^i is not an integral vector and z_i is smaller than or equal to the best known lower bound on the value of MILP, then S_i cannot contain a better solution and the problem is *pruned by bound*.
   (c) If x^i is not an integral vector and z_i is greater than the best known lower bound, then S_i may still contain optimal solution to MILP. Let x^i_j, be a fractional component of vector x^i. Let $f' = x^i_j$, define the sets $S_{i1} = S_i \cap \{(x, y) : x_{j'} \leq \lfloor f' \rfloor\}$, $S_{i2} = S_i \cap \{(x, y) : x_{j'} \geq \lceil f' \rceil\}$ and repeat the above process.

By following these rules, the procedure is searching for an optimal solution by *branching*, that is, partitioning the set S in smaller subsets with the goal of bounding the objective value of the suproblems.
Bounding the objective value of a subproblem is done by solving its natural linear programming relaxation. This is called *linear programming bounding*.
The branch-and-bound algorithm has a list of linear programming problems obtained by relaxing the integrality requirements on the variables x_j, $j = 1, \ldots, n$ and imposing linear constraints, such as bounds on the variables $x_j \leq u_j$ or $x_j \geq l_j$. Each such linear program corresponds to a node of the enumeration tree.

A *branch* is the creation of two new nodes from a parent node. A branch occurs when the bounds on a single variable are modified, with the new bounds remaining in effect for that new node and for any of its descendants. For example, if a branch occurs on a binary variable, that is, one with a lower bound of 0 and an upper bound of 1, then the result will be two new nodes, one node with a modified upper bound of 0 (the downward branch, in effect requiring this variable to take only the value 0), and the other node with a modified lower bound of 1 (the upward branch, placing the variable at 1). The two new nodes will thus have completely distinct solution domains.

Denote with $\mathcal{L}$ the list of nodes that must still be solved (not pruned nor branched), N_i is the i-th node and N_0, called the *root node* is the node associated with the original linear programming relaxation. z_i is the optimal value associated with LP_i relaxation of node N_i, and $\underline{z}$ is a lower bound on the optimum value z^*. Good lower bounds on $\underline{z}$ can be derived by any heuristic method for the solution of the original MILP.
This process implies the construction of a search tree. Due to the exponential growth in the size of such a tree, exhaustive enumeration would quickly become hopelessly computationally expensive for MILP problems with even dozens of variables. The effectiveness of the branch-and-bound algorithm depends on its ability to prune nodes [54]. This is why in the branch-and-bound approach, the tightness of the upper bound is crucial for pruning the enumeration three. Tighter upper bounds can be calculated by applying the cutting plane approach to the subproblems. This leads to the *Branch-and-Cut* method. Before the branching phase usual of the branch-and-bound procedure, a cutting-plane is added.
A cutting-plane term derive from the definition [27] of a new inequality that is added into our problem. The idea is to find an inequality $\alpha x + \gamma y \leq \beta$ such that it is satisfied by every poitnt in S and such that $\alpha x^0 + \gamma y^0 > \beta$, when $x^0 \notin S$.
An inequality $\alpha u \leq \beta$ is *valid* for a set $K \subseteq \mathbb{R}$ if it is satisfied by every point $\bar{u} \in K$. A valid inequality $\alpha x + \gamma y \leq \beta$ for S that is violated by (x^0, y^0) is a *cutting plane* separating (x^0, y^0) from S. Let $\alpha x + \gamma y \leq \beta$ be a cutting plane and define:

$$P_1 = P_0 \cap \{(x, y) : \alpha x + \gamma y \leq \beta\} \tag{3.35}$$

Since $S \subseteq P_1 \subset P_0$, the linear programming relaxation of MILP based on P_1 is stronger than the natural linear programming relaxation used in the branch-and-bound approach, in the sense that the optimal value of the linear program:

$$\max\{cx + hy : (x, y) \in P_1\} \tag{3.36}$$

is at least as good as z_0 as an upper-bound on the value z^*, while the optimal solution (x^0, y^0) of the natural linear programming relaxation we know does not belong to P_1 because of the cutting-plane. We want to generalize the past definition to be repeated iteratively, where the root node will generally be the i-th node, and the child branches will be the $(i + 1)$-th. In practice it is good to generate multiple cutting-planes to separate (x^i, y^i) from the set S. All of the cutting planes will be used to create the successive subsets.

In practice the cutting-plane is called *cut* and this is a constant added to the model with the purpose of limiting the size of the solution domain for the continuous LP or QP problems (problem relaxations) represented at the nodes, while not eliminating legal integer solutions. The outcome is thus to reduce the number of branches required to solve the MILP. There exist both *global cuts* and *local cuts*. A global cut is a cut that is valid for all nodes of the branch-and-bound tree, even if that global cut was found during the analysis of a particular node. In contrast, a local cut is a cut that is valid only for a specific node and for all of its descentant nodes.
The branch-and-cut procedure, consists of performing branches and applying cuts at the nodes of the tree.

Branch-and-Cut Algorithm

1. *Initiate*
   $\mathcal{L} = \{N_0\}, \underline{z} = -\infty$ or *initial guess*, $(x^*, y^*) = \emptyset$.

2. *Terminate?*
   if $\mathcal{L} = \emptyset$, the solution (x^*, y^*) is optimal.

3. *Select node*
   Choose a node N_i in $\mathcal{L}$ and delete it from $\mathcal{L}$.

4. *Bound*
   Solve LP_i. If it is infeasible, go to Step 1. Else, let (x^i, y^i) be an optimal solution of LP_i and z_i its objective value.

5. *Prune*
   If $z_i \leq \underline{z}$ go to Step 1.
   If (x^i, y^i) is feasible to MILP, set $\underline{z} = z_i, (x^*, y^*) = (x^i, y^i)$ and go to Step 1.

   Otherwise:

6. *Add cuts?*
   Decide whether to strengthen the formulation LP_i or to branch.
   In the first case strengthen LP_i by adding cutting planes and go back to Step 3.
   In the second case, go to Step 6.
   The decision of whether to add cuts or not is made empirically, basing the choice on the success of previously added cuts and the characteristics of the new cuts such as their density.

7. *Branch*
   From LP_i, construct $k \geq 2$ linear programs $LP_{i_1}, \ldots, LP_{i_k}$ with smaller feasible regions whose union does not contain (x^i, y^i), but contains all the solutions of LP_i with $x \in \mathbb{Z}^n$. Add the corresponding new nodes $N_{i_1}, \ldots, N_{i_k}$ to $\mathcal{L}$ and go to Step 1.

CPLEX heuristics

In practice, CPLEX will operate as the algorithm proposed above, but it implements some additional features to speed up computation, so trying to give an optimal solution, or a good approximation quickly.
When processing a node, CPLEX starts by solving the continuous relaxations (LP) of its subproblems, that are the subproblems with no integrality constraints, as extensively explained above. If the solution violates any cuts, CPLEX may add some or all of the to the node problem and may resolve it, if CPLEX has added cuts. This procedure is iterated until no more violated cuts are detected by the algorithm. If at any point in the addition of cuts the node become infeasible, the node is pruned (as in Step 5). Otherwise (as the second condition of Step 5) CPLEX checks whether the solution of the node-problem satisfies the integrality constraints. If so, and if the objective value is better than that of the current bound $\underline{z}$, the solution of the node-problem is used as the new $\underline{z}$. If not, branching will occur (Step 7), but first a heuristic method may be tried at this point to see if a new incumbent can be inferred form the LP-QP solution at this node.
At any point in the algorithm there is a node N_i whose z_i is better (less, in the case of minimization problem, or greater for a maximization problem) than all the other nodes. The best node value can be compared to the objective function value of the current best

solution. The resulting MILP gap, expressed as a percentage of the best solution, serves as a measure of progress toward finding and providing optimality. The two values will converge while the algorithm iterates, and thus the gap will become zero. CPLEX stops when the gap is lower than 0.01%, that is sooner than a completed proof of optimality. Going on with the computation will require additional computaion time with low increase in the quality of the solution.

In CPLEX, *heuristics* are procedures that try to produce good or approximate solutions to a problem in few time but which lack theoretical guarantees. When solving a MILP, a heurisitc method may produce one or more solutions that satisfy all the constraints of the problem but which lack an indication of whether it has found the best solution possible. Heuristics can speed up the final proof of optimality, or they can provide a sub-optimal but high-quality solution in a shorter amount of time than by branching alone. For example, *node heuristics* apply heuristic approach to find a feasible solution to the branch-and-cut node. RINS (Relaxation Induced Neighborhood Search) is a heuristic that explores a neighborhood of the current incumbent (the current best solution) solution (it implements a local search) to try to find a new improved bound. The neighborhood is constructed using information contained in the continuous relaxation of the MILP mode. Neighborhood search is formulated as a MILP model itself and solved recursively [32].

Solvers can solve the problem even when a given starting point is not a feasible solution to our problem or it is incomplete. They will first of all test for the feasibility of the starting point, and if not feasible they will search for a feasible starting point in an iterative fashion.
When providing a starting point to a MILP, CPLEX will process it before starting the branch-and-cut. If the starting points (it is possible to give more than one) define a solution to the MILP, then CPLEX will use the best of these solutions as the incumbent solution, that is the best solution present for the kind of problem (maximization or minimization).
The advantage of having a feasible starting solution from the beginning of the the branch-and-cut algorithm call is that CPLEX can eliminate portion of the search space and thus it may result in smaller branch-and-cut trees. Moreover, having an incumbent also allows CPLEX to use heuristics which require an incumbent, such as Relaxation Induced Neighborhood Search (RINS heuristic).
To this aim, in order to give to the algorithm a good starting point for the solution of the MILP, we used a feasible starting solution that also tries to satisfies the criteria on which our optimization problem is based. A greedy assignment [5] has been designed for planning a first coarse assignment algorithm that still respect the constraints of our problem.

3.5.3 A greedy assignment as a starting solution for the ILP

A first heuristic assignment wants to match vehicles with trips in a way that maximizes the number of requests while minimizing the global cost of the assignment. The greedy algorithm is allocating trips to vehicles, assigning trips by decreasing size and by growing cost, from the set of edges $\epsilon_{i,j}$. This means that it will assign trips of size k first, then trips of size $k-1$ and so on, if present. The last trips to be assigned will be trips of size one.
Assigning by increasing cost means that the trips sets will be ordered by increasing cost

and the assignment of the trips of a certain size $f \leq k$ will start from trips that have a lower cost within the trips of the set $\mathcal{T}_f$.
As we assign trips to vehicles we need to keep track of the vehicles that have been assigned and of the tasks contained in the trips that have already been assigned. Doing that way we will guarantee that our solution is feasible and the branch-and-cut algorithm of the CPLEX solver will not need to find for a feasible starting point. Moreover, the cost has been already taken into consideration and the global cost is assumed to be not as far from the optimal solution of the MILP. This will help the algorithm when searching for a solution.

3.6 Vehicles fleet rebalancing phase

Due to imbalances of the distribution of vehicles with respect to the distribution of the service requests, it may happen that some requests remain unassigned during the optimal assignment phase (solution of the MILP) and some vehicle remain idle. Indeed, it was not a constraint of the problem that all requests and vehicles must be assigned. This would be in contrast with the temporal constraints imposed during the matching phase and therefore against constraints expressed by equations 3.14, 3.19.
Therefore, we want to give the possibility to those unassigned tasks to be assigned. It may happen, though, that the assignment of these tasks to vehicles is in contrast with the delay constraints imposed by the parameter Δ in our system. If the task is located in a place of high request, it might be probably served at the next algorithm call or when a new vehicle in its neighbors becomes available. This, even if not always true, is highly probable.
Instead, if the request is located in a place where the service is low (for instance a suburban area), then it is difficult that the request would be reconsidered even at future calls of the optimal assignment.
It is up to the customer to accept an assigned vehicle/route based on his/her common sense to understand that the area where the request has been located is not suitable for being served under strict delay constraints.
As in [5] we call $\mathcal{R}_{ko}$ the set of tasks that have not been assigned during the optimal assignment problem and $\mathcal{V}_{idle}$ the set of vehicles that have remained idle with no trip to serve.
The goal of the rebalancing phase is to assign single tasks to vehicles building a new optimization problem.

Cost function: The first step will be to compute the cost of each r-v possible matching based on the travel time required by the vehicle to move from its current position to the pick-up location of the task and then travel with the passenger on board to the drop-off location of the task.
We compute the cost as in 3.22:

$$c_{v,r} = \sum_{i=0}^{|Loc|-1} tt(Loc_{i+1} - Loc_i) \tag{3.37}$$

where Loc is the 2D-vector of all the pivot points of the route for all the possible v-r combinations.
The cost function is a linear combination of the variables $y_{v,r} \in Y_{v,r}$ by the cost vector $c_{v,r}$ where $Y_{v,r} = \{y_1, \ldots, y_n\}$ that represent all the possible couples v-r. $n = l_{\mathcal{R}_{ko}} \times l_{\mathcal{V}_{idle}}$

is the dimension of the variable vector $l_{\mathcal{R}_{ko}}$ is the number of unassigned tasks and $l_{\mathcal{V}_{idle}}$ is the number of idle vehicles. The cost function will be:

$$\sum_{v \in \mathcal{V}_{idle}} \sum_{r \in \mathcal{R}_{ko}} c_{v,r} y_{v,r} \tag{3.38}$$

Constraints: In this phase we want that each of the idle vehicle has maximum one task assigned and that every task is assigned to one vehicle only. The set of constraints that we will use in order to satisfy these requirements is similar to that described by equations 3.14 and 3.19.
The first of the requirements, the one stating that each of the vehicles must be assigned a maximum of one task, is formalized as follows:

$$\sum_{r \in \mathcal{I}^R_{V=v}} y_{v,r} \leq 1, \ \forall v \in \mathcal{V}_{idle} \tag{3.39}$$

where $\mathcal{I}^R_{V=v}$ is the set of the indeces r of the tasks in $\mathcal{R}$ that have been matched with vehicle $v \in \mathcal{V}$. Because of how we have build the matchings, every vehicle will be linked to every task. With the following example we want to show the matrix formulation of the set of constraints 3.39 and show how the matrix is shaped.

Example. We consider to have a problem with three vehicles $\mathcal{V} = \{v_1, v_2, v_3\}$ and three requests $\mathcal{R} = \{r_1, r_2, r_3\}$. $m = 3$ will be the number of vehicles and $n = 3$ will be the number of requests. The set of constraints can be written in matrix form as:

$$A_1^{reb} y_{v,r} \leq 1_{m \times 1} \tag{3.40}$$

where matrix A_1^{reb} has size $m \times (n \cdot m)$ and has entries equal to one when the request-vehicle couple represented by column $r \cdot v$ is matched with the vehicle represented by column v:

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccccccccc} y_{v_1,r_1} & y_{v_1,r_2} & y_{v_1,r_3} & y_{v_2,r_1} & y_{v_2,r_2} & y_{v_2,r_3} & y_{v_3,r_1} & y_{v_3,r_2} & y_{v_3,r_3} \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

The second set of inequality constraints that we need is used in order to avoid that a task is assigned to more than one vehicle. This is formalized with:

$$\sum_{r \in \mathcal{I}^V_{R=r}} y_{v,r} \leq 1 \ \forall r \in \mathcal{R}_{ko} \tag{3.41}$$

where $\mathcal{I}^V_{R=r}$ is the set of indeces that tells what vehilce v has been matched with what request r. With an example we will show how to build the constraint and how it is shaped.

Example. We consider the same problem of the previous example, with three vehicles $\mathcal{V} = \{v_1, v_2, v_3\}$ and three requests $\mathcal{R} = \{r_1, r_2, r_3\}$. The matrix formulation of 3.41 is the following:

$$A_2^{reb} y_{v,r} \leq 1_{n \times 1} \tag{3.42}$$

where matrix A_2^{reb} has size $n \times (n \cdot m)$ and has entries equal to one when the request-vehicle couples represented by column $r \cdot v$ is matched with the request represented by column r:

$$\begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \end{array} \begin{array}{c} \begin{array}{ccccccccc} y_{v_1,r_1} & y_{v_1,r_2} & y_{v_1,r_3} & y_{v_2,r_1} & y_{v_2,r_2} & y_{v_2,r_3} & y_{v_3,r_1} & y_{v_3,r_2} & y_{v_3,r_3} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

A_2^{reb} is made of m identity matrix blocks of size $n \times n$.

Next, we need to add some additional constraints since the maximum number of vehicle/requests that can be assigned depends on the number of idle vehicles and number of unassigned requests. Therefore, the maximum number of total assignments during the rebalancing phase will be the minimum between the number of idle vehicles in $\mathcal{V}_{idle}$ and the number of unassigned requests in $\mathcal{R}_{ko}$. We will formulate this as:

$$\sum_{v \in \mathcal{V}_i dle} \sum_{r \in \mathcal{R}_{ko}} y_{v,r} = min(|\mathcal{V}_{idle}|, |\mathcal{R}_{ko}|) \tag{3.43}$$

Next, we want that the variables of our problem are subject to the constraints $0 \leq y_{v,r} \leq 1 \; \forall y_{v,r} \in \mathcal{V}$, so that our problem can be formulated as a linear programming problem. The optimal assignment for the rebalancing phase is the solution to the following optimization problem:

$$\begin{aligned} \min_{x} \quad & \sum_{v \in \mathcal{V}_{idle}} \sum_{r \in \mathcal{R}_{prior}} c_{v,r} y_{v,r} & \\ \text{subject to:} \quad & \sum_{r \in \mathcal{I}^R_{V=v}} y_{v,r} \leq 1, & \forall v \in \mathcal{V}_{idle} \\ & \sum_{r \in \mathcal{I}^V_{R=r}} y_{v,r} \leq 1, & \forall r \in \mathcal{R}_{prior} \\ & \sum_{v \in \mathcal{V}_i dle} \sum_{r \in \mathcal{R}_{prior}} y_{v,r} = min(|\mathcal{V}_{idle}|, |\mathcal{R}_{prior}|) & \\ & 0 \leq y_{v,r} \leq 1 & \forall y_{v,r} \in \mathcal{V}_{idle} \end{aligned} \tag{3.44}$$

The optimization problem has been solved by CPLEX that uses a dual simplex method as default option. There are many different options and algorithms to be used in CPLEX to solve linear programming problems, but this has been proven experimentally by the IBM community to provide the best overall performances.

3.7 Priority assignment

When a customer is using the service, it may be that he is willing to pay more to have the item delivered or the vehicle available for pick-up as soon as possible. A trivial example is the possibility that Uber gives to customers to decide whether to use the UberPool option or going with the standard private vehicle service. This depends on the freedom given to the client to customize the service. For example, if this was a taxi service, it might be the case where the customer can decide whether to share its vehicle with someone else, because of privacy issues and social attitude [83], or simply because of the need to hurry. In this case, the customer is the one having direct benefits on these kind of choices. Having to wait on the street for lot of time to share the ride, or in the case of adverse weather condition, the customers might prefer to pay a higher amount to be served as soon as possible.
Instead, if the service was a food delivery service, the customer could be less interested in the fact that the food is being transported in the same vehicle of other customers. Larger waiting times could be acceptable, because maybe the order has been placed from indoor location and even in case of bad weather, a large waiting time may not have much influence. Customers may want to pay an additional fee to have the food delivered directly to them respecting the estimated arrival time of the delivery. It can also be the case where priority is given in the sense that the request asking for priority is being dropped-off first. In this case, the matchings should take into account that this task is not allowing to be dropped-off as second task.

In our case we want to give the possibility to the customer of the service to ask for priority. Since our algorithm never guarantee that all the tasks of the set are satisfied, with our priority mode we want to make the prioritized tasks to be assigned for sure when vehicles are assigned to trips during the ILP solution. Priority can be guaranteed only when there are free vehicle in $\mathcal{V}_{idle}$. If no vehicle is free at the moment of the algorithm call, it will be difficult to assign priority. However, in the case no vehicle is available to ensure a task to be picked up and served, we can try to increase the chance for that task to be assigned. One idea could be to increase the parameter of delay Δ of such tasks when finding for matchings, so that to increase the chances to be assigned. This, though, cannot be a service that is charging an additional cost to the customer. In this case, priority could be assigned to tasks that requires because of their nature. For example, in a service whose purpose is to deliver items, we can give priority to items such as food and other fresh products that otherwise would loose their value if received with a lot of delay. If we increase parameter Δ, there is more possibility for the task to be shared with other tasks, because of the increased flexibility of its time windows.
In other scenarios, where customers might pay more to be served as soon as possible, we can implement a system that checks for availability of vehicles and then, if there is any, the algorithm assign at most a number of tasks equal to the number of available vehicles. Then the application can notify the customers and ask for the confirmation of the priority option. If approved, the prioritized assignment $v - r$ will be executed. Instead, in the

case the vehicle are not enough to ensure the coverage of all the tasks asking for priority, the algorithm will assign the most convenient one (it will require the solution of another optimization problem) and the remaining tasks will be denied priority.
Actually, it is not true that paying for the priority option will result in decreased travel time. Indeed, if the priority is assigned to the task (in the case priority has been asked and later confirmed because of the presence of idle vehicles) the algorithm will make sure that the task is served, according to the need also of the other prioritized tasks. Unfortunately, it could be that with the solution to the main optimization problem, the same task (but with no priority) is assigned to more convenient vehicles. Again, we remark that in this case, the task would be assigned for sure.

We decided to implement the prioritized assignment in a similar way the rebalancing phase assignment is implemented. Therefore, for assigning vehicles to tasks that ask for priority, we will solve a linear programming problem to select the couples $v - r$ within the possible matchings that we will generate.
We can guarantee priority only to a number of tasks that is at most equal to the number of available vehicles in the fleet. In this way, we will know that some or all of the idle vehicles will have to guarantee some requests to be served.
Given a set of requests asking for priority defined as $\mathcal{R}_{prior}$ and a set of free vehicles $\mathcal{V}_{idle}$ we will generate all the possible matchings between requests and vehicles. To each of the matchings we will link a cost that is given by the travel time required by the vehicles $v_j \in \mathcal{V}_{idle}$ with $j = 1, \ldots, m_{idle}$ to completely operate the rides of tasks $r_k \in \mathcal{R}$ with $k = 1, \ldots, n_{prior}$ taken singularly. The assignments for the priority phase are only trips of size one. The cost used is similar to 3.37:

$$c_{v,r} = \sum_{i=0}^{|Loc|-1} tt(Loc_{i+1} - Loc_i) \tag{3.45}$$

The formulation of the set of constraints for the priority assignment are of the same kind of the ones used during the rebalancing phase. Indeed, the two problems are very similar. In both cases our aim is to produce a number of optimal assignments that is at most equal to the minimum between the number of requests entering this assignment phase and the number of vehicles ready to be assigned.
Therefore, from set of constraints 3.39 and 3.41 we have:

$$\begin{aligned} \sum_{r \in \mathcal{I}^R_{V=v}} p_{v,r} \leq 1, \ \forall v \in \mathcal{V}_{idle} \\ \sum_{r \in \mathcal{I}^V_{R=r}} p_{v,r} \leq 1 \ \forall r \in \mathcal{R}_{prior} \end{aligned} \tag{3.46}$$

where $p_{v,r}$ for $v \in \mathcal{V}_{idle}$ and $r \in \mathcal{R}_{prior}$ is our optimization variable. Similarly to the rebalancing problem, to guarantee that a number of requests at most equal to the available vehicles is assigned, we will define a set of constraints similar to 3.43:

$$\sum_{v \in \mathcal{V}_{idle}} \sum_{r \in \mathcal{R}_{prior}} p_{v,r} = min(|\mathcal{V}_{idle}|, |\mathcal{R}_{prior}|) \tag{3.47}$$

The linear programming problem for the priority assignment will be, analogously to the rebalancing problem:

$$
\begin{aligned}
\min_{x} \quad & \sum_{v \in \mathcal{V}_{idle}} \sum_{r \in \mathcal{R}_{prior}} c_{v,r} p_{v,r} \\
\text{subject to:} \quad & \sum_{r \in \mathcal{I}^R_{V=v}} p_{v,r} \leq 1, && \forall v \in \mathcal{V}_{idle} \\
& \sum_{r \in \mathcal{I}^V_{R=r}} p_{v,r} \leq 1, && \forall r \in \mathcal{R}_{prior} \\
& \sum_{v \in \mathcal{V}_i dle} \sum_{r \in \mathcal{R}_{prior}} p_{v,r} = min(|\mathcal{V}_{idle}|, |\mathcal{R}_{prior}|) \\
& 0 \leq p_{v,r} \leq 1 && \forall p_{v,r} \in \mathcal{V}_{idle}
\end{aligned}
\tag{3.48}
$$

Once the priority assignment has been completed, we will end up with a set of assignments $v - r$. Next, it will be the turn of the main assignment algorithm to work on the tasks and vehicles to find the the global optimal matchings of vehicles to tasks. At this point, if the main algorithm, which involves the generation of the shareability network *RV-Graph* and of the feasible trips set *RTV-Graph*, was called with no knowledge of the priority assignments, the priority assignments would be lost. Indeed, *RV-Graph* and *RTV-Graph* do not account for priority assignments by this point.
In order to consider the computed priority assignments we decided to forcibly add them to the list of edges $e(T_i, v_j)$, if not already present. We remind that those edges exist when a trips $T_i \in \mathcal{T}$ of any size can be satisfied by vehicle v_j. To every edge, a cost of the assignment is associated, and it depends on the choice of the cost representing our problem.
After the set of edges of our problem is updated with the priority assignments, we want to make sure that the tasks with priority are assigned during the solution of the mixed integer linear programming. To this aim, we need to add a set of constraints that forces the algorithm to assign those tasks. The assignment will be feasible, since the number of tasks assigned is at most equal to the number of idle vehicles and all the other constraints are satisfied.
It may happen that some or all of the $r - v$ matchings from the priority assignment already exist in the set of edges of our problem. In this case, the priority assignment is not added. Another case is when one or more edges of the *RTV-Graph* contain one or more of the requests that belong to priority matchings. In this case, the algorithm will force the fact that at least one of the edges containing tasks with priority is assigned.
Instead, in the case a vehicle v_j that belongs to a priority assignment $v_j - r_k$, is also part of one or more edges $e(T_i, v_j)$ where T_i is a trip not containing task r_k, we will need to suppress those edges, forcing the MILP problem to not assigned them. We will add these constraints to the set of equality constraints 3.19.
Given a set of priority assignments of the kind $v_j - r_k$, to force task r_k to be assigned we need that:

$$
\sum_{j \in \mathcal{I}^R_k} \epsilon_{i,j} = 1, \ \forall r_k \in \mathcal{R}_{prior,ass}
\tag{3.49}
$$

where $\mathcal{I}_k^R$ is the set of indeces j of vehicles $v_j \in \mathcal{V}_{idle}$ such that the trip T_i corresponding to edge $e(T_i, v_j)$ has task r_k in its route and vehicle v_j can serve that trips. This means that $T_i = \{\dots, r_k, \dots\}$ and that edge $e(T_i, v_j)$ exists. $R_{prior,ass}$ is the set of tasks with priority that has succeded in obtaining a priority assignment.

We then want to deny some edges to be assigned. These are edges that contains the vehicle v_j belonging to the priority assignment $v_j - r_k$, but linked to trips T_i that do not have task r_k in their route. To this aim we need a constraint of the kind:

$$\sum_{i \in \mathcal{I}_j^V} \epsilon_{i,j} = 0, \ \forall v_j \in \mathcal{V} \tag{3.50}$$

where $\mathcal{I}_j^V$ is the set of indexes i of the trips that are linked to vehicle v_j but not to the corresponding task r_k in the priority assignment. Also, it is the set of indexes i of trips such that $r_k \in T_i$ but v_j is not the assigned vehicle for that edge.
We want to show with an example how such constraints are build.

Example. We assume to have a set of requests $\mathcal{R} = \{r_1, r_2, r_3\}$ and a set of vehicles $\mathcal{V} = \{v_1, v_2\}$. From the *RTV-Graph* of this problem we have the set of trips $\mathcal{T} = \{T_1, T_2, T_3, T_{1,2}, T_{1,3}, T_{3,2}\}$ and a set of edges $\mathcal{E}$. Suppose that tasks r_1, r_2 and r_3 are asking for priority and that only vehicles v_1 and v_2 are idle. At this point only two of the three tasks could be assigned with priority. Suppose these are r_1 and r_3, so that the priority matchings are $r_1 - v_1$ and $r_3 - v_2$. Now we need to build constraints so that request r_1 is assigned with vehicle v_1 in at least one of the possible trips and that tasks r_3 is assigned to vehicle v_2 in at least one of the trips. From constraint 3.49 we have, for requests r_1 and r_3, assuming a certain set of edges between trips and vehicles:

$$\begin{aligned} &e(T_1, v_1) + e(T_{1,2}, v_1) = 1; \\ &e(T_3, v_2) + e(T_{1,3}, v_2) = 1; \end{aligned} \tag{3.51}$$

where we are guaranteeing that at least one of the edges containing matchings $r_1 - v_1$ and $r_3 - v_2$ is assigned. Trip $T_{3,2}$ does not appear because this trip cannot be served by vehicle v_2 but only by vehicle v_1.
In matrix form, we will have a matrix A_1^1 of size $|\mathcal{R}_{prior,ass}| \times l$, where entries are one if the edge in the j-th column has the entire matching (both vehicle v_j and task r_k), zero otherwise.
Instead, to avoid that vehicles other than v_1 for task r_1 and v_2 for task r_3 can serve the requests, we need to forcibly unassign the candidates containing these other vehicles and the tasks within the respective trips. The following equations are a possible implementation for our example of the constraints 3.50, that want to make sure that the vehicles and tasks of the priority assignments $r_1 - v_1$ and $r_3 - v_2$ are not split up:

$$\begin{aligned} &e(T_1, v_2) + e(T_{1,2}, v_2) + e(T_2, v_1) = 0; \\ &e(T_3, v_1) + e(T_2, v_2) + e(T_{1,3}, v_1) + e(T_{3,2}, v_1) = 0; \end{aligned} \tag{3.52}$$

where equations 3.52 are reffering respectively to the two priority assignments that we have. These constraints can also be written in compact form in one equation, forcing the sum of all the terms to be zero.
In matrix form we will have a vector A_2^p of size $1 \times l$ of which entries are one when the j-th edge does not respect the math $v_j - r_k$ in any way, either because vehicle v_j is not assigned to task r_k or because task r_k is assigned to a vehicle that is not v_j. In Algorithm 4 we have an example of the computation of the matrixes used to ensure the assignment of tasks with priority.

Algorithm 4 ILP constraints to preserve the priority assignment

```
1: if priority_list ≠ ∅ then
2:     A_1 = 0_{priority_list.size}
3:     for each ride p ∈ priority_list do
4:         for each edge e ∈ E do
5:             if (∀r_e) ∈ p then
6:                 A_1^p [p] .push(1)
7:             else
8:                 A_1^p [p] .push(0)
9:             if (∀r_e) ∈ p then
10:                A_2^p.push(1)
11:            else
12:                A_2^p.push(0)
13:    insert(A_2, A_1^p)
14:    insert(A_2, A_2^p)
15:    insert(b_2, 1_{priority_list.size×1})
16:    insert(b_2, 0)
```

3.8 Dynamic algorithm design

Measure the dynamism of a dynamic vehicle routing system is not a trivial assignment. In contrast to a static vehicle routing problem, the performance of the dynamic counterpart is assumed to be dependent not only on the number of customers and their spatial distribution, but also on the number of dynamic events and the time when these events actually take place [59].
In dynamic problems, with respect to static problems, fast computation is required. In a static setting, the dispatcher may afford the luxury of waiting for a few hours. In a dynamic setting this is not possible, because the dispatcher needs a solution to the assignment as soon as possible.
We need also an information update mechanism to check the state of vehicles and tasks. This is not needed in a static problem instead. We need therefore to know at any time the position of our vehicles and requests. In a static case the location of vehicles and their capacity over time is not a requirement, since it is important that it is known only at the beginning. Therefore a dynamic systems needs a very good communication system and a software infrastracture able to manage the information of the fleet and the information of the incoming requests. A definition of dynamic system comes from [79]:

- *A dynamic system is one that manages dynamic data;*

- *A model is dynamic if it incorporates explicitely the interaction of activities over time;*
- *We have a dynamic application if a model is solved repeatedly as new information is received.*

Dynamic data are a set of information that are constantly changing. They might include real-time customer demands, traffic conditions, or driver statuses. Dynamic applications of models place tremendous demands on access to real-time data and on the performance of algorithms. Typically, it is necessary to update information, optimize and return results in a matter of minutes, seconds or even less.
There are two main possibilities to use the algoirthm in a dynamic manner. These depend on the size of the problem. The first would be to call the algorithm as soon as a new task is added to the set of pending tasks. This solution is more suitable for small size problem, where the frequency of new arriving requests is not very high and the computation time of the algorithm is on average shorter than the time between two consecutive algorithm calls. This possibility would require the estimation of a worst case execution time, without knowledge of the frequency of tasks. Instead, the second possibility could be of calling the algorithm every fixed period of time. In this case, the algorithm will still have to be executed before the next call, but being the number of tasks smaller, its execution will be faster. Knowing the size of the problem and the number and frequency of the arriving tasks, we can estimate a worst case execution time and call the algorithm every fixed time step. All the requests arrived within a certain time period, will be processed at the next algorithm call. This solution is suitable for big size problem, where the algorithm is not fast enough to withstand the frequency of the new arriving tasks.
The next step for the implementation of the possibility for trips to be added to routes when they have already been assigned or picked-up is to include already computed assignments to the new pending tasks set. We decided that a new task can only be added to trips of size one. This because the total number of trips of size three out of the total number of trips is low, and the computational effort added to transform size two trips into size three trips is higher than its benefits. In fact the chance of having size three trips is a lot lower than having trips of size one and two. Therefore, we think that the possibility that a new trips of size three is built on top of a size two trip is very low, and so we discard this possibility.
There are two cases for trips of size one that can accept new tasks dynamically:

- The vehicle v_j has been assigned to a trip T_i and it is heading to the pick-up location of the request r_k that composes that trip.
- The vehicle v_j has already picked up the task of trip T_i and it is now heading to the destination of the task.

In the first case, the task to which the vehicle is heading to may become the second task to be picked up, only if the maximum delay constraints given by Δ are still satisfied. Therefore, if this is the case, we will only need to consider that the vehicle is not in its original position but in a different location. The conditions to verify if the share route is feasible or not is the same of $travel\_size\_two()$.
Instead, in the case the task of trip T_i has been already picked-up and the vehicle is on his way to the destination, we have to plan a change in the route, where we need to be able to still satisfy the delay constraint on the arrival time of the original task of trip T_i.

Example. Assume to have a trip T_i of size one, composed of request r_k. A vehicle v_j has been assigned to the trip and the task has already been picked up at o_k location. Assume also that there is a new pending request r_d that is seeking for a vehicle to be assigned. We will include task r_k in the assignment problem as it was a new task, but its status will specify that it has already been picked-up by vehicle v_j. At this point we procede with the generation of the *RV-Graph* as it was the standard case of two new pending tasks. In fact, the assumption made when looking for the possibility to share two requests, is that a vehicle starting from the origin of the first of the two tasks, is able to satisfy constraints 3.2 or 3.3. At this point, when a vehicle is on its way to reach the destination d_k of the already picked-up task r_k and the vehicle is deviated from its way to be able to pick-up also task r_d at origin o_d then, if the two tasks had not the possibility to be matched with a *shortest path* through pivot points $o_k \rightarrow o_d \rightarrow d_k \rightarrow d_d$ (or $o_k \rightarrow o_d \rightarrow d_d \rightarrow d_k$), then they would have not the possibility to be matched with a dynamic assignment. We can see an example of these situation in Figure 3.5, where we clearly see that $path_1 + deviation < path_3$. If the *red* path do not exist in the RTV-Graph, then the two tasks cannot be shared with the rerouting of the vehicle.

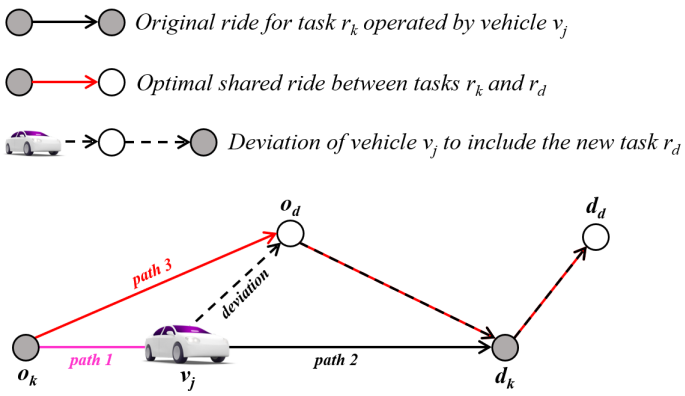


Figure 3.5: Example of dynamic deviation of the vehicle to pick-up a new task.

Instead, when generating the *RTV-Graph*, at the moment of checking for the feasibility of a candidate trip of size two using function $travel\_size\_two()$, we will need to take into account that part of the candidate trip ($path_3$ in Figure 3.5) has already been travelled by the vehicle. Function $travel\_size\_two()$ will have to be modified in order to verify whether task r_k is already picked-up, and if this is the case, then it needs to implement new conditions to verify the feasibility of the candidate trip. The new conditions to be satisfied when task r_k is already on board of the vehicle are:

- If the two tasks can be combined by route $o_d \rightarrow d_k \rightarrow d_d$:
  1. $T_{clock} + tt\left(v_{j,xy}, o_{r_d}\right) \leqslant st_{r_d} + \Delta$;
  2. $T_{clock} + tt\left(v_{j,xy}, o_{r_d}\right) + tt\left(o_{r_d}, d_{r_k}\right) \leqslant at_{r_k} + \Delta$;
  3. $T_{clock} + tt\left(v_{j,xy}, o_{r_d}\right) + tt\left(o_{r_d}, d_{r_k}\right) + tt\left(d_{r_k}, d_{r_d}\right) \leqslant at_{r_d} + \Delta$;
  4. $r_{k.pass} + r_{d.pass} \leqslant c_{v_j}$;
- If the two tasks can be combined by route $o_d \rightarrow d_d \rightarrow d_k$:
  1. $T_{clock} + tt\left(v_{j,xy}, o_{r_d}\right) \leqslant st_{r_d} + \Delta$;

2. $T_{clock} + tt\left(v_{j,xy}, o_{r_d}\right) + tt\left(o_{r_d}, d_{r_d}\right) \leqslant at_{r_d} + \Delta$;
3. $T_{clock} + tt\left(v_{j,xy}, o_{r_d}\right) + tt\left(o_{r_d}, d_{r_d}\right) + tt\left(d_{r_d}, d_{r_k}\right) \leqslant at_{r_k} + \Delta$;
4. $r_{k.pass} + r_{d.pass} \leqslant c_{v_j}$;

If these conditions are verified, then a trip of size two $T_i = \{r_k, r_d\}$ is generated and added to the set of trips $\mathcal{T}$. Therefore, edges $e(r_k, T_i)$, $e(r_d, T_i)$ and $e(T_i, v_j)$ are added to the *RTV-Graph*, and will be variables of the optimization problem 3.12.
In addition, we need to make sure that the already picked-up task r_k will still be assigned to vehicle v_j even if no other task is added dynamically. To this aim, we need similar constraints to 3.49 and 3.50, so that to force task r_k to vehicle v_j to be matched and avoid that task r_k is assigned to other vehicles (or not assigned) and that vehicle v_j is assigned to other tasks.
If we were not making sure of this, it would be possible that the pre-existent assignment $r_k - v_j$ is violated and this would lead to instability in the assignment and failure of the system, even if the mathematical problem was feasible.
We call $\mathcal{R}_{assigned}$ the set of the requests that are assigned to trips of size one, and $\mathcal{E}_{dyn}$ the set of $r - v$ matchings that can be dynamically modified. These are all the assignments of trips of size one. The constraints we need to ensure a safe assignment of vehicles to trips are the sets of equality constraints:

$$\begin{aligned} \sum_{j \in \mathcal{I}_k^R} \epsilon_{i,j} = 1, \ \forall r_k \in \mathcal{R}_{assigned} \\ \sum_{i \in \mathcal{I}_j^{\mathcal{E}_{dyn}}} \epsilon_{i,j} = 0, \ \forall (r_k - v_j) \in \mathcal{E}_{dyn} \end{aligned} \tag{3.53}$$

where $\mathcal{I}_k^R$ is the set of indeces j of the set of edges $\epsilon_{i,j}$ that has the couple $r_k - v_j$ in the clique represented by the edge, and $\mathcal{I}_j^{\mathcal{E}_{dyn}}$ is the set of indeces i for which the *dynamic candidate* assignment $r_k - v_j$ is corrupted.
Algorithm 5 is used to update the constraints in order to implement the dynamic assignment in the optimal assignment phase using the ILP.

Algorithm 5 ILP constraints for the dynamic assignment of tasks

```
if rides_list ≠ ∅ then
    A_ok = 0_{rides_list.size},  A_ko = 0
    for each ride a ∈ rides_list do
        for each edge e ∈ E do
            if (∀r_e, v_e) ∈ a then
                A_ok [a] .push(1)
            else
                A_ok [a] .push(0)
            if (v_e ∈ a and ∀r_e ∉ a) or (v_e ∉ a and r_e ∈ a) then
                A_ko.push(1)
            else
                A_ko.push(0)
    insert(A_2, A_ok)
    insert(A_2, A_ko)
    insert(b_2, 1_{rides_list.size×1})
    insert(b_2, 0)
```

where *rides_list* is the vector containing the size one assignment that are already assigned or picked up, A_2 and b_2 are the matrix and vector of known terms for the equality constraints of the ILP for the optimal assignment. Also function *travel_size_two*() and part of the *RTV-Graph* generation has to be adapted to the dynamic arrival of tasks. However, the pseudocode represents the main concept behind adding tasks to already existent trips.

3.8.1 Future improvements for dynamic assignment

Previously, we stated that our algorithm is not able to plan future assignments. This is true if we consider a schedule that tells each vehicle, what would be the entire tasks queue for a certain time period in the future. However, there is a possibility to have a one-task ahead scheduled assignment using our assignment algorithm.
When considering trips of size two, we consider only tasks combination that lead to a the tasks interleaving, so there is an overlap between the two. This means that there will be a time, even if small, where the vehicle will have both tasks on board. These situations are represented by cases *c*)-*g*) of Figure 1.1.
Instead, try to imagine a task combination where the first task can end before, or at the same time when the second task starts. These are represented by cases *a*), *b*) and *h*) of Figure 1.1. To be able to match the two requests, though, we need to add these possible additional combinations to the set that has been considered by now. The additional possible combinations are:

- $o_i \rightarrow d_i \rightarrow o_j \rightarrow d_j$
- $o_i \rightarrow d_i \equiv o_i \rightarrow d_j$

From which we derive a set of time windows conditions to be respected when looking for shareability, similar to conditions 3.2-3.5:

$$\begin{aligned} st_i \leqslant pt_i \leqslant st_i + \Delta_1 \quad &\text{(Cond 1)} \\ at_i \leqslant pt_i + tt\,(o_i, d_i) \leqslant at_i + \Delta_1 \quad &\text{(Cond 2)} \\ pt_i + tt\,(o_i, d_i) + tt\,(d_i, o_j) \leqslant st_j + \Delta_2 \quad &\text{(Cond 3)} \\ pt_i + tt\,(o_i, d_i) + tt\,(d_i, o_j) + tt\,(o_j, d_j) \leqslant at_j + \Delta_2 \quad &\text{(Cond 4)} \end{aligned} \tag{3.54}$$

where Δ_1 and Δ_2 are generally different, since the second customer, that is sending request r_j will have to accept to wait a certain amount of time so that to let the other task to finish. Δ_1 and Δ_2 can be dynamically changed by the algorithm so that to increase/decrease the chance of matching the requests.
These combinations are a formalization of how most of the Uber and Lyft routes are planned dynamically when requesting for a ride. At the time of the confirmation of the request form the user, the application tries to match the new request to nearby vehicles. The ride must be accepted by a driver via the app. If the driver denies the matching, a new assignment is proposed to another vehicle. If the driver has a ride on board of the vehicle, he can still accept the new ride and this will become a dynamic matching. The matching will require the first rider to be dropped off at a certain location and the new customer to be picked up at her/his location. There will be the need for the second ride to have a large pick-up time window so that to accomodate the end of the other task within the maximum delay time allowed. Therefore, in this case, the system could ask the second customer to wait a longer time than usual at its pick-up location. During this waiting time, the customer can move from her/his original pick-up location

and meet the vehicle in another location that is more convenient for both the driver and the customer, so that to cut the waiting time simply by using part of it by moving to a different location, if accessible. The change in the meeting points is also an aspect that interest the matching of rides, since the possibility of matching more rides increases as we increase the possible pick-up locations for the two tasks. This problem is addressed in [90].

Since the number of possible matchings increases drastically when we allow also for the new tasks combination, we have to consider that the algorithm will take longer to compute the $RV - Graph$ and in particular the $RTV - Graph$. This suggests us to try to decrease the the number of tasks to be matched, so that to lower the number of possible combinations to analyze. One possibility is to implement a regional assignment, so that to exclude the possibility to combine tasks that are too distant form each other. Dividing the set of requests into regions with hard clustering techniques will deny requests that are close to the borders to be matched with requests close to the same border but belonging to the neighbor regions. An approach could be therefore to try to match requests from adjacent clusters or to use overlapping clusters. Unfortunately, overlapping clusters have a very high computational complexity thus making them unrealistic for real-life problems [8].

3.9 Randomization of the algorithm for better performance

Since most of the code is implemented by using *for cycles* in order to scan the entire set of possible requests matchings and the possible couplings between vehicles and the feasible trips, when the number of requests and vehicles increase the algorithm takes longer to compute the optimal assignment. Therefore, we want to make our algorithm robust against the size of the problem, so that it is considerable scalable to real-life problem instances.
There are two parameters that we introduce to solve two very computationally demanding steps of our algorithm. These steps are:

1. *Vehicles-requests search in the RV-Graph*: When building a trip of size *one* for a vehicle. For each vehicle we build size *one* trips from requests that are linked to the vehicle. This information is stored in the *RV-Graph adjacency* matrix. Though, this operation could be time consuming in the case the *RV-Graph* has vehicles linked to many tasks (because maybe of high demand density in its surroundings). We introduce a limit n_1 on the number requests that will be analyzed for that vehicle. Therefore, using function `std::sample` we pick randomly n_1 requests for each vehicle from each vehicle tasks list, as we can see at lines 3 and 4 of Algorithm 1.

2. *Trips of size two search*: When building a trip of size *three*, since we try to merge size *one* trips with size *two* trips for each vehicle, we need to scan the size two trips linked to that vehicle in the *RTV-Graph adjacency* matrix. In a similar way to the previous case, the *adjacency* matrix is representing lists of trips that are linked to the vehicles. If the number of tasks is high, the number of trips of size *two* linked to one vehicle might be high as well. Hence, we introduce a parameter n_2 that will randomly pick size *two* trips from the lists using again function `std::sample`, as shown at lines 6, 7 of Algorithm 3. This way the number of possible combinations to be checked decreases and so it does also the computation time.

Unfortunately, by randomly picking just some of the possible combinations, we will loose

some degree of optimality with respect to the complete analysis of all the possible combinations. However, we will show how the loss in optimality will be highly compensated by a steep decrease of the computation time. Also, we will see what parameter will affect more the computation time and the total cost of the assignment.

Chapter 4

Problem instances and algorithm performance

The algorithm has been first designed in Matlab because of the simplicity in debugging the algorithm and the simple interface to access data structures, especially when working with vectors and matrices. Moreover, Matlab environment provides a good plotting tool that was useful in proving the solutions with graphical reuslts. In Matlab, to solve the integer linear programming and linear programming problems, we used IBM ILOG CPLEX Optimization Studio to model and solve the optimal assignment problem. The Matlab interface for CPLEX Studio resumes the classic interface of the built-in functions of the Optimization Toolbox of Matlab. Indeed, once the cplex library path is set on Matlab, then it is possible to simply use functions `cplexbilp()` when solving the integer linear programming (this actually is the function to solve the binary integer linear programming) and function `cplexlp()` when solving the linear programming problem for the rebalancing phase.
After having designed the algorithm and all its functionalities in the Matlab environment, we decided to implement it in C++ programming language, in order to ease the development process of a future application where vehicles are actually addressed to tasks in a real environment. Also, C++ gives us the possibility to build an optimized application where we can measure the performance of the algorithm in different working scenarios. The C++ application has been developed in VisualStudio IDE (integrated Development Environment) and for the modeling and solution of the optimization problems, we still relied on CPLEX Optimization Studio. In this case, we need to make use of the CPLEX Concert Technology libraries. This time, the modeling of the optimization problems required to dig into the library objects and methods in order to be able to build a working and efficient optimization application. After the implementation of the algorithm in C++, we started testing the algorithm to try do discover its potentials and its limits. Since our greatest concern is about the execution time, because of the possibility then to use the algorithm in a real setting, we introduced some heuristic (these are parameters n_1 and n_2, that we already explained in the previous chapter) in order to solve the assignment in an acceptable time. Otherwise, due to the complexity of the algorithm and the great number of variables and related constraints, it would be difficult if not impossible (the development PC ran out of memory in certain big size instances of the problem) to solve the assignment in tolerable execution times. The aformentioned randomization of the algorithm made it possible to have the software to work in stressful conditions and produce good results for the optimization. Following, we have some of the instances we used to compare the different performances and to study the effects of the randomness introduced

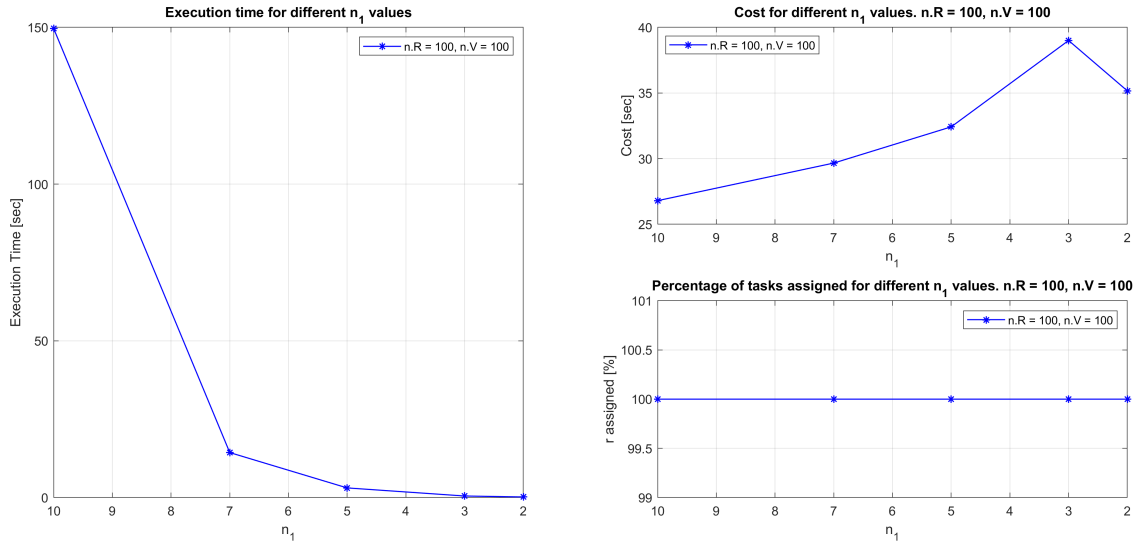
into some of the steps of the algorithm.

4.1 Tests and algorithm evaluation

For each test we have carried out *five* simulation and computed the average value of the outcomes. We specifically are interest in: *execution time*, *cost of ILP assignment*, *number of tasks served by the ILP assignment*, *relative number of trips of different sizes*, *number of idle vehicles after the assignment*. In the following pictures we do not show all the metrics of evaluation but we will highlight the most important ones.

4.1.1 Effect of the randomization parameters on the algorithm performance

The first test we are interested in is about the effect of the first randomization parameter n_1 that will decrease the number of trips of size *one* per vehicle. In Figure 4.1 we see the effects on the execution time, total cost of the assignment and number of tasks assigned, when we decrease the value of n_1. The number of tasks and vehicles is big, and the execution time is exponentially affected by a decrease on parameter n_1. For $n_1 > 10$ the execution time is so big that we cannot even end the simulation. We computed an instance for $n_1, n_2 = 100$ and we got a computation time of about 12715 *seconds* while the optimal cost was about 50% less than the case with $n_1 = 2$ and $n_2 = 100$. Indeed, if we look at the cost, we can see that the trend is not as steep as the one of the execution time. Even more impressive is that the number of assigned tasks is always constant within all the istances. Therefore, these results can suggest that tuning the randomization, we can strongly improve the computational performance of our algorithm without infering too much on the cost of the assignment and quality of the service. It will follow a series of tests where we will focus on the effect of the randomization parameters for different instances with different number of tasks and vehicles.
We want to say that we also tested the effect of parameters n_2 only. Its effects, though, are not as strong as the ones of n_1. Hence, leaving n_1 with a high value and tuning n_2 do not bring to satisfying results. This is why, from the next tests, we will tune n_1 and n_2 together, so that to consider their effects together.



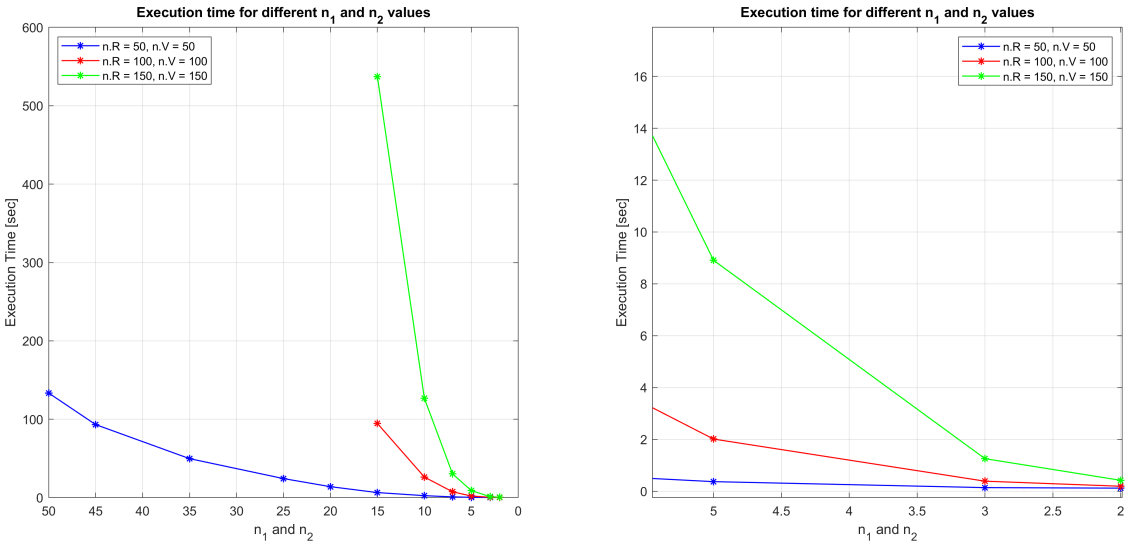
(a) (b)

Figure 4.1: *a*) Execution time of the algorithm for varying n_1 parameters. For this instance we have 100 tasks and 100 vehicles. *b*) The first plot is the cost of the assignment for decreasing values of parameter n_1. The second plot is the number of tasks assigned by the ILP optimization stage.

Changes in the sizes of the vehicles and requests sets

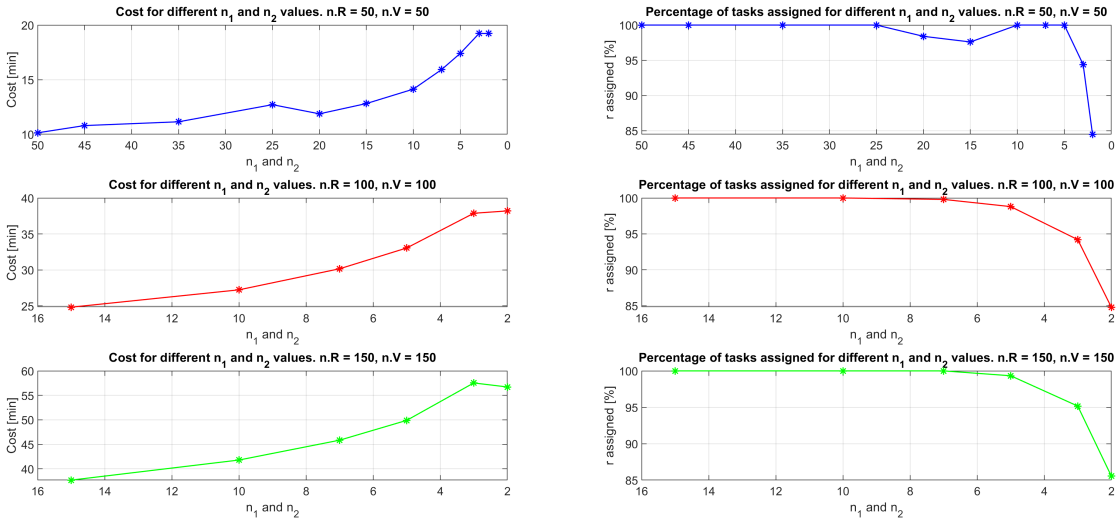
With this set of tests we want to see the effects of the randomization parameters (n_1 and n_2) when the size of the problem changes. In this case we are changing the size of both the set of tasks and the set of vehicles. As we can see in Figure 4.2, the execution time highly depends on the size of the problem when there is no randomization (we could not even compute the optimal results for the instances when $dim(\mathcal{R})$ and $dim(\mathcal{V})$ were close to 100 or 150. Indeed, for instances with 150 tasks and 150 vehicles the development PC ran out of memory). Moreover, as we have seen in Figure 4.1, the relation between the execution time and n_1 and n_2 is exponential. Although we could think that bigger size problems could make the problem unfeasible, we verified that when n_1 and n_2 are very small, the difference between problems of different sizes is attenuated. Indeed, for $n_1, n_2 < 5$ the execution time is under 10 *seconds*, that still is an acceptable timing. This means that for small randomization parameters, the execution time does not grow a lot with the problem size. Moreover, we see from Figure 4.3 that low n_1 and n_2 values are not making the cost of the assignment too high and that the percentage of assigned tasks is still large (very close to 100%). The cost is increasing in the same proportion for all the problem sizes and this happens also for the number of tasks assigned.
The choice of whether to have some low values for n_1 and n_2 or high values depends on the purpose of the application. First, we need to know what is the maximum allowed execution time and if not fixed, we need to trade off with the other metrics of evaluation. We may prefer to have very low execution time and accept that 95% of the tasks are satisfied, at higher cost, or we might prefer to satisfy 100% of tasks at lower cost and with longer execution time. These factors all depend on the design of the final application due to its purposes. Usually the purpose of dynamic ride-sharing services is to satisfy the most nnumber of customers as possible, while having them to wait the less amount of time. If we can satisfy almost 100% of the tasks while satisfying all the time windows of all the

tasks, then we should be happy with the algorithm's outcome. This is the reason why very low randomization parameters could be a great choice, when the need is to provide a very good service to the customers. The algorithm will have the possibility to be run with high frequency by assigning vehicles to tasks almost instantly.



(a) (b)

Figure 4.2: *a*) Execution time for decreasing values of parameters n_1 and n_2, for three different instances with different number of tasks and vehicles. For the three different instances, we have that both the number of tasks and the number of vehicles are increased. The instances represented are: 50 tasks and 50 vehicles (blue), 100 tasks and 100 vehicles (red), 150 tasks and 150 vehicles (green). *b*) Close up of image (*a*).

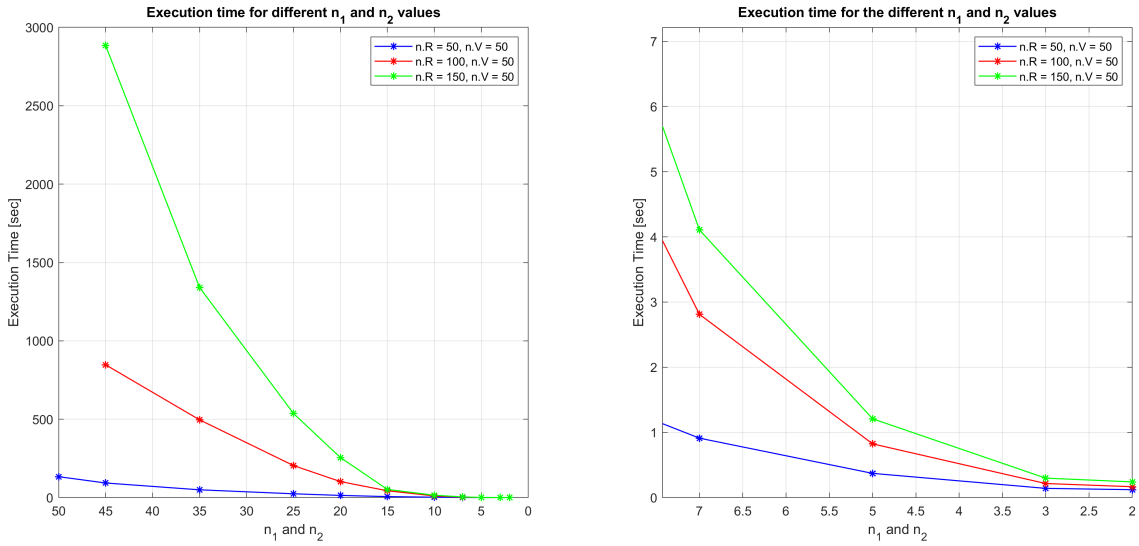


(a) (b)

Figure 4.3: *a*) Cost of the assignment in minutes after the ILP problem is solved for decreasing values of n_1 and n_2. For the three different instances, we have that both the number of tasks and the number of vehicles are increased. The instances represented are: 50 tasks and 50 vehicles (blue), 100 tasks and 100 vehicles (red), 150 tasks and 150 vehicles (green). *b*) Percentage of the number of tasks assigned, out of the total number of tasks, after the ILP problem solution for decreasing values of n_1 and n_2. Three instances are represented: 50 tasks and 50 vehicles (blue), 100 tasks and 100 vehicles (red), 150 tasks and 150 vehicles (green).

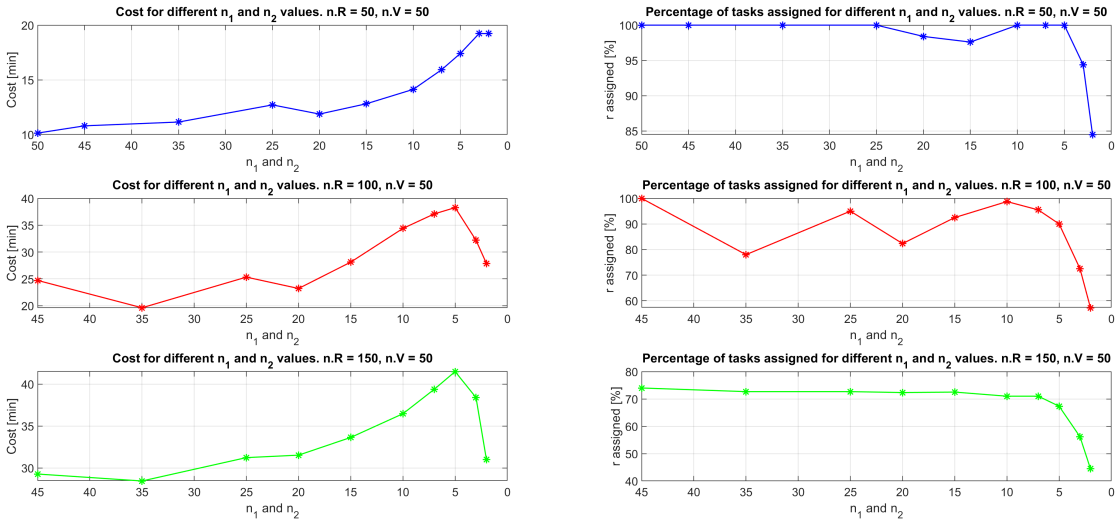
Changes in the size of the requests set only

As we can see in Figure 4.4 the execution time now depends less on the size of the problem. Indeed, a variation on the tasks set only is giving the algorithm the possibility of having higher parameters n_1 and n_2. This time, when setting $n_1, n_2 = 7$ we still have execution time $t_{exe} < 5$ for all our instances. We have less increase in the cost required to have small running time. In fact, the cost is falling back to almost optimal values when the randomization parameters have small values. This therefore comes to the cost of having less assigned requests, because the number of size one trips is increasing (because we have less possible combinations to assign). Indeed, especially for big tasks sets, the percentage of assigned requests is decreasing with respect to the previous cases. This is mostly due to the fact that the vehicles and tasks sets are not even, and therefore there is a problem of fleet unbalance with respect to the demand of customers. Again, the choice of the randomization parameters depends on the purpose and use of the algorithm.



(a) (b)

Figure 4.4: *a*) Execution time for decreasing values of parameters n_1 and n_2, for three different instances with different number of tasks and vehicles. For the three different instances, we have that only the number of tasks is increased, while the number of vehicles is fixed to 50. The instances represented are: 50 tasks and 50 vehicles (blue), 100 tasks and 50 vehicles (red), 150 tasks and 50 vehicles (green). *b*) Close up of image (*a*).

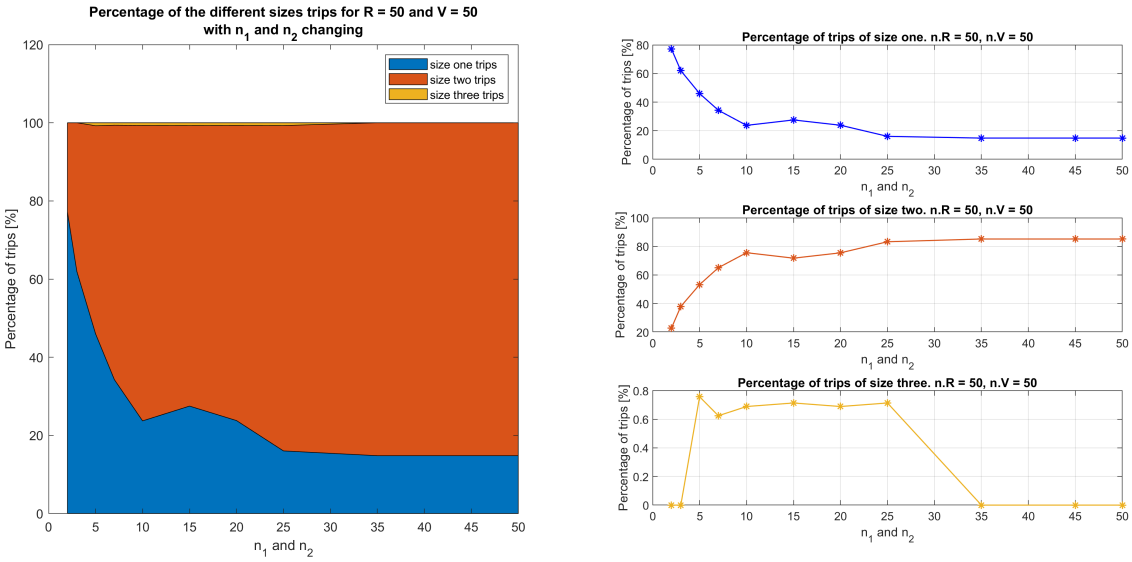


(a) (b)

Figure 4.5: *a*) Cost of the assignment in minutes after the ILP problem is solved for decreasing values of n_1 and n_2. For the three different instances, we have that only the number of tasks is increased, while the number of vehicles is fixed to 50. The instances represented are: 50 tasks and 50 vehicles (blue), 100 tasks and 50 vehicles (red), 150 tasks and 50 vehicles (green). *b*) Percentage of the number of tasks assigned, out of the total number of tasks, after the ILP problem solution for decreasing values of n_1 and n_2. Three instances are represented: 50 tasks and 50 vehicles (blue), 100 tasks and 50 vehicles (red), 150 tasks and 50 vehicles (green).

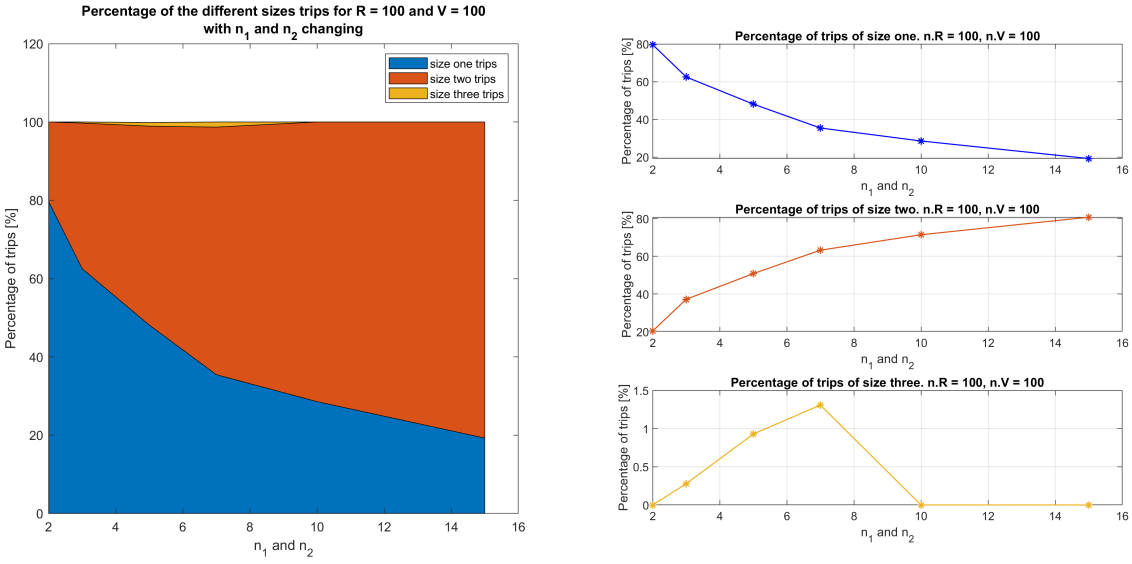
4.1.2 Effect of the randomization parameters on the shareability of rides

Although the main objective of the algorithm is to assign as many tasks as possible, we also want to analyze what kind of trips the algorithm preferred to assign when seeking for an optimal solution. We will see what is the amount of shared rides when the randomization parameters n_1 and n_2 are tuned, and what is their effect for different size problems.
First, we compare Figures 4.6, 4.7 and 4.8 where the size of the vehicles and tesks set are both growing. We can see that as the randomization paramater increases, the number of trips of size *one* decreases and consequently the number of trips of size *two* increases. This is probably due to the fact that as n_1 and n_2 increases, tthe more matchings are found and consequently the set of the solutions will grow, therefore allowing for a more optimal solution. This means that, as we have seen before in Figure 4.5 *a*) and Figure 4.3 *a*), the cost will decrease for increasing n_1 and n_2. At the same time, as we just noticed, the number of shared trips will increase. This is a proof that shareability is able to manage a set of tasks by decreasing the total travel costs. This is verified for all the three instances with different numbers of vehicles and tasks. In the case both tasks and vehicles numbers grow, the number of trips of size *three* remains low even when global shareability increases. This is maybe due to the fact that the number of vehicles with respect to the tasks is higher and the fleet is well distributed in the 2D-map.
Instead, when we compare Figures 4.6, 4.9 and 4.10, we see that shareability grows even faster than the previous case, especially in the case of 150 tasks and 50 vehicles. In Figure 4.10 we can notice the speed with which size *one* trips are no longer assigned as soon as the randomization parameters increase. When a trade-off between number of assigned requests and fleet capabilities is needed, size *one* requests are becoming inconvenient. Now the number of vehicles is low with respect to the number of tasks for two of three instances, and this is probably why vehicles need to be assigned in a more complex way in order to satisfy more requests. Also, in this case the number of size *three* trips is more consistent than the prevous case. In the case of Figure 4.9 it reaches a peak to then go down back to zero. Instead, in the case represented in Figure 4.10 the number of trips of size *three* constitutes a consistent portion of the total number of trips, being around 20% when n_1 and n_2 are big. We could not push further the analysis because of the computational requirements. In fact, for instances with 150 requests and 50 vehicles, the running time for $n_1, n_2 = 45$ is about $t_{exe} = 2880$ *sec* with exponential growth for increasing randomization parameters.



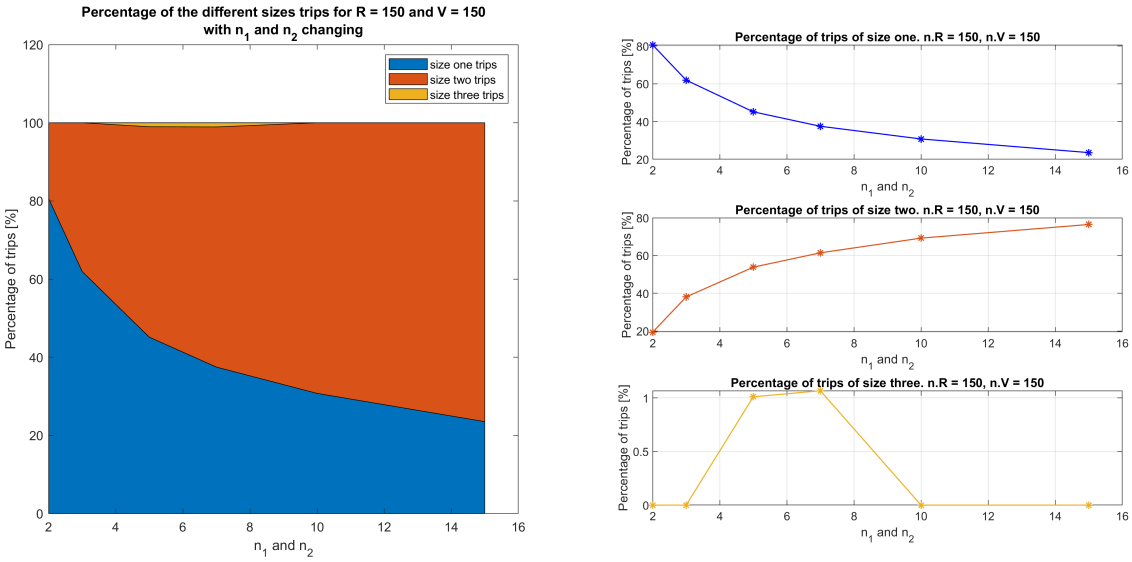
(a) (b)

Figure 4.6: *a*). Area plot of the percentage of the different size trips with respect to the total number of assigned trips, for increasing values of n_1 and n_2. *b*) Again, plots for the representation of the percentage of different size trips. In both plots we have a single instance with 50 tasks and 50 vehicles.



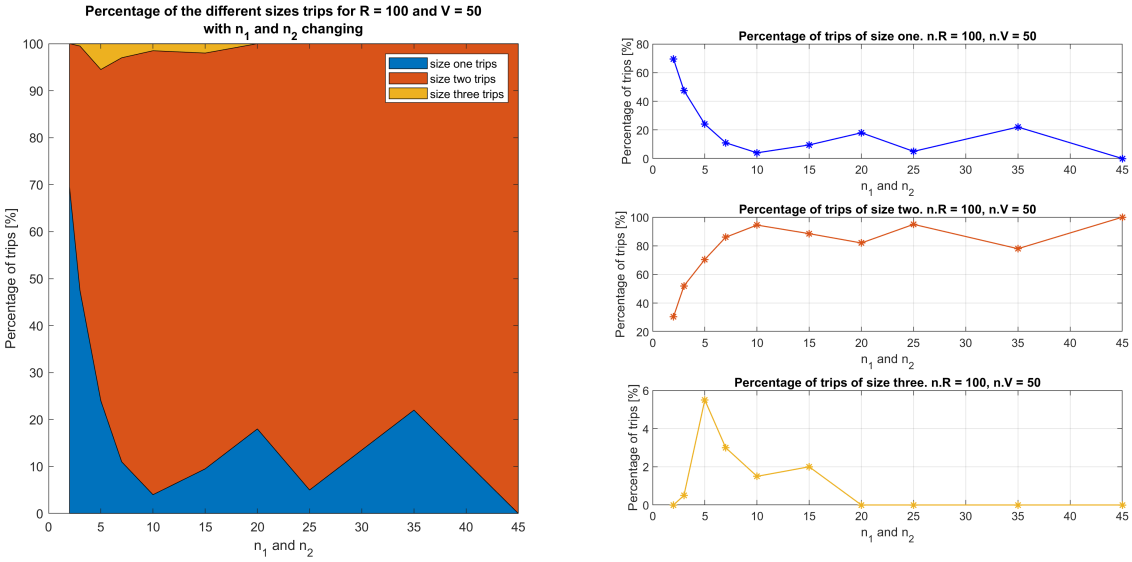
(a) (b)

Figure 4.7: *a*). Area plot of the percentage of the different size trips with respect to the total number of assigned trips, for increasing values of n_1 and n_2. *b*) Again, plots for the representation of the percentage of different size trips. In both plots we have a single instance with 100 tasks and 100 vehicles.



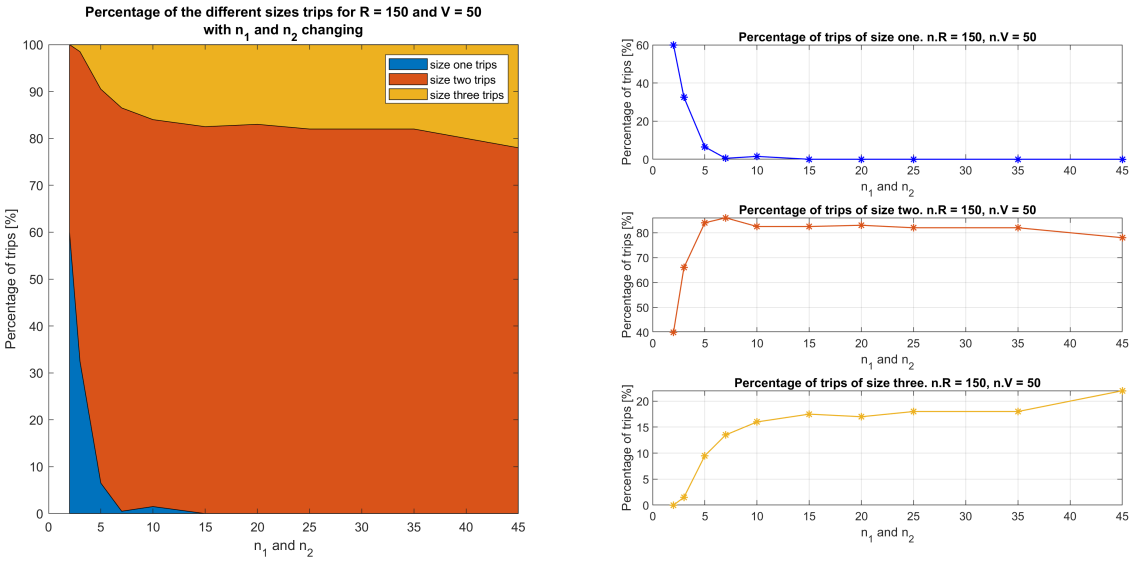
(a) (b)

Figure 4.8: *a*). Area plot of the percentage of the different size trips with respect to the total number of assigned trips, for increasing values of n_1 and n_2. *b*) Again, plots for the representation of the percentage of different size trips. In both plots we have a single instance with 150 tasks and 150 vehicles.



(a) (b)

Figure 4.9: *a*). Area plot of the percentage of the different size trips with respect to the total number of assigned trips, for increasing values of n_1 and n_2. *b*) Again, plots for the representation of the percentage of different size trips. In both plots we have a single instance with 100 tasks and 50 vehicles.



(a) (b)

Figure 4.10: *a*). Area plot of the percentage of the different size trips with respect to the total number of assigned trips, for increasing values of n_1 and n_2. *b*) Again, plots for the representation of the percentage of different size trips. In both plots we have a single instance with 150 tasks and 50 vehicles.

Chapter 5

Conclusions

The first part of the thesis studies and implements a dynamic ride-sharing vehicle assignment algorithm, where the vehicles of a fleet are assigned to trips. The trips can be constituted of up to three tasks. This means that a vehicle can have more than one request on board at the same time while satisfying the time constraints of the single tasks on board. The tasks can be dynamically assigned to vehicles, with constraints allowing to reassign vehicles to trips by adding rides to the trips on board the vehicles. To avoid that customers can see their rides reassigned to new vehicles, we do not allow for matchings to be cancelled, but only that the corresponding trips are increased in size, with the addition of new tasks on the way. The matchings of the tasks and vehicles, that together will constitute the set of feasible trips (the feasible set of our optimal assignment) are computed exploiting the properties of network theory. First, a shareability graph (*RV-Graph*) will compute the possibility of matching together two tasks and the chance of having vehicles serving single rides. Then, another graph (*RTV-Graph*) is built on top of the shareability graph, by investigating what are its complete subgraphs, to later call functions that will analyze these subgraphs and eventually tell if the trip is feasible or not, given all the tasks and vehicles information and constraints. The *RTV-Graph* will constitute the feasible set of the integer linear programming (ILP) for the assignment of the trips. The optimal assignment from the set of feasible trips is computed with the use of the IBM ILOG CPLEX Concert Technology libraries, which allow for fast computation of an optimal solution. Also, we implemented a priority assignment, which only is available when there are idle vehicles in the fleet. Tasks with priority are assigned to free vehicles via the solution of an LP problem, that simply minimizes the total cost of the priority assignment. Then, these assignments will enter again the main problem to explore for possible trips of size greater than one that can still meet the requirements of the priority tasks. After the solution of the optimal assignment, unassigned requests are assigned to idle vehicles through the solution of an LP problem. This phase is called rebalancing phase, since we try to balance the unbalances between the fleet and the taxi demand.
The algorithm is randomizing the search for trips of size one and trips of size three in order to reduce the execution time of the matching phase, that otherwise would have to go through every possible combination in the set, given the high shareability of rides when the area is dense in the number of tasks. Many tests have been carried out to prove that randomization can bring to very good results by asking just for little computation time, even in the case of large size problems. While the travel cost for a randomized assignment can increase of up to 30-50%, the computation time is decreased of almost 99.9% and the number of assigned tasks in some cases do not change. This might motivate the choice of

considering the use of strong randomization.
We later had a look at how the shareability is affected from the randomization parameters. We have verified how randomizing the trips search is decreasing the possibility of finding matching between tasks and therefore the possibility of finding convenient shared rides. Therefore, we have low-medium shareability (60-80% of trips are of size one) for strong randomness while we could find that shareability has a steep increase (80-90% of total of trips are of size two) when the random search for trips is slightly decreased.
By now the algorithm considers trips to be of size greater than one only if the tasks are together on board the vehicle for at least some time during their ride. In the future, we want to implement the possibility of having trips of size two also when the first passenger is dropped off before the second ride is picked up. Unfortunately, this new task combination increases the number of possible matchings to be analyzed and therefore also the execution time.
Another important further step would be the implementation of a more complex function for the computation of the travel times between the locations of the vehicles and the tasks on the map. By now, we have used a trivial function that assumes constant velocity of the vehicle and the L_1 distance between points in the map. Possibly, a network of points could be implemented and locations could be assigned to the nodes of the network. Then, a path planning algorithm like the Dijkstra's algorithm can be used to compute the shortest route travel time between nodes. The thesis has focused on the high-level implementation of the task assignment without concerning the low-level implementation of the road network. Also, another way to improve the algorithm would be to make the generation of the RV and RTV graphs more efficient, in order to decrease the computation time required. In fact, they are the most time demanding steps of the algorithm. Specifically, it is needed a faster way to gather information about pre-existing trips from the *RTV-Graph*.
The priority assignment can be revisited, in order to implement queues of tasks with priorities. This would allow to have more levels of priorities, depending on multiple criteria, like the order the tasks have entered the problem.

Part II

Taxi Demand Forecasting

Chapter 6

Introduction

It is a fact that in cities, taxi service is imbalanced. In some areas passengers wait too long for a ride, while in other areas many taxis roam without passengers. As we have seen in the first part, one of the problems of any transportation system is the presence of vacant vehicles, which have not been assigned because they are not able to accomodate a certain ride either because of its capabilities or because it is too far from the points of interest in the map.
Understanding the taxi demand in the future would give an opportunity to organize the taxi fleet better. It also reduces the waiting time of passengers and cruising time of taxi drivers. The information about taxi demand in the future can be used to guide both inexperienced and experienced drivers to withstand the taxi demand in the city in a smaller time, without having passengers to wait for long times. Demand forecasting is thought to help balancing the demand for service with the distribution of the fleet in space and time.
In this way, taxi drivers can drive to high demand areas, and taxi companies may reallocate their vehicles in advance to meet passenger demand. If ride quantities can be estimated in the proper way, it can give taxi companies (e.g. Uber, Lyft) a larger picture of how to position their vehicles and is thus a potentially stronger tool than the predictive capability of an individual driver. We can also think of introducing the forecast in the assignment problem, so that to ease the allocation of rides in order to have the drop-off locations close to the next points of interest. In this way, taxis will be slowly heading to different regions accordingly to the forecasted demand distribution.
The digital transformation of the tansportation systems and the emergence of sharing economy have given rise to vast amounts of spatiotemporal data. Hence, spatiotemporal traffic network analysis has become a crucial subject for traffic managers, route planning, traffic policy adjustments and transportation network planning [100].

6.1 Recent technological improvements

The past decade has seen a phenomenal growth of Internet and social network services, an explosion of sensor-equipped mobile devices. Indeed, within the improvements that are allowing a fast and radical change of the transportation systems, sensors are playing one of the most important roles. It is called Urbanet a new type of spontaneous urban network composed of heterogeneous and potentially mobile sensors. Sensors create a rich, open sensing environment where people can share resources to give mobile users a real-time acces to sensed data [81]. The large scale of the shared data is making new-generation

sensors different from what where first-generation sensors. The sensors installed in each vehicle are providing new opportunities to automatically discover knowledge, which in return deliver information for real-time decision making.
In particular, the recent advances GPS (Global Positioning System), GSM (Global System for Mobile Communication) and WiFi have provided a new way to communicate with running vehicles whilst collecting relevant information about their status and location [68].
These sensors are nowadays available in almost every vehicle and every personal mobile device. Even when the vehicle is not equipped with one of these instrumentations, it is likely that the driver is using some mobile application running on a personal mobile device, and therefore it is still providing data to a network about its status.
Cell phones and portable computers leave digital footprints of their user's activities, which are a reflection of the economical and societal interactions of a community. Researchers have sought to extract communal behaviors and intelligence from this data to obtain a better understanding of the hidden dynamics of individuals and communities. Social and community intelligence (SCI) is an emerging research field that leverages the capacity to collect and analyze these footprints to reveal human behavior, patterns and community dynamics. Wearable sensors, in contrast to static sensing infrastractures that are constrained to a fixed location, transform people and vehicles into mobile sensors for both personal and environment monitoring [98].
We call *social dynamics* the field studying the collective behavior of a city's population, as observed by their movement in the city. Researchers are interested in where people are heading, what are the points of interes (or also called *hottest* spots), what are the connections between different areas of a city. *Operational dynamics* instead refers to the general study of the modus operandi of taxi drivers. This methods want to study and learn from taxi drivers' excellent knowledge of the city [21]. This both are two of the methods how to study social and community behaviors.

Chapter 7

Data acquisition

The GPS traces of a taxi or customer can tell us, with a fair amount of precision, where passengers were picked up, where they were dropped off, which route was taken and what steps the driver took to find a new passenger. We want now to use the classification of [100] for what can be the sources of traffic data. These can be divided into:

- *Location-based data* are usually gathered from traffic surveillance systems, e.g. road cameras, inductive loop detectors, and wireless sensor networks in urban traffic networks;
- *Vehicle-based data* are collected and achieved in our daily lives with the help of GPS-equipped vehicles (such as taxis, probe cars and public buses). The trajectory data usually constitute series of location information with corresponding time stamps, which can be a fundamental source for traffic management;
- *Human-based data*, with the developement of mobile devices, are recording users real-world movements in the form of spatiotemporal data streams, both intentionally and unintentionally. *Intentional data* is when travelers log travel data into applications that share those information, *unintentional data* is when the user is not aware of sharing location information through some app or some automatic service;
- *Supplementary data*: various other sources can provide data, like weather data, road infrastructure data, traffic management status data. They can be provided in spatiotemporal format as well, so that they are compatible with the other data;

Our project is based on the analysis of a dataset defined by GPS locations. It is characterized by rides containing information about the trip, regarding both the vehicle and the customers of the ride. We can think of our data as a vehicle-based dataset, where locations, corresponding time stamps and other information are stored row-by-row. Our dataset is a real-world data generated by yellow taxis in New York City, USA from April 1 to June 30, of years 2015 and 2016. This is a public dataset provided by the Taxi and Limousine Commission (TLC) [73].
The data was recorded through meters installed in each taxi. The dataset specifies for each trip event the pick-up and drop-off GPS locations together with the corresponding timestamp. In total it contains 19 columns, which inlcude *pickup time*, *dropoff time*, *number of passengers*, *trip distance*, *pickup location longitude*, *pickup location latitude*, *dropoff location longitude*, *dropoff location latitude*, *trip fare* within other information that will not be useful in our case. We need now to introduce some basic techniques used to process a spatiotemporal traffic network database that are noise filtering, compression, segmentation and heterogeneous data management. To load and manage the dataset, we have

used Pandas libraries [76] [66] for Python. We also have used Pandas dataframe methods to perform noise filtering and segmentation of the dataset, with the help of Scikit-learn libraries [77].

Chapter 8

Data preprocessing

8.1 Noise filtering

In real-time systems, data are naturally noisy. This may be casued by instrumentation failure, incomplete or missing data or other reasons. Those noisy data could affect the performances of our algorithms, introducing outliers and data carrying information that will deviate from the generalization capabilities of the data mining techniques.
Some of the types of the noisy data include [21]:

- *Missing Data Entries*: Occasionally, the GPS data is not received properly, with a lapse of several minutes or even hours between entries. The reasons for this lapse could be errors in the GPS device or a lack of GPS signal due to a location with low GPS signal coverage. This can lead to a jump between locations since there is no information about locations where the GPS signal was not received.

- *Erroneous Data Entries*: Some of the entries to our dataset include erroneous data, that could be wrong longitude and latitude or time information. These entries usually can be identified looking at the average behaviour of the surrounding entries. They will be labeled as outliers in our dataset.

- *Occupied Flag Improperly Set/Detected*: The state flag that marks vehicles as occupied/available may have some malfunctions or be set not properly by the driver. This may cause some information to be erroneously provided to the central system, negatively affecting the vehicle statistics.

Noise reduction can improve the quality of the information provided by our dataset. Existing methods for noise reduction fall into two major categories [100]:

- *Data Smoothing and Inference Methods*: The most common method in this category is Kalman filtering (KF), a real-time algorithm for traffic estimation using an extended Kalman Filter (EKF), or using the KF in combination with other tools [96]. Other methods use Bayes estimation based data fusion. This process in general is referred to as data fusion.

- *Outlier elimination methods*: In contrast with replacing noisy data with estimated values, outlier elimination methods were proposed to identify outlier noise data and delete them from the dataset. Outliers can be identified on the basis of some metric chosen to analyze the average behaviour of the dataset. We might for example investigate the average duration and speed of normal trajectories and identify some ranges within which entries of our dataset should lay [94].

We decided to use the second method in order to clean the dataset of most of the noisy data. It is possible that the metrics used are not able to identify all the outliers of our system, but we are confident about the fact that most of the noise will be cancelled. The metrics we use to identify the outliers are: longitude and latitude of pick-up and drop-off locations, trip duration, vehicle speed, trip travel distance and trip fare.
We first need to generate a new dataframe containing information about the speed of the vehicle during the trip and a converted time measure for the pick-up and drop-off times. In fact, we need to convert the *date and time* from *YYYY-MM-DD, hh:mm:ss* to the *Unix time* format. *Unix time* is a system for describing a point in time and it is defined as the number of seconds that have elapsed since the *Unix epoch* minus leap seconds. The *Unix epoch* is fixed at 00:00:00 UTC on 1 January 1970. After having converted the time into the new Unix time format, we will be able to trivially compute the speed of each of the trip (that is, each entry of our dataset) as:

$$Speed_{entry} = \frac{Distance_{entry}}{Time_{entry}} \tag{8.1}$$

where $Time_{entry}$ is defined as the difference between the Unix time at drop-off and the Unix time at pick-up: $Time_{entry} = t^{Unix}_{dropoff,entry} - t^{Unix}_{pickup,entry}$. After the new dataframe with the updated time feature and the speed information is generated, we can start the investigation to find and delete outliers from the dataset. Most of the preprocessing phase has been inspired by [87].

8.1.1 Trip duration

First, we want to look at the durations of the trips in order to identify some of the outliers by the duration in minutes of each trip. We want to compare the duration of each single trip with a known threshold for the total duration of the trip. Therefore, we need first to find a trip duration threshold that can consider trips to be valid or not depending on their time duration. Then, given the duration in minute of each trip, we will delete trips from the dataset when they have duration longer than the fixed threshold. The threshold in this case could be chosen according to the New York City Law stating that taxi drivers cannot drive for longer than 12 *hours* a day because of issues related to the physical fatigue of the driver and therefore to safety measures. However, printing the trip duration percentiles division (Table 8.1) of the probability density function distribution of the trip duration we see that 99.9% of the trips has a duration lower than 119.13 *minutes*. Therefore we will use this as an upper limit instead of the limit imposed by the New York City Government.
We call t_{upper} the upper limit of time duration that we have fixed and we suppose that the lower bound t_{lower} is 1 *minute*. We have chosen the lower bound to be 1 *minute* in order to consider invalid all the trips that have negative time duration and to have a safety interval of one minute for erroneous trip duration, being a 1 *minute* ride very close to the 0 *minutes* bound.
Hence, we will filter out all entries that satisfy the following inequalities:

$$duration_{entry} < 1\ min\ \vee\ duration_{entry} > 119\ min; \tag{8.2}$$

Following, we can see in Figure 8.1 the boxplot of the quartile distribution of trips' time durations for the whole dataset before cleaning the dataset. Instead, Figure 8.2 is showing the boxplot quartile distribution of the trip durations after having cleaned the dataset.

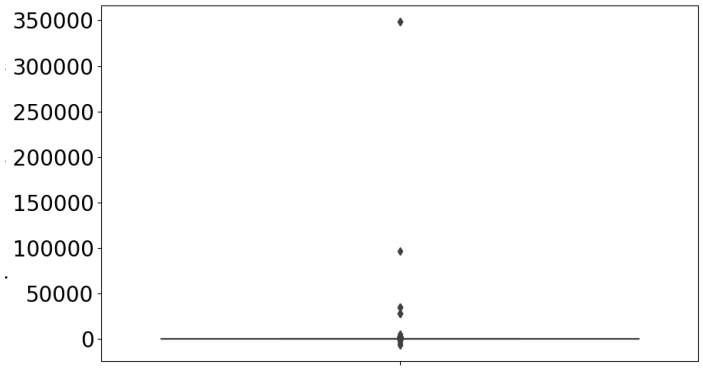


Figure 8.1: Boxplot of the trip durations in *minutes* of all the entries of our dataset for the month of June 2015 before the dataset is filtered using the trip duration metric. As we can see, some of the trips have duration of 5000 *minutes* or higher, which is not allowed by our constraints. Moreover, we can notice that some trips have negative duration, which is physically impossible.

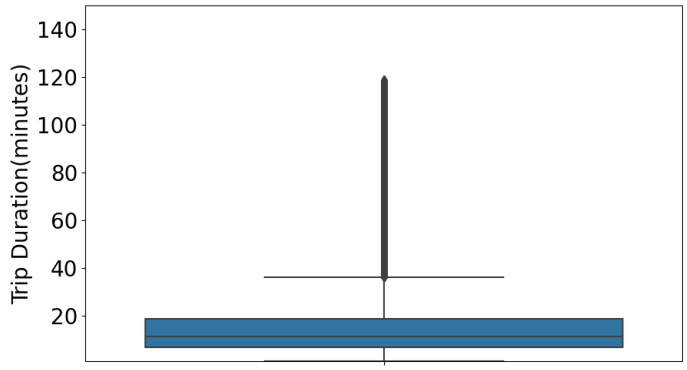


Figure 8.2: Boxplot of the trip durations in *minutes* of all the entries of our dataset for the month of June 2015 after the dataset has been filtered using the trip duration metric. As we can see, all the trips now are within the interval [1, 119] minutes. After the analysis, the 99.9^{th} percentile is telling us that 99.9% of the trips have duration of less than 119 *minutes*.

Table 8.1 contains the percentile analysis for the duration of the trips in the dataset for June 2015.

Percentile analysis for the trip duration

Percentile	Trip duration (minutes)
0^{th}	-6729.22
10^{th}	4.15
20^{th}	5.95
30^{th}	7.63
40^{th}	9.40
...	...
90^{th}	28.35
91^{st}	29.55
...	...
99^{th}	60.23
99.1^{th}	61.92
...	...
99.9^{th}	119.13
$100th^{th}$	3.49×10^5

Table 8.1: In this table we show the percentiles from 0^{th} to 100^{th}, where higher percentile accuracy is used from percentiles from 90^{th} to 100^{th} first and then from 99^{th} to 100^{th}. Decreasing the step of the percentile, gives the possibility of having higher precision when looking for the upper bound of the time duration value.

8.1.2 Speed of the trip

Another of the metrics used to evaluate whether the trips are valid or not is to verify that their average speed is compatible with physical limits of standard vehicles. We want to fix both lower and upper bounds on the speed of the vehicle. If one of the two bounds is not satisfied, the entry of the dataset will be discarded.
First, we can fix as lower bound a minimum speed of 0 $miles/hour$. Instead, the upper bound needs to be found by investigating the statistics of the entire dataset, since we do not have any knowledge of what could be an acceptable average speed for taxis in New York City. Therefore, we will analyze the percentile distribution of the speed probability density function of the dataset, as we already have done for the trip duration analysis. In order to do this, we first need to plot and print the percentiles for the speed probability density function and find a speed upper bound whithin which the majority of the trip speeds lay. Then, given the speed of each trip, we will delete trips from the dataset when they have speed greater than the threshold or smaller than the lower bound. The level of trust of the threshold for the upper bound is given by the percentage of entries of the dataset that have speed within that limit.
We call v_{upper} the upper limit of speed that we will identify via the percentile analysis and v_{lower} will be the lower bound.

Percentile analysis for speed of trips

Percentile	Speed (miles/hr)
0^{th}	0.00
10^{th}	5.63
20^{th}	7.12
30^{th}	8.33
40^{th}	9.47
...	...
90^{th}	19.81
91^{st}	20.48
...	...
99^{th}	34.23
99.1^{th}	34.81
...	...
99.9^{th}	44.85
$100th^{th}$	1.03×10^8

Table 8.2: In this table we show the percentiles from 0^{th} to 100^{th} where higher percentile accuracy is used from percentile 90^{th} to 100^{th} first and then from 99^{th} to 100^{th}. Decreasing the step of the percentiles, gives the possibility of having higher precision when looking for the upper bound of the speed value.

Following, Figure 8.3 represents the boxplot regarding the speeds in our dataset.

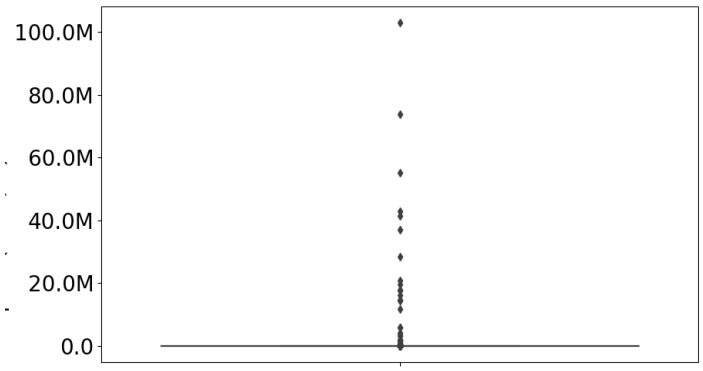


Figure 8.3: Boxplot of the speed, in *miles/hour* in our dataset before filtering the speed outliers. We can see that there are entries of our dataset that have speeds in the order of millions of *miles/hour*. Since none existent ground vehicle is capable of reaching these speeds values, we will discard the corresponding dataset entries, according to the percentiles of the data.

We can see from Table 8.2 that 99.9% of trips have speed under 44.85 $miles/hour$. Therefore, we can keep these 99.9% of data and discard the 0.1% that have velocity such that $v_{0.1\%} > 44.85$ and $v_{0.1\%} \leq 1.03 \times 10^8$.
The average speed of all the rides that have been kept for June 2016 is $v_{ave} = 11.93$ $miles/hour$. Figure 8.4 shows the boxplot of the speed distribution for the cleaned dataset, where all the speed outliers have been removed.

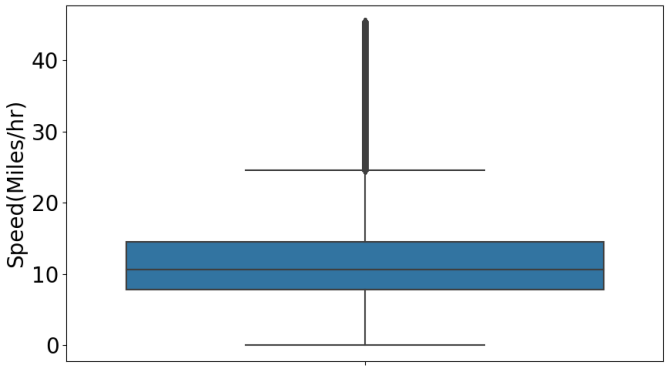


Figure 8.4: Boxplot of the speed, in $miles/hour$ in our dataset after filtering the speed outliers.

8.1.3 Trip distance

Another metric we use to clean the dataset from noise data, is to look at the distance travelled by the trips. In contrast with the previous two metrics, this is closely related to errors in the coordinates of the locations of pick-up and drop-off. It could be that some pick-up or drop-off location coordinates have been mistaken by the GPS device and therefore the corresponding route ends up to be unfeasible for realistic conditions.
We will analyze the percentiles that divide the probability density function of the distance of all the trips that are left in our dataset. Hence, we will print the percentile as in table 8.3.

Percentile analysis for travel distance of trips

Percentile	Distance (miles)
0^{th}	0.01
10^{th}	0.07
20^{th}	0.94
30^{th}	1.20
40^{th}	1.49
...	...
90^{th}	7.0
91^{st}	7.64
...	...
99^{th}	18.64
99.1^{th}	18.88
...	...
99.9^{th}	24.04
$100th^{th}$	155.54

Table 8.3: This table shows the percentiles that divide the probability density function of the distances of each trip. First a step of 10% has been used and it has been seen that 90% of data have an associated trip distance of about 7 *miles*, while the 100^{th} has associated a distance of 155.54 *miles*. Therefore, we need higher accuracy to investigate what is happening for percentiles close to the 100^{th}. This is the reason why from the 90^{th} percentile to the 100^{th} percentile we use a step of 1%. The improved accuracy made us discover that 99% of data lays within a travel distance of 18.64 *miles*. Going further, we use a step of 0.1% form 99^{th} percentile to 100^{th}. Doing so, we figure out that 99.9% of data has a travel distance below 24.04 *miles*. Therefore, we will exlude only 0.1% of the dataset.

As we can see in Table 8.3, 99.9% of trips data have a travel distance smaller than 24.04 *miles*, while 0.1% of the data have travelled a distance that is $t_{0.1\%} > 24.04$ *miles* and $t_{0.1\%} \leq 155.54$ *miles*. This means that we will keep 99.9% of the data that underwent the distance filtering step.
Following, Figure 8.5 shows the boxplot of the travel distance before filtering, Figure 8.6 shows the boxplot of the travel distance after filtering.

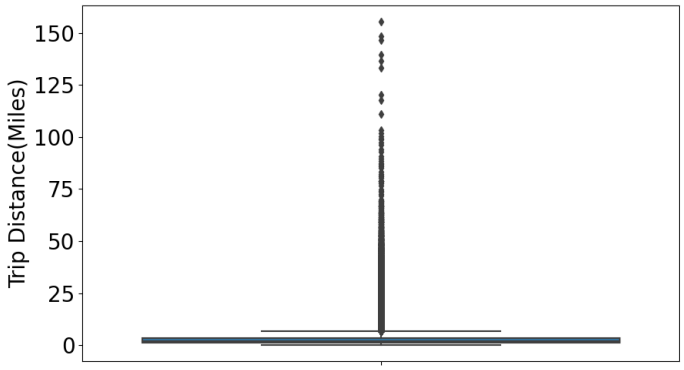


Figure 8.5: Boxplot of the trip distances in *miles* in our dataset before filtering the travel distance outliers.

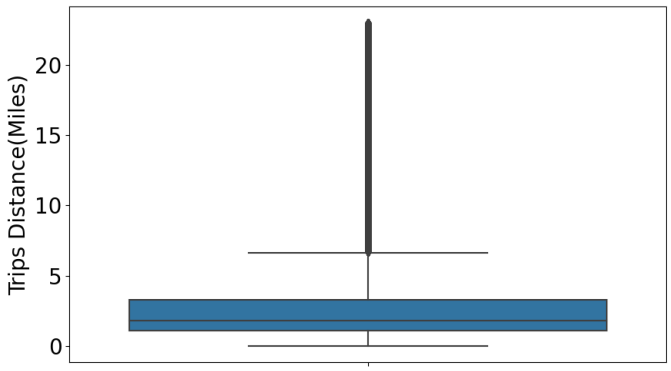


Figure 8.6: Boxplot of the trip distances in *miles* in our dataset after filtering the travel distance outliers.

8.1.4 Trip fare

The next metric we want to use to evaluate if the entries of the dataset are outliers, is the fare of the trip, which is the total cost of the trip for the customer. The trip fare usually is computed with algorithms that accounts for the taxi call and dispatch service, the travel time of the ride. Additional charges depend mainly on the time of the day that accounts for rush hours, the fact that trips could require mainly fast track lines, road tolls, the tip and many other factors like weather conditions [74]. Because of all these factors, trips fare is difficult to predict and it is even more difficult to estimate what could be an upper bound for our case, given the great number of influencing factors and an incomplete knowledge of the dynamics behind the taxi service dispatcher of New York City. Therefore, we want to use statistical analysis to try to find an upper bound for the trip fare for our case so that to exlude entries that do not enter on that threshold we will fix.
Indeed, it is probable that if the fare is greater than the threshold, some error may have occured when acquiring GPS and ride status data. An excessive fare might also simply

suggest that a trip is so peculiar that it should not enter our taxi demand modelling, since it could deviate from a generalization of the forecast process.
We will do a percentile analysis of the probability density function of the trip fare of our dataset. Table 8.4 shows what are the bound values for the percentiles of our dataset.

Percentile analysis for fare of trips

Percentile	Fare (US Dollars)
0^{th}	-317.8
10^{th}	6.80
20^{th}	8.16
30^{th}	9.30
40^{th}	10.79
...	...
90^{th}	30.25
91^{st}	32.16
...	...
99^{th}	69.99
99.1^{th}	69.99
...	...
99.8^{th}	80.55
99.9^{th}	94.8
$100th^{th}$	8002.8

Table 8.4: This table shows the percentiles that divide the probability density function of the fares of each trip. First a step of 10% has been used and it has been seen that 90% of data have an associated fare of about \$30, while the 100^{th} has associated a fare of \$8002.8. It is unlikely that a customer would be willing to spend that amount for a taxi trip. Therefore, we need higher accuracy to investigate what is happening for percentiles close to the 100^{th}. This is the reason why from the 90^{th} percentile to the 100^{th} percentile we use a step of 1%. The improved accuracy made us find out that 99% of data have an upper bound fare of \$69.99. It is still significant the difference between the 99^{th} and the 100^{th} percentiles. Going further, we use a step of 0.1% form 99^{th} percentile to 100^{th}. Doing so, we figure out that 99.9% of data have a fare below \$94.8. This fare, though, is not close to the \$80.55 bound of the 99.8^{th} percentile. Therefore, we will exclude only 0.2% of the dataset and keep the 99.8% of data that has a fare upper bound of \$80.55.

Figure 8.7 and 8.8 show the boxplots of the taxi trip fare before and after filtering the dataset, respectively.

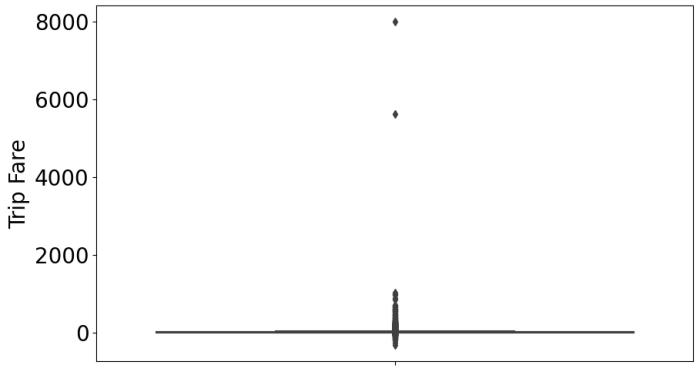


Figure 8.7: Boxplot of the trips fares, in $USDollars$ in our dataset before filtering the ride fare outliers.

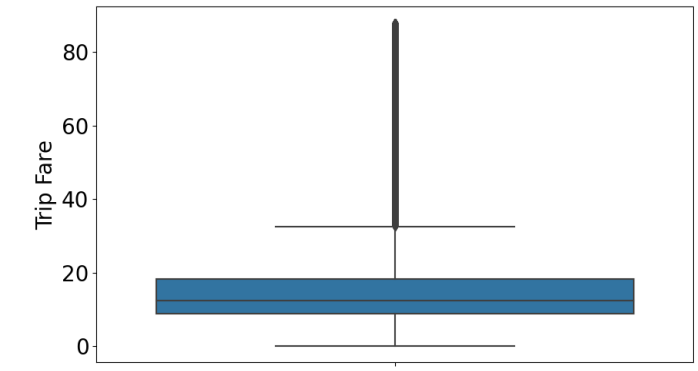


Figure 8.8: Boxplot of the trips fares, in $USDollars$ in our dataset after filtering the ride fare outliers.

8.1.5 Pick-up and drop-off locations coordinates

The *terrain* of a city is a continuous two-dimensional area (i.e. a subset of $\mathbb{R}^2$) and can be defined as [21]:

$$\mathbb{D} = [lon_{min}, lon_{max}] \times [lat_{min}, lat_{max}] \tag{8.3}$$

where $\mathbb{D} \subseteq \mathbb{R}^2$ is the area covering all of the GPS entries within the city limits that are defined by lon_{min}, lon_{max}, lat_{min}, and lat_{max}. The boundaries for the city of New York are published on the website of the Department of City Planning (DCP).
Therefore, we can define the area $\mathbb{D}$ as:

$$\mathbb{D} = [-74.257159, -73.699215] \times [40.495992, 40.915568] \tag{8.4}$$

Our filtering algorithm will remove from the dataset all the trips that have pick-up or drop-off coordinates outside of the city boundaries. The following will be the conditions to be verified for each entry of the dataset:

$$
\begin{aligned}
-74.257159 \leq p_{lon} \leq -73.699215 \\
40.495992 \leq p_{lat} \leq 40.915568 \\
-74.257159 \leq d_{lon} \leq -73.699215 \\
40.495992 \leq d_{lat} \leq 40.915568
\end{aligned}
\tag{8.5}
$$

where $[p_{lon}, p_{lat}]$ are the coordinates of a generic pick-up point and $[d_{lon}, d_{lat}]$ are the coordinates of a generic drop-off point.
Figures 8.9 and 8.10 show, respectively, the pickup and dropoff locations that have been removed from the dataset because they were outside the city boundaries.

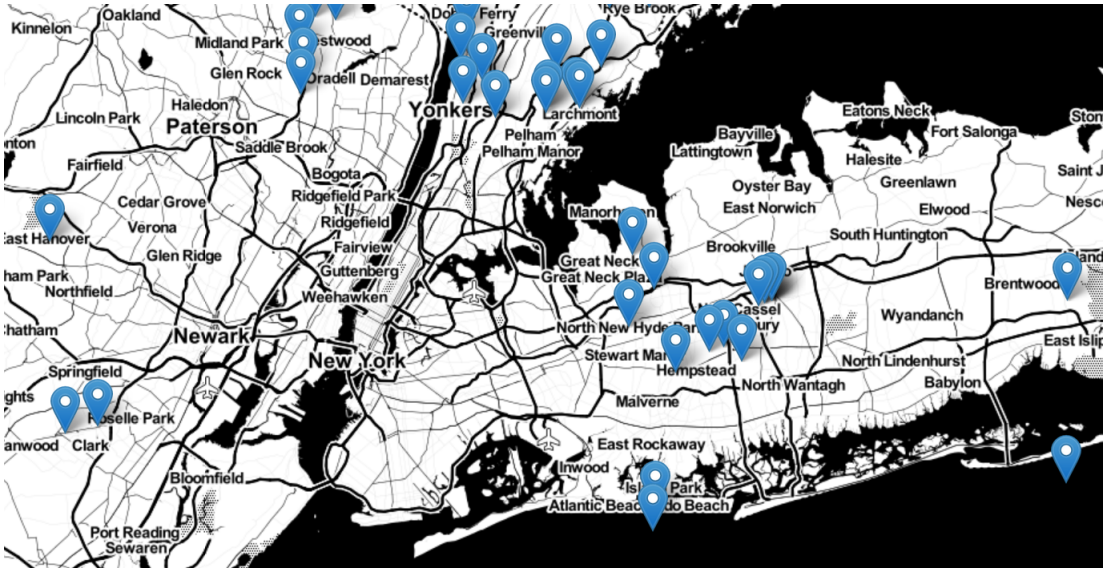


Figure 8.9: Taxicab rides pick-up locations that are placed outside the NYC boundaries, on a map of the city of New York and surrounding areas.

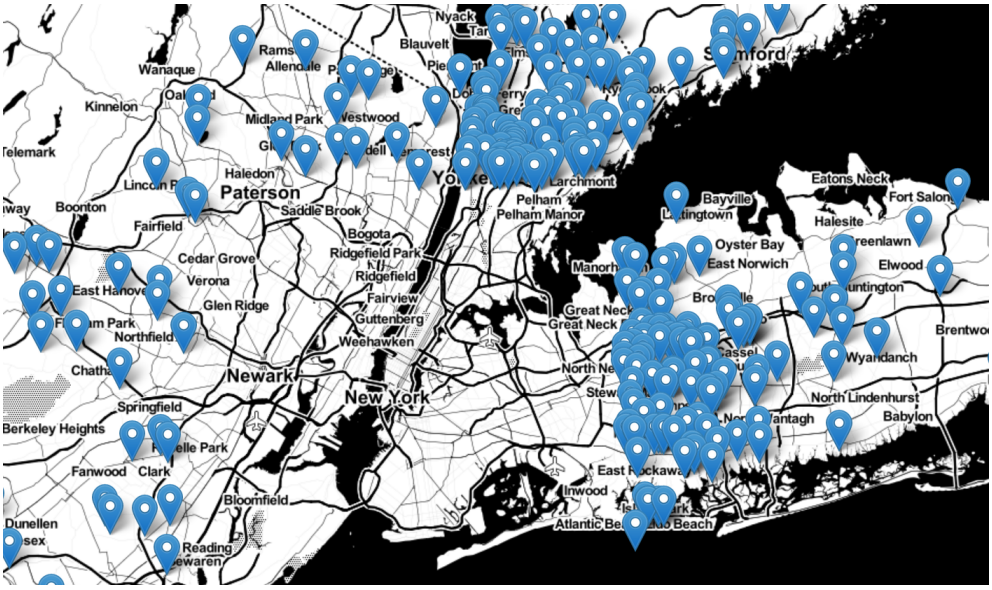


Figure 8.10: Taxicab rides drop-off locations that are placed outside the NYC boundaries, on a map of the city of New York and surrounding areas. As we can see, many drop-off locations are outside the city limits.

Figures 8.11 and 8.12 show, respectively, the pickup and dropoff locations that are still part of the dataset since they are within the area defined by the city boundaries of Equation 8.4. The two images show 100 samples each, from the filtered dataset locations. These are only portion of the pick-up and drop-off locations that would include millions of entries.

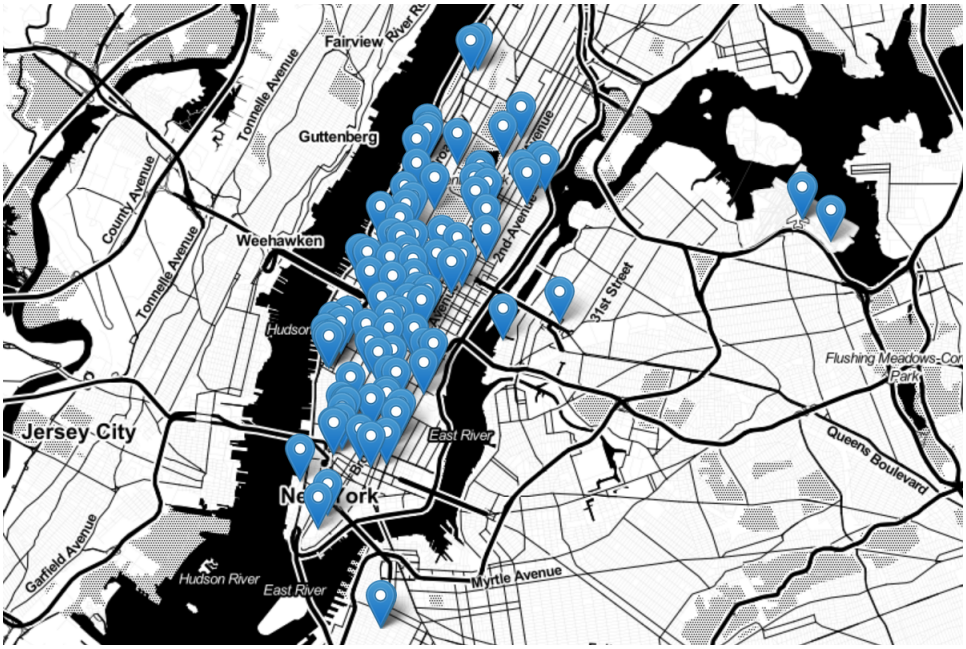


Figure 8.11: Taxicab rides pick-up locations that are placed inside the NYC boundaries, on a map of the city of New York and surrounding areas. These are only 100 random samples from the entire dataset of pick-up locations inside the city boundaries.

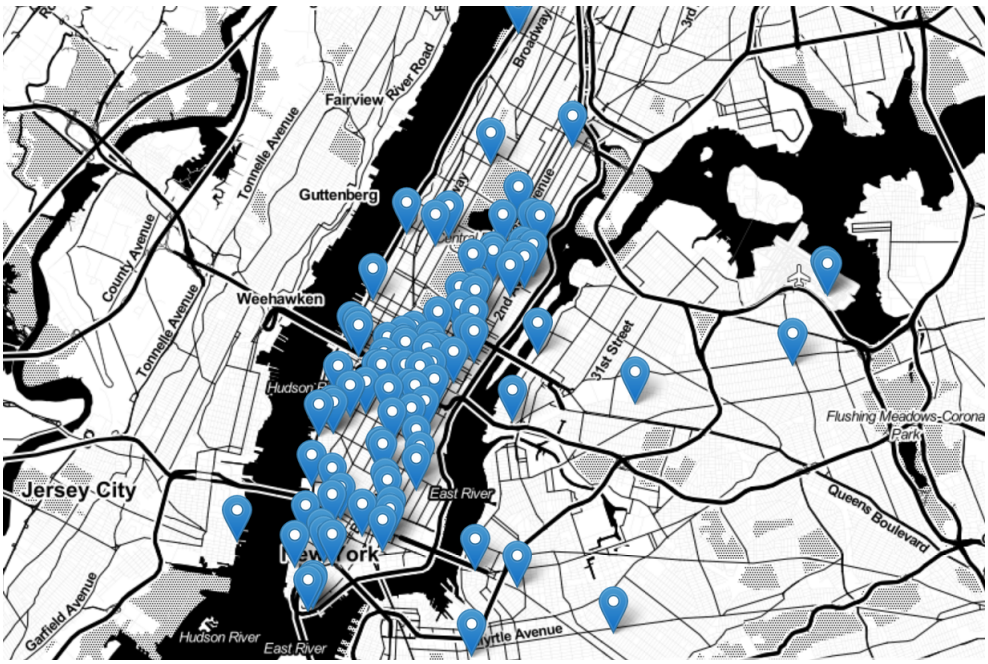


Figure 8.12: Taxicab rides drop-off locations that are placed inside the NYC boundaries, on a map of the city of New York and surrounding areas. These are only 100 random samples from the entire dataset of drop-off locations inside the city boundaries.

After having completed the filtering of the data following the aforementioned criteria, we will end up with a new dataset, that we can consider to be clean of most of the outliers that could badly affect the perdormance of our forecasting. The filtering operation have removed about 3.3% of the original entries of the dataset. Therefore, we will end up having 96.7% of the original data.

8.2 Segmentation

Segmentation is part of data preprocessing and it is used to reorganize our dataset so that it gains meaning for the purposes of our analysis. Segmentation not only reduces computational complexity, but also enables us to mine more specific patterns, such as day-to-day or regional traffic patterns.
Generally, segmentation is done on the basis of three different criteria:

- *Based on time intervals*: Traffic data usually are characterized by some sort of periodicity. The demand for taxi services exhibits, like other modes of road transportation, a periodicity in time on a daily basis (see Figure 8.13) that reflects patterns of the underlying human activity [68]. We can therefore divide the rides dataset into smaller segments that could represent the rush hours at different moments of the day. We can also use hourly time bins by calculating the aggregated demand in every hour [94]. These data segmentation helps in highlighting what are the time patterns of our dataset.

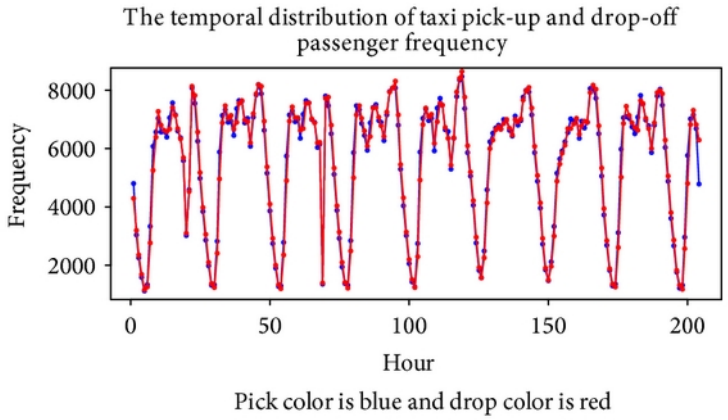


Figure 8.13: Example of taxi demand when data have been organized in hourly time bins. We can see a strong periodicity of 24 *hours*.

- *Based on space*: Data can be divided into sub datasets by regions for regional pattern study or specific research objectives. Some of the possibilities for spatial segmentation could be of dividing the map previosuly defined as $\mathbb{D}$ with a *grid* [21]. This simple approach can decompose a city map into equal sized areas. The sub-areas of the map can be further clustered using some distance function that describe some property within the sub-areas so that they are placed in the same cluster. Another possibility is when we have a digital map of the city available, which means we have a network of pivot points represented by nodes V in a graph (V, E). Streets in this case are composed of one or more edges of the graph. An example of the use of a graph to represent the road network is given in [22] to study the presence

of anomalies in the traffic flow, where regions are nodes and roads are edges. A *hierarchical road network* can be used to ease the manipulation of the digital map. This is one of the clustering methods used to cluster a digital map. We will assign roads to groups of higher and lower importante in the network, in a multi-level organization of the road set.
Another way to spatially segment data is to use a cluster algorithm to divide the set of rides in regions. Spatial clustering is one of the main techniques used in spatial data mining.

- *Based on other factors*: Other factors such as weather can influence the traffic on a road network and therefore also the number of taxi rides. Especially rain is influencing traffic not only for changes in the quantity of people deciding to move with personal vehicles but also for an increase in car accidents and a consequent traffic slowdown [51].

Segmentation in general is useful to brought up the characteristics and information of our dataset that will be useful for our purposes. In fact, our aim is to predict the number of pick-up requests that will happen in the next hour, knowing what was the historical trend of the pick-ups. Morevover, we are interested in differentiating between areas of the map in order to have a prediction that spans both in time and space, so that we will be able to suggest vehicle where to head to pick-up the most customers and avoid imbalances. Therefore, we need to have a dataset where the information about the hourly pick-ups for each region of our dataset is easily accessible.

8.2.1 Temporal clustering

We divided the dataset into hourly time bin that is, we will have a set of time slots that all together will decribe completely the time span of our dataset. For instance, if we consider the dataset of June 2016, knowing that June is a 30-days month, we can compute the total number of time bins needed to divide the dataset by hour. This will simply be $24 \cdot 30 = 720$ *time bins*. For the dataset of April 2016 and May 2016 we will have, respectively, 720 *time bins* and 744 *time bins*.
It may happen that some of the bins have no pick-up occurring for that area and hourly time bin. This happens usually in areas of low demand. Even if rare, we need to fill those gaps, to not incur in errors and misinterpretation of the dataset. What we do, is to fill those gaps with time bins having as number of pickups the average of the number of pickups of the last existing time bin in the sequence before the gap and the upcoming existing bin. In Figure 8.14 we show an example of what we intend by filling the missing data in the time bins of our dataset for some regions. We will fill the gaps with time bins containing the sum of the two time bins adjacent to the two ends of the gap (one time bin for case *b*) and *c*) of Figure 8.14) divided by the number of time bins in the gap *plus* the number of adjacent time bins. Therefore, referring to the proposed example, in case *a*) we will use Equation 8.6 in case *b*) we will use Equation 8.7 and in case *c*) we will use equation 8.8, similarly to case *b*), where t_i is the i-th time bin identifier and p_i is the number of pickups occurring during the i-th time bin.

$$p_7 = p_8 = p_9 = p_{10} = \frac{p_7 + p_{10}}{4} \tag{8.6}$$

$$p_1 = p_2 = p_3 = p_4 = \frac{p_4}{4} \tag{8.7}$$

$$p_{15} = p_{16} = p_{17} = \frac{p_{15}}{3} \tag{8.8}$$

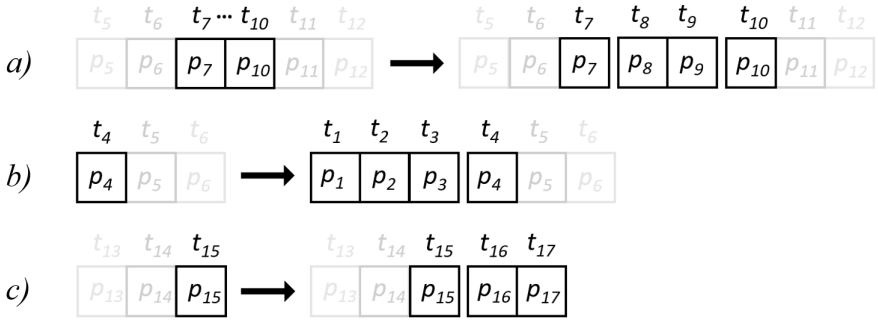


Figure 8.14: In example *a*), the missing data are in the middle of the time bins sequence, we will add time bins t_8 and t_9. In case *b*) the missing value are at the beginning of the time sequence, we will therefore add time bins t_1, t_2 and t_3. In case *c*) the missing values are at the end of the time bins sequence, therefore we will add time bins until the number of supposed time bins is reached, in this case we add t_{16} and t_{17}.

Table 8.5 is showing how data have been segmented so that they are meaningful to our purposes.

Segmented dataset

Spatial cluster	Temporal cluster	Number of Pick-ups
Region 1	Hour 1	50
	Hour 2	25
	⋮	⋮
	Hour n	34
Region 2	Hour 1	18
	Hour 2	20
	⋮	⋮
	Hour n	12
⋮	⋮	⋮
Region m	Hour 1	123
	Hour 2	108
	⋮	⋮
	Hour n	110

Table 8.5: How the main body of the dataset is structured. For each region of the map, we have the time bins that divide into equal spots the time of the entire month in hours. n is 720 if the dataset is a 30-days month, and 744 if the dataset is a 31-days month. As a third column, we have the number of pick-ups that happened for the specific region at the specific time bin.

The next step will be to use this dataset and integrate additional information such as the day of the week and weather to obtain a new more complete dataset with features that are influencing the prediction of the number of pick-ups.

8.2.2 Spatial clustering

During segmentation, as anticipated, our goal is to obtain a dataset that is both temporally and spatially segmented, which means we have extraced information from the entries of our dataset to position them in new temporal and spatial sets that will identify the original dataset entries from then on. This means that the single trip, represented by an entry of the dataset will lose its specific information about pick-up and drop-off locations coordinates and the respective pick-up and drop-off date and time. Instead, this information will be substituted with location and time that are now identifying the entry as belonging to a group of data.
For what concern the spatial clusters, they have to be defined and at the same time the different taxi rides have to be assigned to these clusters. Then, the single taxi ride will be identified by a certain cluster pick-up location, that at the same time identifies other tasks belonging to the same cluster.
As explained is Part 1, *clustering* is the process of grouping a set of physical or abstract objects into classes of similar objects. Classification also is an effective method to distinguish groups or classes of objects, but usually requires costly collection and labelling of a large set of training tuples or patterns, which the classifier uses to model each group. Instead, clustering does not require such labeling at all [56].
Clustering can be used to deal with large dataset and is frequently used in data mining applications, and is a useful tool in geographical data analysis. Geographic data can in general have different forms, but in our case we are dealing with point data, which are defined by the coordinates of the pick-up locations in particular. One of the most trustest application to divide geographic points into clusters is to use hexagons, in a similar way to how grids are used. Indeed, hexagons have some nice properties that made them on of the most accurate way to divide a map into areas, and then assign the points to one of the areas defined by the hexagons, based on the location of the points. The main idea behind the use of hexagons, is that they reduce the sampling bias form edge effects of the grid shape, which is related to high perimeter to area ratios. Theoretically, the circle would have the lowest ratio, but it cannot form a continuous grid. Thus, hexagons are used since they are the closest to circle that can still form a grid. Other advantages of hexagons are the possibility of represeting curves and curvatures of the surface, the easier management of neighbor definition and neighbors search [16].
In Section 1 we mentioned how clustering algorithms such as DBSCAN can be used to create groups of trajctories [47], that are sequences of points in time, hence more difficult to describe and classify by similarity because of the strong relation they have with the time dimension. Given the kind of data and the freedom we have in deciding how many regions we want in our map, we decided to use the k-means clustering method to divide our set of pick-up point data.
Our objective is to divide the entire set of pick-ups into regions. Each region will be labeled by a center, identified by latitude and longitude.

8.2.3 K-means algorithm for spatial clustering

We remind that we are interested in k-partition clustering that is, given a set of n points in Euclidean space (in general it is a d-dimensional space $\mathbb{R}^d$) and an integer k, find a partition of these points into k subsets, each with a representative, also known as a center [10]. We can distinguish between three different formulations of k-partitioning clustering [49] and these depend on the objective function being used: k-center, where the goal is to minimize the maximum distance between a point of a cluster and the closest cluster representative, k-median, where the goal is to minimize the sum of the distances between the points and their closest cluster representatives, and k-means, where the goal is to minimize the sum of the squares of these distances. All these three formulations are $\mathcal{NP}$-hard, but constant factor approximations of the optimal value is known. The k-means clustering algorithm is an unsupervised algorithm that is used for quickly predicting grouping within an unlabeled dataset. It is by far the most popular clustering algorithm used in scientific and industrial applications [55]. The goal of clustering is to assign data points to a finite system of k subsets (clusters). These subsets do not intersect (however, this requirement is sometime violated in practice) and their union is equal to the full dataset with the possible exception of outliers.

$$X = C_1 \cup \cdots \cup C_k \cup C_{outliers}, C_i \cap C_j = 0, \forall i, j \text{ such that } i \neq j \tag{8.9}$$

The advantage of k-means is its simplicity: starting with a set of randomly chosen initial clusters, one repeatedly assigns each input point to its nearest center, and then recomputes the centers given the point assignment [10]. This local search, called Lloyd's iteration, continues until the solution does not change between two consecutive rounds. Theoretically the algorithm is not efficient and the running time can be exponential in the worst case. In practice, k-means remains a fast and simple clustering algorithm.
k-means is under the class of partitioning relocation methods, together with k-medoids and probabilistic clustering. Because checking all possible subsets is computationally infeasible, certain greedy heuristics are referred to collectively as iterative optimization. As previously outlined, iterative optimization refers to different relocation schemes that iteratively reassign points between the k clusters. Hierarchical clustering, another of the clustering methodologies, do not revisit clusters after construction, while relocating algorithms can gradually improve clusters, by iterating.
Partitioning relocation clustering approaches starts with the definition of an objective function, then also, we can distinguish between clustering detection methods by other aspects. One group of techniques examine the points pairwise to see if they are closer to each other than expected or if similar measurements (variables measured at the point locations) are closer to each other than expected [63]. Pairwise (between all combinations) distances or similarities can be used to compute the measures of intercluster and intracluster relations. From these distances we can obtain the value of the objective function. In iterative improvement approaches, such pairwise computations would be too expensive. For this reason, in the k-means and k-medoids approaches, each cluster is associated with a unique cluster representative, such that we will compute the distance function for all points but only with respect to the cluster representatives. In this case, the computation of an objective function becomes linear in the number of elements N and the number of clusters k. In k-medoids, the cluster representatives are actual elements of the clusters, while in k-means, the representatives are abstract points, computed as the mean between the points belonging to the cluster. k-means is sensitive to the presence of outliers and noise data, since a small number of such data can substantially influence the mean value.

The k-means algorithm is sensitive to outliers because an object with an extremely large value may substantially distort the distribution of data. This effect is particularly due to the use of the squared-error function as a distance criterium. This squared error criteria is, typically [67]:

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} |p - m_i|^2 \tag{8.10}$$

where m_i is the mean of all the points in the i-th cluster, that is the representative of cluster i. With k-means algorithm, the representative of the set is always the mean of the points belonging to the set.
The cluster is found based on a distance function that expresses the distance between elements of the cluster and its centroid. The objective function to find for the best cluster division will be the sum of these distance functions. For example, the L_2 norm-based objective function, that is the sum of the squares of the distances between the points of the cluster and the corresponding centroid, is also equal to the total intracluster variance.

$$E(C) = \sum_{i=1}^{k} \sum_{p_j \in C_i} ||p_j - m_i||^2 \tag{8.11}$$

Following, we want to talk about Lloyd's method, the most famous algorithm implementation for k-means. Usually referred to simply as k-means, Lloyd's algorithm begins with k arbitrary centers, typically chosen uniformly at random from the data points. Each point then is assigned to the nearest centroid, and each centroid is recomputed as the center of mass of all points assigned to it. These two steps (assignment and center calculation) are repeated until the process stabilizes. Lloyd's algorithm is based on the simple observation that the optimal placement of a center is at the centroid of the associated cluster [49].
Formalizing we will have that given any set of k centers Z, for each center $z \in Z$, let $V(z)$ denote its neighborhood, that is, the set of data points for which z is the nearest neighbor. In geometric terminology, $V(z)$ is the set of data points lying in the Voronoi cell of z. Each stage of Lloyd's algorithm moves every center point z to the centroid of $V(z)$ and then updates $V(z)$ by recomputing the distance from each point to its nearest center. These steps are repeated until some convergence condition is met.

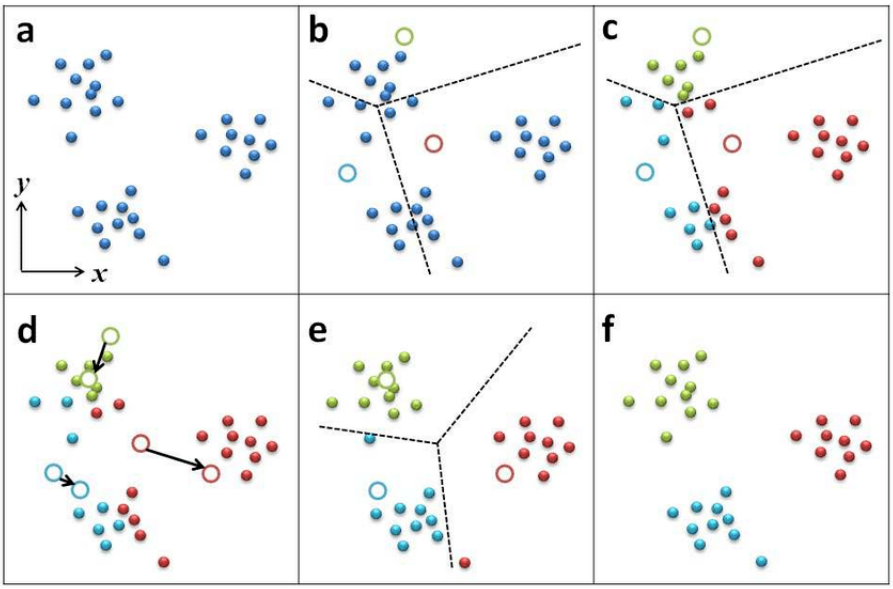


Figure 8.15: A schematic illustration of the k-means algorithm for two-dimensional data clustering with $k = 3$.
a) Set of data points to be clustered in $two - dimensional$ space.
b) The hollow circles represent the random location of the cluster centers. We will associate each point of the dataset to the center that is cosest in euclidean distance.
c) Now the space is divided into three subspaces through three decision boundaries, each containing the corresponding data points whose closest center is within the subspace. At future iterations, it may be that data points will not belong to the same cluster (subset).
d) The center of the cluster is computed as the mean of the data points within each cluster. The black arrows show how the centers are moving from the starting positions to the mean positions.
e) We assign data points to clusters according to the new centers locations. Steps *c*) and *d*) are repeated iteratively until convergence.
f) This is the final clusters we obtain from the original dataset.

The method is relatively scalable and efficient in processing medium size data sets because the computational complexity of the algorithm is $O(nkt)$ where n is the total number of objects, k is the number of clusters and t is the number of iterations. Normally, $k << n$ and $t << n$. The method often terminates at a local optimum. In fact, randomness introduces heuristic in the cluster search. [18] shows how to scale k-means clustering to very large data sets through sampling and pruning. In fact, they state how it would be in principle necessary to scan multiple times the dataset, and for large dataset it would be prohibitively expensive. Moreover, note that Lloyd's algorithm does not specify the initial placement of centers. Therefore, the algorithm might encounter scenarios in which it degenerates to brute-force search, so very little pruning takes place and the algorithm has to undergo a brute force $O(knt)$ search [49]. By augmenting k-means with a very simple, randomized seeding technique, we obtain an algorithm that improves both speed and accuracy of k-means [7].
The default k-means algorithm of `MiniBatchKMeans()` available with Python libraries Scikit-learn [77], makes use of k-means++ to choose the initial points. k-means++ is an algorithm to choose the initial values (also called seeds) for the k-means clustering algorithm. This should speed up convergence of the main clustering algorithm. As is well-known, a proper initialization of k-means is crucial for obtaining a good final solu-

tion. The recently proposed k-means++ initialization algorithm achieves this, obtaining an initial set of centers that is provably close to the optimum solution [10]. A major downside of the k-means++ is its inherently sequential nature, which limits its applicability to massive data: one must make k passes over the data to find a good initial set of centers.
Using k-means++ improves both the speed and the accuracy of k-means algorithm. [7] proves that their k-means++ algorithm is $O(logk)$-competitive on all data sets while [75] proves it also gives a constant approximation to the optimum when data are well distributed.
Some naïve implementation of the k-means++ will have to make k passes over the data in order to produce good initial centers. In [10] they develop a parallel version of the k-means++ initialization algorithm and they empirically demonstrate its practical effectiveness. Also, we can change the number of iterations that are used by the algorithm in order to decrease the computation time, therefore loosing optimality.
`MiniBatchKMeans()` function implements a mini-batch optimization for k-means clustering. This reduces computation cost by orders of magnitude compared to the classic batch algorithm while yielding significantly better solutions [86]. The standard batch algorithm is slow for large dataset. As outlined before, the classic batch k-means algorithm is requiring $O(knt)$ computations. In mini-batch k-means, at each iteration, a subset b (that we call mini-batch) of randomly sampled elements $x \in X$ where X is the set of all the points in our dataset is used to compute then the standard Lloyd's algorithm. This method drastically cut the computation time required for clustering the dataset.

Since, as we mentioned, the k-means algorithm needs the number k of clusters to be known a-priori, we want to use some criteria to define what might be the optimal number of cluster for our case.
We know, from the first analysis of the dataset that the average speed of the vehicles involved in the trip of our dataset is about 12 $miles/hour$. Hence, we can do some reasoning about the region-to-region movements of the vehicles. We can suppose that, having a one hour prediciton for what will be the pick-ups in the different areas of the city, we need to tell vehicles with little advance where to head in the next hour. We hope that the vehicles are already distributed in such a way that they are not far from the optimal configuration, since the forecast algorithm has already worked on distributing the vehicles according to the past predictions. Consequently, we can think of a limited time that the vehile require to move from one area to another, hopefully with some customer on board, so that to have the most of the next time slot available to serve the requests in the vicinity of the assigned region.
We will see that, with 30 regions, taken one of the clusters, approximately 75% of the remaining clusters will be at an average distance $d > 2; miles$ from the chosen cluster, while the remaining 25% will be at an average distance $d < 2; miles$. This reflects the fact that moving to adjacent clusters will require the vehicle about $10minutes$, since it is travelling at an average speed of 12 $miles/hour$. Therefore, if a vehicle spends 10 $minuts$ to move from the center of a region to the center of a neighbor region, it will have about 50 $minutes$ left to work in that region.
On the other hand, we can think of fixing a minimum intercluster distance between the different regions of the map. This will avoid to have too narrow regions, that would otherwise lead to too high particularisation of the map and this would result in loss of freedom from the vehicle assignment point of view. Indeed, when moving vehicles to satisfy rides, it is likely that the vehicles will move far away (with narrow regions) from their original

location, and as a result the optimal distribution of vehicles will be lost. This is why we think it is not worth to have regions that are closer than $2-3$ *minutes* travel time distance. We have decided that the minimum distance between clusters should be around 2.5 *minutes* travel time, that with a speed of 12 *miles/hour* corresponds to about 0.5 *miles* minimum intercluster distance.
The intercluster distance is computed using the coordinates of the clusters centers in latitude and longitude and computing the haversine distance between the two points. The haversine distance has been computed with a method of the object `geo` defined in the library `gpxpy` (this is a Python library used for parsing and manipualating GPX files. GPX is an XML based format for GPS tracks).
In Table 8.6 we present the results of one instance running the function `MiniBatchKMeans` with varying number of clusters k. We start from $k = 10$ and we try multiples of 10 until $k = 90$.

Distances between cluster centers

N. Clusters	Minimum distance (*miles*)	N. clusters for $d < 2$	N. clusters for $d > 2$
10	0.99	2	8
20	0.79	4	16
30	0.46	7	23
40	0.45	8	32
50	0.41	11	39
60	0.29	14	46
70	0.29	17	53
80	0.20	21	59
90	0.29	20	70

Table 8.6: This table shows an instance of the problem with data relative to June 2015. The first column represents the number of cluster k for that specific call of the `MiniBatchKMeans` function. The second column represents the minimum distance between two differents cluster centers. The third column represents the average number of clusters that results closer than 2 *miles* if we measure the distance between one center and the other for all the k cluster centers of the set. The fourth column represents instead the average number of clusters that are more than $2; miles$ away from each cluster taken one at a time.

For our case, we chose the number of cluster k to be 30, so that the minimum intercluster distance will be roughly $0.5; miles$. Consequently, it means that all our pick-ups data have been assigned to one of the 30 clusters that divide the entire map into subregions. A visualization of a sample of our pick-up data divided per region is represented in Figure 8.16.

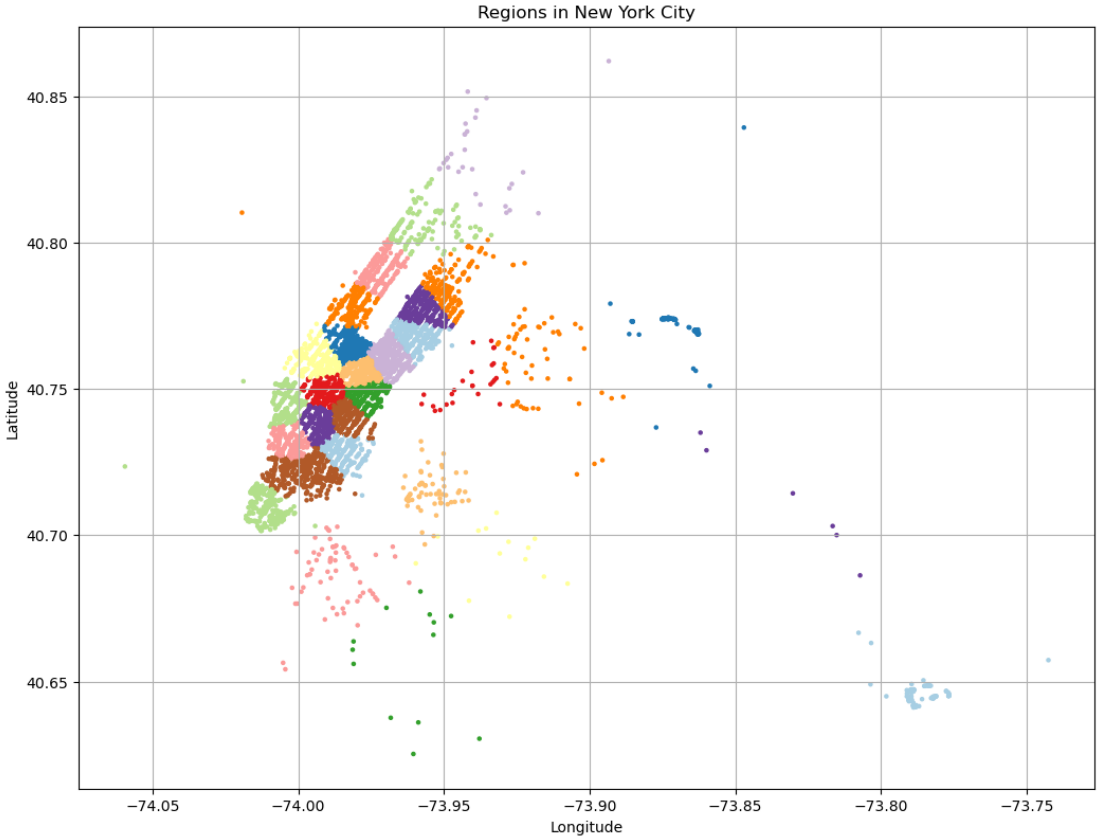


Figure 8.16: This example shows a sample of 7000 pick-ups from our dataset divided into the 30 clusters. Each cluster, then, will have associated a representative, which is an abstract point computed as the mean of the elements inside the cluster. The centers have their own longitude and latitude coordinates.

8.3 Feature selection

Many are the factors that can influence the taxi traffic in a city. Other than the very strong spatial and temporal dependencies, there could be other influences for the trend in pick-up demand to change. Additional factors can be modeled as a separate input to the problem. Such external factors who might affect the number of pick-ups are, for example, weather conditions and city events [61]. The reason why to add external factors to the model is to try to improve the prediction accuracy. Unfortunately, not all the external factors that are usually tested in the literature, such holidays, weather conditions, particular events like concerts, traffic congestions data and road constructions are not easily accessible.
Weather data is one of the easiest dataset to obtain, because of the many fileds in which it is involved. [61] makes use of a very detailed weather data record, which include hourly data regarding: *temperature*, *wind speed*, *precipitation*, *weather type*, *snow fall*, *snow depth*, *sustained wind speed*, *weather type*. A complete analysis of what are the most influencing weather factors on traffic in a city environment is explained in details in [71]. They show how especially pavement conditions can highly influence the capacity of roads and their occupancy, therefore influencing the traffic congestions that then generate traffic slowdowns. Sadly, it is hard to quantify pavement conditions and therefore it is even harder to be able to accurately build models to relate pavement conditions to road traffic.

The study [28] indicates how precipitations, cloudiness, and wind speed have a clear diminishing effect on traffic intensity, while maximum temperature and hail have a significant increasing effect on traffic intensity.
Weather effect has also been analyzed in [43], where they try to estimate what is the effect of temperature and precipitation on the taxi demand of New York City. Analyzing the effect of temperature, they find out that from a mild temperature of 64°F to colder temperatures and to hotter temperatures, the frequency of pick-up requests is becoming higher towards both very cold and very hot temperatures. This means that temperature has a great influence on the taxi demand as we can suppose that with cold temperatures, people ask for taxis to move within the city to avoid getting cold, while very hot temperature might coincide with good weather days, presumably summer days, were many people are visiting the city and the demand for taxi services becomes high also on average, with less fluctuations. Instead, the effect of the temperature, is that fluctuations of the taxi demand are accentuated. This is given probably by the fact that during rainy days people try to move as less as possible and plan ahead exits. Then, it is likely that most of the planned exits occur at the same moments, that are the peak times.
Our dataset covers the periods that go from April 2015 to June 2015 and from April 2016 to June 2016. In this time span we have examples of both cold days in April, where we still reach temperatures of around 20°F and hot periods in June, where temperatures reach about 86°F.
In [69] they use both amount of precipitations (in *mm*) and the air temperature (in °C) as additional features to their data to predict taxi demand in the city of Tokyo. We would like to include the same weather data in our dataset, but unfortunately, the weather dataset that we have available has very poor rain data, because of very low accuracy in the measurements of the amount of precipitations. Hence, we decided to use temperature data, so that to capture at least the effects of temperature peaks on the taxi demand.
We decided to include weather conditions [93] although [97] is showing that the influence of weather is not very strong on the prediction of the number of pick-ups. In Figure 8.17 we can see how weather is not as influencing as other parameters like *pick-up locations*, *day of the week* or *drop-off locations*. Instead, we can see from Figure 8.18 that models where the *pick-up locations* feature and the other features are combined have similar performances. The reason probably is that the information about the pick-up location is so informative that makes the effect of other factors very small. In both images, the accuracy of the prediction is measured with the sMAPE (symmetric mean absolute percentage error), a common measure of relative error in prediction problems. The drop-off location has been proved by [97] to be a high influencing factor in predicting the taxi demand. This means that the drop-off pattern has a close relation to the pick-up pattern. In our case, it is difficult to include the drop-off locations information on our dataset, because of the data segmentation we have used. In fact, many pickups are gathered together into time bins and it would impractical if not impossible to include the different drop-off locations for these pickups. Therefore, we are forced to loose the information about the drop-off locations, because of the nature of our dataset.

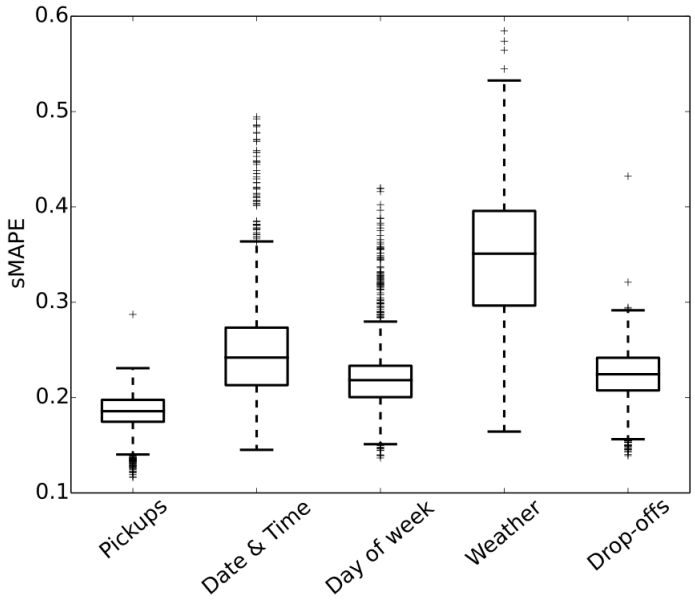


Figure 8.17: Example of prediction performances with respect to different single impacting factors [97]. The performance is evaluated using the sMAPE.

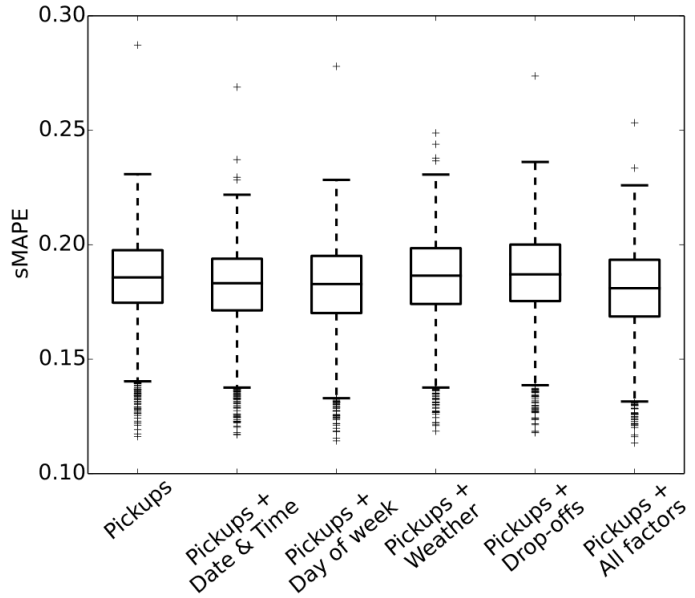


Figure 8.18: Example of prediciton performances on different impacting factors, where pick-up location information is integrated with the other factors one at a time [97]. The performance is evaluated using sMAPE.

Moreover, we want to add information to each entry about the previous pick-ups that happened in the cluster before the considered time bin. This will be useful to the prediction model to collect information about the dynamics of the pick-up trend for that region. Consequently, we added to each entry (which represent a time bin of a certain region) the previous $five$ pick-ups quantity of that region. Therefore, we are forced, when building the dataset, to not include in our new dataset time bins from t_0 to t_4, since they do not have a previous history of $five$ time bins. We will start, for each region, from time bin t_5

until t_n, where $n = 720$ for 30-days months and $n = 744$ for 31-days months.
The features that will compose our dataset and therefore the input to our prediction model will be:

- *pick-up cluster latitude*: This is the latitude coordinate of the center of the cluster to which the entry of the dataset belongs.
- *pick-up cluster longitude*: This is the longitude coordinate of the center of the cluster to which the entry of the dataset belongs.
- *time bin*: This is the hourly time span of the month to which our dataset belongs. It goes from 1 to 720 for 30-days months and from 1 to 744 for 31-days months.
- *day of the week*: We assign to each entry a number representing the day of the week to which it belongs. Monday will be 1 while Sunday will be 7. The other days of the week are the values in between.
- *previous five pick-ups*: Each entry has associated the number of pickups of the previous $five$ time bins for the region where the entry is located.

The *label data*, that is the data we want to predict, and it is the number of pickups for the present time bin.

Chapter 9

Powerful tools for demand forecasting

9.1 Statistical modeling vs. machine learning

As we have seen, the accuracy of the prediction is influenced by many parameters like weather, accidents, holydays and many others. Moreover, the city network is a set of several segments, and therefore, segments and regions can be influenced by the neighbor regions and these increase a lot the complexity of the study. Usually traffic forecasting is treated as a typical time series forecasting model, and the long temporal dependency of historical observations is hard to capture with traditional forecasting approaches, especially when modelling periods and trends [91].
Traffic flow forecasting is usally treated with both statistical and machine learning methods.
The correlations that exist in the successive observations of traffic state evolutions has made many researches to use time-series prediction models to solve traffic prediction problems. ARIMA (Autoregressive Integrated Move Average) has been widely used in traffic flow prediction. In [68] they present a model to predict the number of services that will emerge at a given taxi stand, that has been identified as an hot-spot for the arrival of taxis, which represent the 76% of the taxi demand of the city. Their algorithm solves the spatiotemporal distribution taxi-passenger demand using streaming data, that is, it can work in an online environment. Although statistical methods have been widely adopted, they cannot effectively extract inherent spatial and temporal dependencies and remain limited to a single road segment or single region scenarios. To predict city-wide traffic flow volume, a large number of independent models must be constructed. This limits the further application of statistical methods in the city-wide traffic flow prediction problem [91]. Statistical methods fail when dealing with complex and highly nonlinear data (curse dimensionality, that is when we analyze high-dimensional spaces, therefore with many features describing our data).
Neural networks (NN), have been widely applied to various transportation problems, partly because they are very generic, accurate and convenient mathematical models able to easily simulate numerical model components [50]. In trasportation research, Neural Networks have been mainly used as data analytic methods because of their ability to work with massive amounts of multi-dimensional data, their modeling flexibility, their learning and generalization ability, their adaptability and their good predictive ability. New neural networks software packages have simplified the development of extremely sophisticated NN models, while the corresponding statistical models would be almost im-

possible to develop, because of the complexity the algorithm can reach. Another problem of time series fitting models is that they have to be trained separately for each area, hence, the patterns learned in one area can not be used in other areas [97]. Some of the most used neural network algorithms for time series prediciton are *recurrent neural networks* (RNN) that have backward connections between hidden layers. The recurrent neural networks have internal memory which allows them to operate over sequential data effectively. Therefore, RNNs are one of the most popular models for dealing with sequential tasks such as handwriting recognition, language modeling and time series prediction. Although we could achieve good predictions with RNNs, there are some problems related to their use. Indeed, RNNs cannot train on time series with long time lags. Moreover, RNNs need to learn a time sequence on predetermined time lags. Lastly, they suffer of the gradient vanishing problem. Consequently, we will make use of LSTM (Long-Short Term Memory) recurrent neural networks.
Another class of machine learning algorithms that has gained attention for regression and classification is *gradient boosting*. Gradient boosting of regression trees produces competitive, highly robust, interpretable procedures for both regression and classification [38]. Later we will address specifically the use of LSTM recurrent neural networks and of gradient boosting algorithm for prediction.

9.2 Predictability

An interesting point, is the one assessed in [99] about predictability of a time serie. Given that public and private vehicles are the main transportation means in a city, we know that people use vehicles for commuting to and from work, for regular and irregular transfers. When public transportation vehicles are equipped with a GPS that gives information about their locations, we have a set of data about regular trips, that therefore will be predictable. Indeed, those vehicles have to follow fixed routes under specified schedules. On a similar way, also private vehicles give information about almost predictable routes, since it is probable that the vehicle owner is travelling to and from work with a certain level of periodicity. On the other hand, taxis serve the transportation need of an heterogeneous set of requests, that are driven by different needs. Some routes may be considered as predictable, if they happen often from and to frequent points of interest.
Given a predictive algorithm, considering both randomness and temporal correlation of the taxi demand sequence, what is the upper bound of the potential accuracy that a predictive algorithm can reach? The objective of [99] is to quantify the interplay between the regular and thus predictable and the random and thus unforeseeable human mobility. The maximum predictability is defined by the *entropy* of the taxi demand sequence, considering both the randomness and the temporal correlation of our data. Entropy is probably the most fundamental quantity capturing the degree of predictability characterizing a time series [89].
We aim to illustrate what past studies on predictability have proven, displaying their results. We will not directly study predictability for our case.
By measuring the human mobility it is possible to draw conlusions on the maximum predictability, which captures the temporal correlation of taxi demand. We know, as mentioned above and as shown in Figure 8.18, that great part of the prediction of the taxi demand depends on the pickup location. Therefore, the predictability of a region would be influenced by the location in the 2D-space as well. This means that different regions (or cluster) will have different predictabilities. In [99] they found that taxi demand in New York City has a strong temporal pattern.

As already said, entropy is a good measure to identify predictability of some area under study. Entropy can capture the temporal correlation of the taxi demand. In general, a lower entropy means higher predictability. They use three kinds of entropies:

1. *Random entropy*:
$$S^{(i)}_{random} = log_2 N^{(i)} \tag{9.1}$$

2. *Shannon entropy*:
$$S^{(i)}_{shannon} = -\sum_{t=1}^{N^{(i)}} p(d^{(i)}_t) log_2 p(d^{(i)}_t) \tag{9.2}$$

3. *Real entropy*:
$$S^{(i)}_{real} = -\sum_{S^{(i)}_n \subset D^{(i)}_n} P(S^{(i)}_n) log_2 |P(d^{(i)}_t)| \tag{9.3}$$

If we call Π the predictability and Π_{random}, $\Pi_{shannon}$ and Π_{real} the predictabilities related to the aforementioned entropies, it has been proven that $\Pi_{random} \leq \Pi_{shannon} \leq \Pi_{real}$ [89], therefore we will have that $\Pi_{max} = \Pi_{real}$. This means that the maximum predictability we were looking at, will be given by Π_{real} and will be related to S_{real}.
Small entropies are related to high predictability that is when we have high temporal patterns. Unfortunately, it is not possible to use a unique entropy for the entire dataset but we need to differentiate the entropy depending on the location of a subset of pick-ups. Therefore, we will observe that different regions will exhibit different maximum predictabilities. Finally, it is shown that residential and working areas have high predictability and thus exhibit strong temporal patterns. For other places such as hotels and transportation centers, the predictability is not high. The taxi demand for these regions is mainly dependent on the events happened during the day [99].
One very interesting point is the one addressed in [1]. Indeed, studying the number of pickups is not realistically describing the real behaviour of the demand for taxis. To really meet the request for taxis from customers we should be able to include in our prediction model also the requests that have not been picked up. These requests, together with the served requests, reflects the real number of people that were present at a certain spot or certain region waiting for a pickup. If they, either by seeing the mobile application was taking to long to assign a vehicle or by being proposed a too expensive route, have declined the taxi request, they will never be satisfied and they will never be considered into the predictive model. The only time we can trust the number of boardings into the cabs, is when taxis are in abundance with respect to the requests. In this case, the abundance of taxis might be reallocated to areas in need of taxis. The unmet demand is estimated via heuristic, accounting for the observed taxi demand, the imbalances between supply and demand and other parameters. Imbalance is measured on the states of the taxis which tells what is the amount of time vehicles are seen in a *free* state in the different areas of the city. High unmet demand areas will correspond to very little *free* status time of vehicles in these areas. Indeed, when an area is busy of taxi requests, the *free* status of the vehicle is immediately consumed by another request boarding the taxi.

Chapter 10

Demand forecasting models

10.1 Recurrent neural networks

A recurrent neural network (RNN) is an extension of a conventional feedforward neural network, which is able to handle variable-length sequence inputs. They belong to a class of artificial neural networks where connections between units form a directed cycle. This allows the network to have a loop where the signal is sent back to itself. As a result, it creates an internal state of the network which allows the network to exhibit dynamic temporal behaviour [80]. Because of these capabilities, RNNs are mainly used for dynamic information processing and sequential tasks like time series prediction, language modelling. However, because of the way RNNs are designed, the information about *long-term dependencies* is lost. They have a limited memory capacity and therefore they are not able to completely capture the periodicity of signals. In Figure 10.1, we can see a simple example showing a sequence of inputs, each characterized by some features, which are reflected into the input layer of the network. Then, the general architecture is of the same kind of any other artificial neural network.

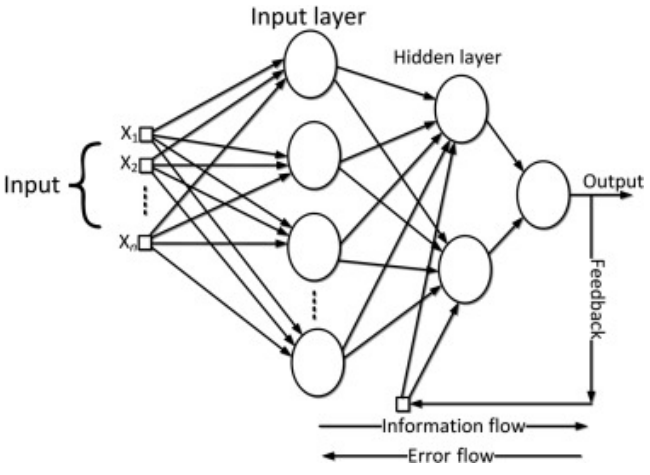


Figure 10.1: This is the general architecture of a recurrent neural network. This architecture is composed of an input layers with as many node as the number of features describing our input elements, an hidden layer whose number of neurons is arbitrary, and a dense output layer composed of just one neuron. The inputs x are represented to show how multiple time instances of the input sequence can be passed as inputs.

Figure 10.2 shows a typical RNN to illustrate the information about the hidden state h

flowing within the neuron at different time instants. From this figure it should be clear to understand how the sequential information is managed by the network.

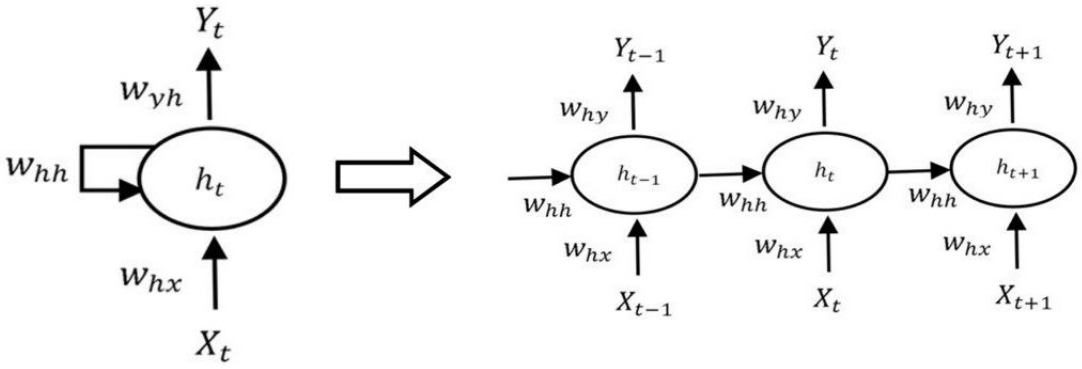


Figure 10.2: On the left, a recurrent neural network showing the feedback action on the weights of the hidden state. On the right, again a recurrent neural network in an unrolled version, showing clearly the flow of the weight matrix information.

Given an input of time series $X = \{x_1, x_2, \ldots, x_t\}$, the RNN computes the hidden state sequence $H = \{h_1, h_2, \ldots, h_t\}$ and output sequence $Y = \{y_1, y_2, \ldots, y_t\}$ iteratively, according to the equations [94]:

$$\begin{aligned} h_t &= f(W_{hx}x_t + W_{hh}h_{t-1} + b_h) \\ y_t &= g(W_{yh}h_t + b_y) \end{aligned} \tag{10.1}$$

where W_{hx}, W_{hh} and W_{yh} denote the input-to-hidden weight matrix, the hidden-to-hidden weight matrix and the hidden to output weight matrix respectively. The vector b_h and b_y are the bias of the hidden layer and the output layer respectively. Usually, since the bias is not affected by any input of the network it can be considered as a generalization unit that assists the net in outputting the more common results [37]. Biases assist the networks in making more general predictions, to avoid overfitting of the training data. On the other hand, if we have a bias that is too large, we will underfit data, which means that our model is not able to fit the given data. Functions $f()$ and $g()$ are the activation functions of the hidden layer and the output layer respectively. Activation functions are functions used in neural networks to compute the weighted sum of input and biases. Activation functions can be either linear or non-linear depending on the functions they represent, and are used to control the outputs of our neural network. A special property of the non-linear activation functions is that they are differentiable, otherwise they cannot work during backpropagation, that is how weights at the nodes are updated during learning.
In order to deal with the short memory problem, a more sophisticated neural network called Long Short-Term Memory (LSTM) recurrent neural network was developed.

10.2 Long short-term memory recurrent neural networks

Lond short-term memory neural network is proposed here to predict the number of pick-ups in different areas of the city of New York City. Long short-term memory neural

networks can capture traffic dynamic in an effective manner. The LSTM neural network overcomes the issue of backpropagated error decay through memory blocks, so that it can capture and model long-term dependencies and determine the optimal time lag. Indeed, in conventional recurrent neural networks, error signals, by "flowing backwards in time", tend to either blow up or vanish. Specifically, an error that blows up during the backpropagation for the learning process may lead to oscillating weights. On the other hand, in the second case where the error approximates to zero because of the vanishing gradient problem, the algorithm may take a prohibitive time to bridge all the time lags, or it is not even able to learn and update the weights at all. The LSTM is designed to fix these error back-flow problems. This is achieved by an efficient, gradient-based algorithm for an architecture enforcing *constant* error flow through internal states of special units [45]. By constant error we mean that it is neither exploding to huge values nor vanishing towards zero.
Therefore, what LSTMs want to achieve is to undergo a *constant error backpropagation.* In the case where the error is exploding, we have that it can grow exponentially with q which measures the step of delay of the backpropagation process. If this happens, the error blows up and conflicting error signals arriving at a certain unit of the network can lead to oscillating weights and unstable learning. On the other hand, when the error is vanishing, it happens that the error decreases exponentially with q. If this happens, the error is close to zero and nothing can be learned in acceptable time.
The *naive approach to constant error flow* is introducing a basic case where we are able to achieve a constant error flow though a single unit j of the network. From [45] we have that the constant error flow for unit j, at time t, is given by:

$$\vartheta_j(t) = f_j^{'}(net_j(t))\vartheta_j(t+1)w_{jj} \tag{10.2}$$

It is trivial to notice that to have constant error flow we need to impose:

$$f_j^{'}(net_j(t))w_{jj} = 1 \tag{10.3}$$

By integrating 10.3 we obtain $f_j(net_j(t)) = \frac{net_j(t)}{w_{jj}}$ for arbitrary $net_j(t)$. This means that, having a constant derivative, f_j has to be linear, and unit j's activation function has to remain constant as:

$$y_j(t+1) = f_j(net_j(t+1)) = f_j(w_{jj}y^j(t)) = y^j(t) \tag{10.4}$$

This implies that the identity function $f_j : f_j(x) = x, \forall x$ has to be always verified. Moreover $w_{jj} = 1$. This is the main feature of LSTM neural networks, and it comes under the name of *constant error carrousel* (CEC). However, unit j is not connected only to itself, but it is connected to other units. This means that we will have to manage problems related to connections between different units also, like in every other gradient-based approach. Paricularly, we can distinguish the input connection of the unit and the output connection:

1. *Input weight conflict*: we focus on the additional input coming from a single unit i. It may happen that at the same time, the weight w_{ji} associated to input in unit j from unit i, needs to have conflicting values (0 or 1) depending on the instance of the input received. In fact, it may be that for some input of the model, a certain value of w_{ij} (assume 1) is required to decrease the total error while for other inputs of the sequence, it is needed that w_{ij} has another value (assume 0). Therefore, the same weight w_{ij} has to be used to store some inputs and neglect others. Obviously, this creates a contrast.

2. *Output weight conflict*: It may happen that unit j has to provide unit k with some information about its state, and to this aim, a weight w_{kj} is used. During training it could be that weight w_{kj} is used to retrieve content of unit j while other times it has to prevent j from disturbing unit k. Hence, it would generate conflicts between needs for the use of w_{kj}. Sometimes, w_{kj} will provide information about the state stored in j and other times it will be used to protect unit k from the state of j.

Therefore, when using LSTM networks with long lags, the effect of conflicting weight values is particularly strong and can badly affect the performances of the model.

Looking now at the architecture of an LSTM neural network, we say that it is composed of one input layer, one recurrent hidden layer and one output layer. The basic unit of the hidden layer is a memory block, similarly to what we have seen for RNNs. The memory blocks contains memory cells with self-connections memorizing the temporal state, and a pair of adaptive, multiplicative gating units to contro information flow in the block [65]. The self-connection is what we called the constant error carrousel (CEC) and it describes the connection between unit j and itself through a linear relationship.
Moreover, as we just said, an LSTM comes also with a miltiplicative *input gate*, to protect the memory contents of a unit j from information of irrelevant inputs, and a multiplicative *output gate* that is used to protect other units k from possible perturbations by irrelevant contents that may be stored in unit j.
Additionally, more recent LSTM implementations have a *forget gate* that is introduced to throw some of the contents of the cell state, in order to clean the state of some undesired or useless information. The forget layer will operate on the cell state feedback, depending on the information of the new incoming input and of the hidden state of the cell.
We will present the case in presence of the forget gate, since it is the closest to the most modern implementations of the memory cells of LSTMs. In Figure 10.3 we can see an example of the memory cell of an LSTM NN in an unrolled version. This view of the memory cell allows us to observe the cell state moving through time and allows us to see the internal structure of the cell.

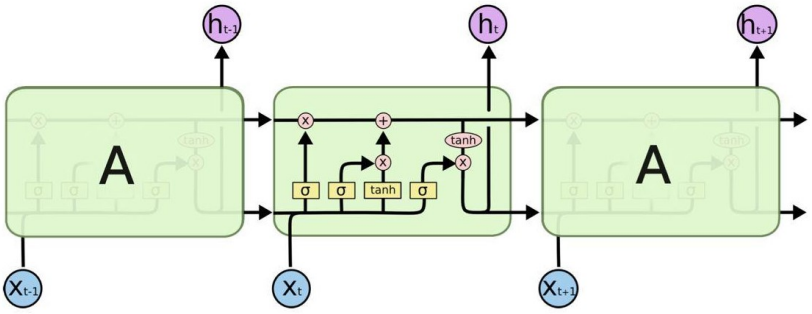


Figure 10.3: The repeating module of an LSTM recurrent neural network. It is an unrolled version of its memory cell, so that we can follow the cell state information flowing through different time instants.

The key feature of the LSTM is the cell state, the horizontal line running through the top of the diagram. Gates like the *input gate*, *output gate* and *forget gate* are a way to optionally let information through. They usually are composed of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between *zero* and *one*, describing how much of each component should be let through. Figure 10.4 shows an example of gate.

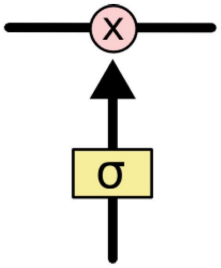


Figure 10.4: Example of a gate with a sigmoid neural net layer and a pointwise multiplication operation. It is used to implement the *input gate*, *output gate* and *forget gate*.

where the outputs from the three gates are respectively represented as i_t (for the *input gate*), o_t (for the *output gate*) and f_t (for the *forget gate*). The sigmoid function is represented by $\sigma()$ that denotes the standard logistics sigmoid function defined as [65]:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{10.5}$$

10.2.1 Step-by-step LSTM memory cell process

Referring to Figure 10.3, the first step in the LSTM is to decide what information we're going to toss from the cell state. This decision is made by a *forget layer* of the kind of Figure 10.4. The exogenous input x_t is concatenated with cell hidden state at previous time step h_{t-1}, and the forget layer assigns to each element a 0 or 1 depending on the values of the cell states C_{t-1}. The forget vector f_t, that is composed of *zeros* and *ones*, will moltiply element of the cell state vector C_{t-1} to decide what to keep from the past state and what to throw. This is computed as:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{10.6}$$

Second, we are interested in knowing what new information we are going to store in the state cell of the unit. First, the already mentioned *input layer* will decide which value of the cell state we will update using a sigmoid activation function that produces a vector i_t of *zeros* and *ones*. This step is the one mentioned before, when speaking of the conflict on the weight w_{ji} at the input. Then, a *tanh* activation function (as the one of Equation 10.8) will create a vector of new candidate values $\tilde{C}_t$ that will later be added to the cell state in order to update it. i_t and $\tilde{C}_t$ are computed as:

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= tanh(W_{\tilde{C}} \cdot [h_{t-1}, x_t] + b_{\tilde{C}}) \end{aligned} \tag{10.7}$$

where $tanh()$ is the hyperbolic tangent activation function and it is defined as:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{10.8}$$

After obtaining i_t and $\tilde{C}_t$ vectors, we can procede updating the cell state by first computing the element-wise product $i_t \odot \tilde{C}_t$ and then adding the result to the cell state C_t. C_{t-1} itself has already been cleaned by the forget layer as $C_t = f_t \odot C_{t-1}$. Therefore, we will have:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{10.9}$$

The cell state C_t is the state that we will feedback again to the memory cell for instant $t+1$. From C_t we will obtain C_{t+1} with analogous procedure to the one just seen. Finally, we need to decide what we are going to output from the memory cell and what will be the hidden state that we will input to the next time instant. We can compute the vector o_t which represents what our *output gate* is deciding to omit of our cell state to not perturbate the behavior of the units connected to the output of the current unit. Hence, the hidden state will depend on the vector o_t. We will have:

$$\begin{aligned} o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\ h_t &= o_t \odot tanh(C_t) \end{aligned} \tag{10.10}$$

At the end, we will obtain the output of our unit at time t as:

$$y_t = W_y \cdot h_t + b_y \tag{10.11}$$

10.2.2 LSTM implementation with Keras API

For the implemetation of the LSTM recurrent neural network we used Keras API [25] [19]. It is an open-source deep neural network library written in Python. Keras offers simple APIs that are running on top of Tensorflow, R, Theano and other machine learning and neural network software libraries. These APIs make it easy to work with and implement the most sofisticated neural-network building blocks such as layers, obejctives, activation functions and optimizers. Keras offers support also for recurrent neural networks. In particular, it offers the implementation of LSTM recurrent neural networks as one of the classes of its libraries. Specifically, when using Keras, if we want to build the architecture of our model (which has to be defined prior to the training phase) we need to create an object of class `tf.keras.model.Sequential`. By passing to the *Sequential* constructor a list of layers we will be able to build our neural network. Therefore, we will use `tf.keras.layers.LSTM` for our purposes, which allows to build an LSTM layer. Through the use of different parameters of the LSTM class we can customize our memory units in the LSTM layer. Default values have been chosen, which coincide with the LSTM architecture previously described. In particular, we will have parameters `activation = 'tanh'` as activation function of the gate layers and `recurrent_activation = 'sigmoid'` as activation function for the candidate hidden state and output hidden state of the unit.
This LSTM layer will choose different implementations (cuDNN-based or pure-TensorFlow) to maximize the performances of the model architecture and of the training. If a GPU is availbale and all the arguments of the layer meet the requirements of the cuDNN kernel, the layer will use a fast cuDNN implementation. Unfortunately, our model has been developed on a personal PC that makes use of a CPU Intel Core i7-6560U (2.20GHz - 2.21 GHz), 16 GB LPDDR3 RAM and an Intel Iris HD Graphics 540 graphics processing unit which is not capable of running GPU supported machine learning software by TensorFlow. Consequently, training the model was taking a lot of time and the development and tuning of the model parameters required to train it several times.
Our best performing model has a single layer made of 100 memory cells and a final dense layer, which is a regular densely-connected neural network layer. The dense layer is made of one single neuron and has linear activation function. It is implemented by providing the model constructor argument `tf.keras.layers.Dense` where we can specify as argument of the layer the number of neurons composing it. To capture the one-day periodicity of the traffic demand data, we want to input 24 lagging timesteps to our model. Hence, the input will be represented by the sequence $X = \{x_{t-1}, x_{t-2}, \ldots, x_{t-24}\}$. To this aim, we

need to reshape the input in order to specify the number of features of the input and the time length of the input sequence provided.
Then, through method `compile` we can specify the *optimizer*, the *loss function* to be used and the *metrics* to be evaluated by the model during training and testing. *Adam* optimizer has been chosen, given its wide use in recent NN studies and examples found in the literature to implement the neural network learning phase. Adam is a computationally efficient algorithm for gradient-based optimization of stochastic objective functions. It is designed to work with large datasets and/or high-dimensional parameter spaces given its little memory requirements [53].
When dealing with regression problems, which have to predict real-valued quantities, it is suggested to use one of the default loss functions such as *mean squared error* (MSE) and *mean absolute error* (MAE). Many other loss functions can be used, but we will limit our study on testing these two loss functions to find the most suitable for our purposes.
As we can see from 10.12 *MSE* is more sensitive to bigger errors, because of the squared term. This metric will assign more weight to big errors such as outliers in the dataset.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{10.12}$$

where y_i is the i-th term of the real data of our test set and $\hat{y}_i$ is the i-th prediction made by our model from the test input. On the other hand, MAE is defined as:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{10.13}$$

it is the average absolute difference between real values and predictions. MAE is sensible in the same way to small and large errors, since it is proportional to the absolute value of the error, as opposed to MSE, which involves the square of the error. Both loss functions summarize performances in ways that disregard the direction of *over-prediction* or *under-prediction*. After having tested our model with both MSE and MAE loss functions, we have concluded that we can obtain far better learning curves by using the mean absolute error (MAE). Therefore, further implementations and changes to the model have always involved the use of MAE as a loss function for our model.

10.2.3 Model training and prediction

The next step is to train the model. Keras has built-in methods that are optimally implemented to ease the learning process of the model. Hence, we will use `fit` method wich will ask for some simple arguments in order to then continue with the training phase. Prior to this step, we need to divide our dataset into *training* set, *validation* set and *test* set. The training set is the one on which the model is trained. The *test set* is instead a set that is used to evaluate the performances of our model. It has to be a portion of data that our model has never seen before, in order to evaluate the generalization capabilities of our NN model. Our dataset is composed of yellow taxi ride data of April 2015, May 2015, June 2015, April 2016, May 2016 and June 2016. Each of the monthly data has been split as: 70% of the data as *training* set and 30% as *validation* set. We split each of the 30 spatial clusters taken singularly in order to have information about each cluster into the *training* set and also into the *validation* set. Otherwise, the model might have no example for some of the 30 clusters, and this would lead to failures in the prediction when evaluating the demand for these clusters.

As far as the *test* set is concerned, we decided to use the training set from June 2015, that was originally supposed to be used for training. Therefore, this set of data will not be provided to the model during the training phase.
Finally, we trained the model for 100 epochs, looking at the training curve after each training instance was completed. The training of the model required long time, because of the limited capacity of the development PC with which the model has been implemented. The best performances, as we anticipated, have been achieved using a single LSTM layer with 100 neurons, a dropout layer (implemented by `tf.keras.layer.Dropout`) and a dense layer of only 1 neuron. The dropout layer is used to drop some of its inputs, in order to help the model avoid overfitting. It is represented by a float that goes from 0 to 1, which represents the fraction of inputs that are dropped to 0. We choose to apply a dropout of 0.3. Figure 10.5 is the the training curve for 100 epochs training, where the loss is evaluated with MAE using Equation 10.13.

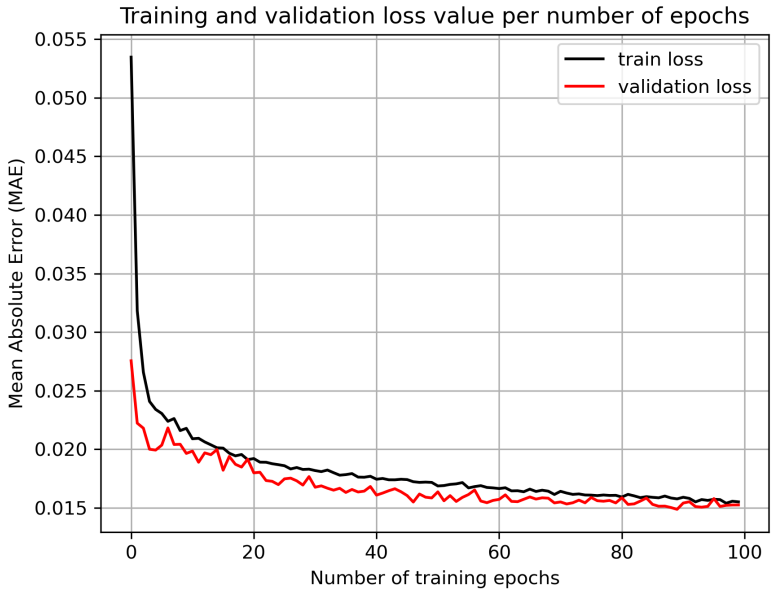


Figure 10.5: Training curve for a 100 epochs training for an LSTM recurrent neural network with one hidden layer of 100 neurons and dropout of 0.3. The loss is represented by the *mean absolute error* (MAE) between the true data y and the predictions $\hat{y}$.

Now we compare part of the prediction on the test dataset with the corresponding real values, just with the purpose of visualize the prediction accuracy. Figure 10.6 is representing the prediction versus the real data on the test set for cluster 2 of our regions.

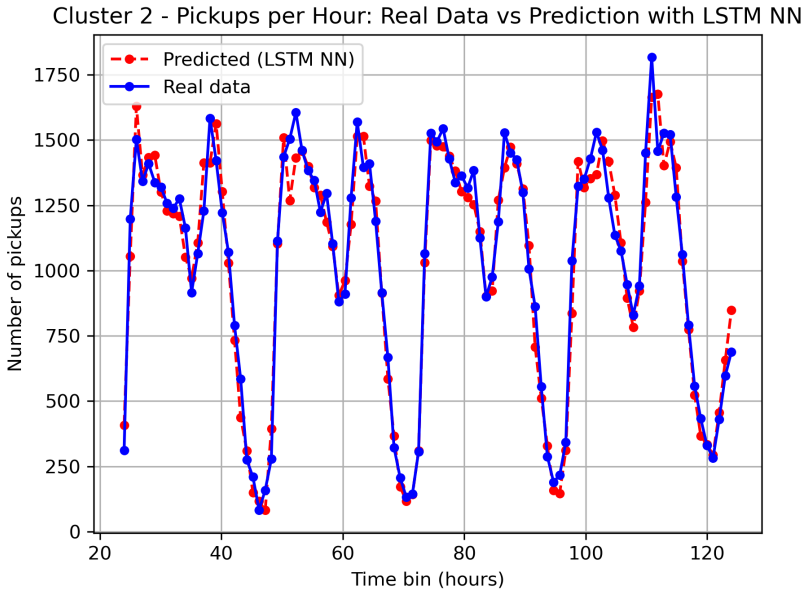


Figure 10.6: Comparison between real data (orange) and predicted data (blue) from the Keras LSTM recurrent neural nework.

10.3 Gradient boosting

Tree boosting is a highly effective and widely used machine learning method. It is a technique for machine learning regression and classification problems, that produces a predictive model in the shape of an ensamble of weak predictive models, that are simple functions. The main idea of boosting is to merge this set of weak learners into a strong one, in an iterative fashion. Using a training sample (x_i, y_i) of known (x, y) vlaues, the goal is to obtain an estimate or approximation $\hat{F}(x)$ of the function $F^*(x)$ mapping x to y, that minimizes the *expected value* of some specified loss function (or objective function) $L(y, F(x))$ over the joint distribution of all (x, y) values [38]. Frequently employed loss functions $L(y, F)$ include squared-error and absolute error for regression and negative binomial log-likelyhood for classification.

$$F^*(x) = \arg\min_{F(x)} E_{y,x} L(y, F(x)) \tag{10.14}$$

Function estimation/approximation is viewed from the perspective of numerical optimization in function space, rather than parameter space. Having functions as parameters, the model cannot be optimized using traditional optimization methods. Instead the model is trained in an additive manner. A general gradient descent boosting paradigm is developed for additive expansions based on any fitting criterion. For instance, boosting approximates function $F^*(x)$ by an additive expansion of the form:

$$F(x) = \sum_{k=0}^{K} \beta_k f(x, a_k) \tag{10.15}$$

where the functions $f(x, a)$ are the base learners and are chosen to be simple functions of x with parameters $a = \{a_1, \cdots, a_n\}$. Coefficients β and a are fit to the data to minimize the objective of Equation 10.14. Because greedy strategies choose one weak model at a

time that most improves our model according to our objective in 10.14, we can say that boosting models use this recursive formulation:

$$F_k(x) = F_{k-1}(x) + \beta f_k(x, a) \tag{10.16}$$

The algorithm can choose to stop adding weak models when $\hat{y} = \hat{F}(x)$ performances are good enough or when $f_m(x)$ does not add anything.
Special enhancement are derived for the particular case where the individual additive components are regression trees [38]. For a given data set with n instances and m features defined as [23]:

$$\mathcal{D} = \{(x_i, y_i)\}(|\mathcal{D}| = n, x_i \in \mathbb{R}^m, u_i \in \mathbb{R}) \tag{10.17}$$

a tree ensamble model uses K additive functions to predict the output.

$$\hat{y}_i = \hat{F}(x_i) = f_1(x_i) + \cdots + f_K(x) = \sum_{k=1}^{K} f_k(x_i),\ f_k \in \mathcal{F} \tag{10.18}$$

where $\mathcal{F} = \{f(x) = w_{q(x)}\}$ $(q : \mathbb{R}^m \to T, w \in \mathbb{R}^T)$ is the space of regression trees. Here q represents the structure of each tree that maps an instance to the corresponding leaf index. Being q the structure of the tree, it is describing a set of rules in the specific tree. T is the number of leaves in the tree. Each function f_k, the base learner, corresponds to an independent tree structure q and leaf weights w.
For fixed tree structures $q(x)$ we compute the optimal weights of the single leafs. After having computed the weights of the leafs, we can calculate the optimal value of our loss function. In this way, we will be scoring the tree structures q of our model. One of the solutions is to use the so called *exact greedy algorithm* which enumerates all the possible splits in each variable. The best split then is chosen according to the loss function. First the algorithm sorts the data according to feature values and visit the data in sorted order to find the gradient statistics. Usually, we need to evaluate, through the objective function, the two branches in which a node has been split. Each split identify a new tree sctructure, that is q. Therefore, doing the analysis for all the tree structures is obviously computationally unfeasible, given the number of instances of our dataset and the number of features that describe our data.
This is why approximate algorithms have been developed in order to efficiently evaluate the tree structure q that will be needed to build the ensamble model. Some of these algorithms find candidate splitting points according to the percentiles of feature distribution.

10.4 XGBoost algorithm for gradient boosting

XGBoost [23] is an open-source software libray which provides a gradient boosting framework for C++, Python, Java and other programming languages. It aims to a scalable, portable and distributed gradient boosting library. The library is focused on computational speed and model performances. Some of the performances of XGBoost on building gradient boosting models are: *scalability*, *regularization*, *parallel processing*, *handling missing values and sparsity*, *non-greedy tree pruning*. Scalability comes from the many improvements that have been developed to manage the tree construction and the gradient boosting.
Split finding: When dealing with large datasets, the search for the split requires a lot of computation being it a non-trivial operation. Most of the existing techniques rely on

confining the data on subsets that are randomly picked. Instead XGBoost has implemented a *distributed weighted quantile sketch* algorithm to find efficiently the splitting points from the quantile analysis of the feature distribution. Additionally, XGBoost can handle sparsity patterns in a unified way. Indeed, it is common that data are sparse or some of the data are missing. XGBoost has predetermined paths designed to handle such situations so that to visit only non-missing split nodes.
Regularization: It helps to smooth the final learnt weights to avoid over-fitting. The regularized ojective will tend to select a model employing simple and predictive functions.

$$L(y, F(x)) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \quad \text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda ||w||^2 \tag{10.19}$$

The regularization term helps in keeping low the complexity of the regression tree functions, by penalizing the weights, in order to help avoid overfitting of the test data.
Other techniques used to avoid overfitting are shrinkage and column susbampling: Shrinkage scales newly added weights by a factor η after each step of the tree boosting. It reduces the influence of each three and leaves for future trees to improve the model. Individual observations are penalized by the presence of paramete η. Moreover, also column (or feature) subsampling is used in order to reduce the possibility of overfitting. It means that each individual tree will be built with only a percentage of the independent variables of our input chosen randomly. Also, a large motivation in restricting th enumber of predictors available to each learner is to encourage variance between trees, even if it is not a huge concern in boosting. Subsampling will allow the model to converge faster with boosting.
Parallel processing: XGBoost implements functionalities to improve the memory usage of the machine by enclosing bunch of data in blocks with each column sorted by the corresponding feature value. Most of the processes in gradient boosting, like the approximate algorithms for split search, are optimized by the use of blocks, allowing for easy handling and scan of the dataset. In addition, collecting the statistics for each column, can be parallelized giving a parallel algorithm for split finding. These methodologies solve the big problem of split search, that otherwise would be computationally expensive.
To use the XGBoost library for our prediction problem, we can rely on the scikit-learn API for XGBoost random forest regression. To use XGBoost on Python we do not need to do particular manipulation of the data except separating the label data from the predictors of our dataset. First we will create an object of class `xgb` with the constructor `XGBRegressor` since the goal of our model will be to predict an outcome from the inputs. By now, though, the model is only initialized and we need to do further considerations in order to add details to the model architecture. In fact, we need to find the hyperparameters who describe our gradient boosting model. The ones we are interested in are *maximum depth* of the trees and *number of trees*. The maximum depth of the model is related to the number of layers of the trees in our model. The deeper are the trees, the more complex our model and the more accurate the predictions will be. However, we need to consider that having higher complexity in our model will make it more computationally expensive to build. Also, if our predictions are too accurate with respect to the training set, then it is likely that our model has only learned to perfectly mimic the training data, while it will not be able to accurately predict new data. A similar discussion applies to the choice of the number of trees. We know that to build the ensamble, we need to generate many trees so that to be able to build an accurate prediction model. However, also in this case, we must be careful on the choice of the number of trees, because an high number of trees will make our model to overfit, which means we are able to almost perfectly fit the

train data but the model is not able to generalize to unseen data.
Once we have defined the set of possible maximum depth and number of trees (by using two distinct vectors), we need to choose the best ones with respect to our data. In order to tune the hyper-parameters from the possible choices provided in the vectors, we would need to do a fine search of these parameters based on our training data and some evaluation metric. To this aim we use the scikit-learn class `GridSearchCV`. A scoring function is used in order to evaluate the prediction of the candidate models on the test set. In fact, when comparing multiple algorithms with different hyper-parameters, we need to assign a score to each model in order to choose the best one based on that score. To this aim, `GridSearchCV` ranks all the algorithms and tells which is the best depending on the assigned score. Scikit-learn will rank with highest score the model with the highest score on the evaluation metric we have provided as a parameter. Since we use typical error metrics such as MAE, scikit-learn will rank the model with the highest MAE as the best model, but this is actually the opposite of what MAE is intended for. This is why we will use a *negative mean absolute error* which will always be a negative quantity. Therefore, the closer to zero the score, the better the model will be. Moreover, we can specify parameter *cv*, which determines the cross-validation strategy for our exhaustive search, specifically it tells what is the value of K for K-Fold Cross-Validation. This means that the training data will be divided into K subsets. Each time the grid search will tune our parameters using $K-1$ subsets as training set and one subset as the test set. The grid search will find a set of hyper-parameters that yields an optimal model which minimizes a predefined loss function. With method `fit` of the object of class `GridSearchCV` we will be able to tune the hyper-parameters following the instructions we provided. The optimal maximum depth for our model is 7, while the optimal number of regression trees is 20.

10.4.1 Model training and prediction

After having found the optimal value of the hyper-parameters, we can build our gradient boosting model for regression by calling again the constructor `XGBRegressor`, this time specifying the optimal values of the maximum depth allowed for each tree and the maximum number of regression trees to build the ensamble model. Then, we can train our model with the train data using method `fit`, in order to find the ensamble model which consitute the gradient boosting estimator. To evaluate our model, we use again a regional *symmetric mean absolute percentage error* (Equation 10.21) on a test set (month of June 2015) that is a portion of data that our model has never seen. In Figure 10.7 we can see a comparison between the pickups occured in cluster 2 of our regions of New York and the predicted number of pickups for the same region using the gradient boosting model.

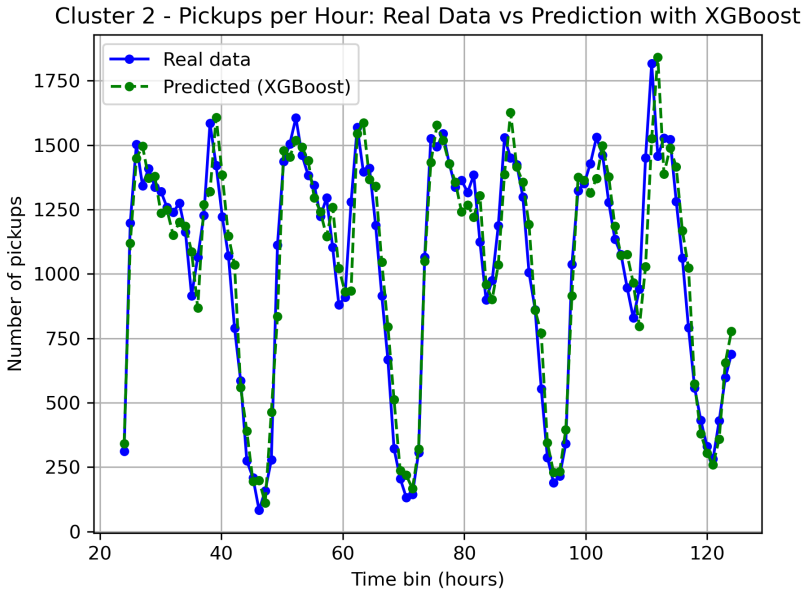


Figure 10.7: Comparison between real data (blue) and predicted data (green) using a gradient boosting model with XGBoost algorithm.

10.5 Baseline prediction model

In order to evaluate the performances of the LSTM model and the gradient boosting model we need to design some simple prediction model to compare with. Our baseline model consists of a naïve forecasting approach. This estimating technique uses the last period's number of pickups as the current period's forecast, without adjusting them or attempting to establish causal factors. The naïve forecasting model will be only used as a comparison with the other more complex models. The model is forecasting the output according to the simple relation:

$$\hat{y}_t = y_{t-1} \tag{10.20}$$

The same test dataset of the other models has been used in order to directly compare the performances and try to evaluate if the LSTM and XGBoost models are able to beat this very simple forecasting model.

10.6 Comparison between the prediction models

10.6.1 Metrics for the evaluation

Finally, as metrics for the evaluation of our model, we have chosen to use the *symmetric mean absolute percentage error* (sMAPE) which is widely used when evaluating performance of time series forecasting. Its advantage is that it avoids the problem of giving different error values when the forecast is higher or lower than the actual value differently from MAPE (mean absolute percentage error) [44]. When evaluating the sMAPE for our test set, we will distinguish between different spatial regions of the map. The sMAPE is defined as:

$$sMAPE_i = \frac{1}{n}\sum_{t=1}^{n}\frac{|y_{i,t} - \hat{y}_{i,t}|}{y_{i,t} + \hat{y}_{i,t} + c} \quad \text{for } i = 1, \ldots, N_C \tag{10.21}$$

where n is the number of time samples for each region, N_C is the number of regions in our map and c is a small constant that is used to avoid that the error can take very large values if the denominator of the t-th error term is zero.
To compute the sMAPE for the entire test dataset, we need to average the N_C sMAPE we have computed as:

$$sMAPE = \frac{1}{N_C} \sum_{i=1}^{N_C} sMAPE_i \tag{10.22}$$

sMAPE is a relative (computed as percentage) error. This means that the size of our problem (the absolute number of pickups per region) do not influence the size of the error. Conversely, error metrics such as MAE, MSE, or RMSE (*root mean squared error*) depend on the number of pickups of the problem instance. We think that using a relative error is more significant when evaluating the prediciton accuracy of our model. It would be different if the algorithm was designed with the goal of having an absolute limit on the maximum allowed error. We will not use this approach because this is very application specific and depends on the particular use of the algorithm.

To evaluate and compare the different models we have also used the RMSE, that is the *root mean squared error*, although we do not think it is representative of the accuracy of the prediction. In fact, it completely depends on the size of the problem. This means that the RMSE will depend on the absolute number of pickup. If instead of New York City we had used another city in which the hourly number of pickups is not in the order of hundreds but in the order of dozens, then we would have ended up, with considerably different RMSE, with no clue on what is the best of the models. Moreover, RMSE is difficult to use to compare the performances of the different regions. Indeed, we would need some clues on the average absolute number of pickups for that region. However, RMSE is giving us a measure of what is the size of the missed requests on the prediction. Therefore, we can have an idea of the order of the missing demand in the case of both high and low sMAPE. In fact, we are more interested in being able to cover the areas with the highest densities of taxi demand. This means that we might consider less important a huge sMAPE error in a suburban region than a big sMAPE in one of the main city areas. In the case of bad prediction and big sMAPE, we can rely on the RMSE to have an idea of the size of the missing demand. The RMSE is defined as:

$$RMSE_i = \sqrt{\frac{1}{n} \sum_{t=1}^{n} (y_{i,t} - \hat{y}_{i,t})^2} \quad \text{for } i = 1, \ldots, N_C \tag{10.23}$$

	Regional sMAPE [%]			Regional RMSE		
Region	Naïve	LSTM RNN	XGBoost	Naïve	LSTM RNN	XGBoost
1	18.408	49.517⋆	15.806	109.413	63.333	82.517
2	12.288	4.898	7.210	229.650	104.479	136.773
3	13.543	4.848	7.135	180.274	52.621	116.936
4	16.234	23.912⋆	16.075	109.870	108.055	130.926
5	12.349	4.159	7.812	239.176	90.100	146.749
6	14.805	5.425	8.273	236.525	103.394	162.224
7	13.694	4.481	7.281	143.967	52.892	93.791
8	12.544	9.228	10.649	72.950	45.336	62.761
9	16.976	23.130⋆	14.265	44.084	38.500	45.963
10	13.255	4.646	7.601	197.480	92.659	143.110
11	11.968	3.919	6.175	239.927	103.007	147.582
12	14.503	6.280	8.269	166.951	69.969	124.086
13	11.158	4.331	7.042	185.448	78.471	122.729
14	13.244	12.648	11.789	74.853	27.041	52.190
15	11.968	6.072	8.104	87.717	44.512	59.068
16	17.767	233.014⋆	19.670	13.621	19.411	21.515
17	13.988	3.993	7.691	247.615	91.752	140.665
18	15.671	9.543	11.152	90.206	44.067	69.361
19	12.636	4.257	7.380	117.878	47.972	75.765
20	11.352	3.838	6.895	181.439	75.391	121.343
21	15.856	6.355	7.824	178.289	66.532	107.148
22	11.243	12.563	10.397	49.192	34.955	44.573
23	12.721	5.335	7.212	124.204	63.534	92.096
24	10.316	4.194	6.661	188.764	96.010	135.138
25	10.138	4.148	6.729	242.641	122.963	174.099
26	12.645	4.732	6.569	188.143	62.419	116.003
27	11.683	4.214	7.472	192.651	81.614	135.053
28	15.836	238.588⋆	14.417	53.486	12.564	43.835
29	18.796	385.159⋆	17.459	20.700	22.598	17.913
30	14.271	5.602	7.993	180.245	74.059	113.891
Average	13.729	36.434	9.634	148.3310667	66.340	101.193

Table 10.1: sMAPE (symmetric mean absolute percentage error) and RMSE (root mean squared error) for testing of the naïve approach, LSTM recurrent neural network and gradient boosting models. (⋆) sMAPE values are showing anomalous behaviour with respect to the other metric of the same model for other clusters.

10.6.2 Performance comparison

As we can see, LSTM model outperforms both the naïve and the XGBoost models in many of the 30 regions. Unfortunately, if we compute the average sMAPE between all the regions, we find out that LSTM is making poor predictions with respect to the other two models. However, if we have a look at all the i-th individual sMAPE for the different regions, we can notice that for some clusters (specifically the ones noted with (⋆), which are the regions identified by numbers: 1, 4, 9, 16, 28, 29), the prediction performance given by sMAPE is very poor. In Figures 10.8 - 10.11 we can see the comparison between the predicted and real data for clusters 4, 16, 28, 29. It is clear how predictions are

not accurate at all for areas of clusters 16, 28, 29. They present irregular demand and therefore the algorithm is not able to generalize the model learning. Instead, the training sets for these area have pretty accurate predicitons, which means the model has overfitted the training data. For what concerns the airpot area (JFK Airport, cluster 4) in Figure 10.8, the predictions can be considered more accurate. We will see that this is related to geographical reasons that highly influence the demand trend.

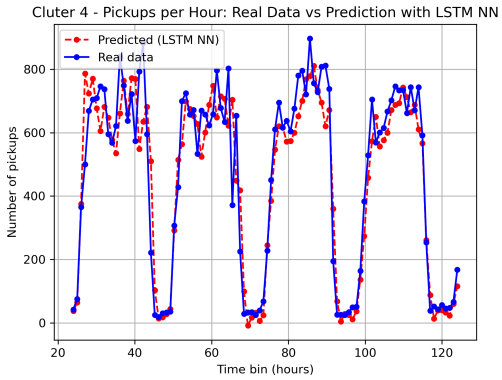


Figure 10.8: Comparison between prediction and real data for cluster 4.

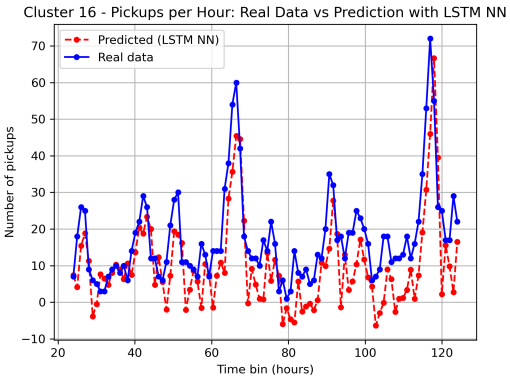


Figure 10.9: Comparison between prediction and real data for cluster 16.

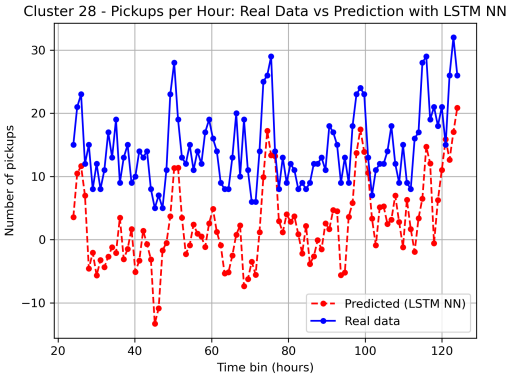


Figure 10.10: Comparison between prediction and real data for cluster 28.

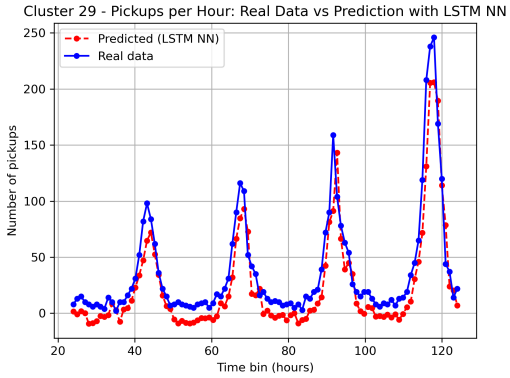


Figure 10.11: Comparison between prediction and real data for cluster 29.

Nevertheless, we do not want to give up the entire LSTM prediction model because of these poor local prediction indicators. We rather want to understand why these clusters are not prone to be modeled by the LSTM recurrent neural network. We also want to emphasize the fact that those regions are also the regions where XGBoost is not showing good performances, although XGBoost's predictions for those clusters are still acceptable. Given the good performances of the LSTM and XGBoost on most of the regions, we suppose that poor predicitons in clusters 1, 4, 9, 16, 28, 29 could be caused by how data are shaped for these regions. One assumption is that some geographical factors are influencing the distribution of the demand, and therefore its predictability. Thus, we want to plot the cluster centers in the New York City map, to see where these clusters are located. In Figure 10.12 we plot the representative of the clusters who are experiencing poor LSTM predictions.

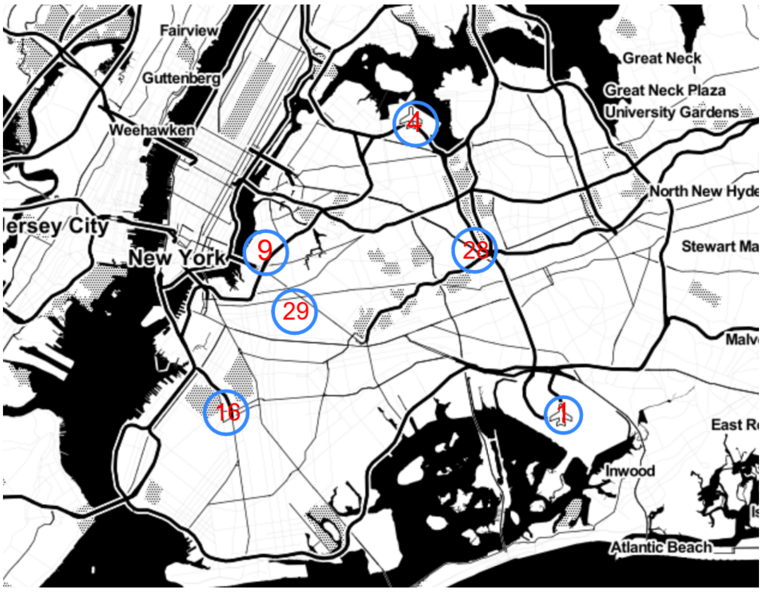


Figure 10.12: Enumerated markers in the New York City map for the centers of the clusters that have poor sMAPE performance for LSTM prediction.

It is clear from Figure 10.12 that the clusters where the prediction is poor are influenced by some geographical factors regarding the difference between the Manhattan area and the Brooklyn-Queens area. Instead, if we plot the cluster centers of the regions who have sMAPE lower than 5 (taken as a threshold just with the purpose of choosing some good predictions), we figure out that they are located into the Manhattan area and specifially they are spread over the Midtown and Downtown districts. We can see it in Figure 10.13 .

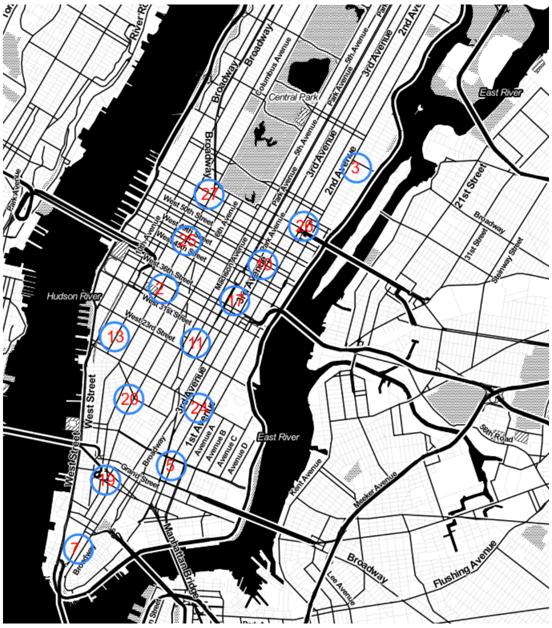


Figure 10.13: Enumerated markers in the New York City map for the centers of some of the clusters that have good sMAPE performance for LSTM prediction.

The dataset we use for our study contains information about the Yellow Taxi Cabs of New York CIty. Though, we need to consider that from August 2013, the NYC Taxi

& Limousine Commission (NYCTLC) introduced a new taxi fleet of Green cabs. These Green cabs were introduced with the goal of providing the residents of Brooklyn, Queens, the Bronx and Upper Manhattan more access to metered taxis. In fact, Yellow cabs prefer to operate in the high demand density areas of Manhattan, and this makes the pickups density very low in the suburban areas of the city. This fact was confirmed by a statement that, tracking the GPS routes of Taxis, found out that 95% of the Yellow cabs pickups happened in Manhattan, and at JKF and LaGuardia airports. Green cabs can only pick-up rides outside of the Manhattan, JFK and LaGuardia airports. Moreover, from years 2014-2015, there has been an increase on the number of pickups operated by Uber company, especially in the outer boroughs where Uber has realised most of its growth [78]. In fact, most of the rides taken outside of the Manhattan and airports areas, are operated by Uber as we can see in Figure 10.14. Uber is increasing in strength outside of Manhattan: its biggest market-share increases came notably in northeastern Queens and southwestern Brooklyn, two of the areas the farthest from Midtown [14]. Therefore, the presence of Green cabs and companies like Uber and Lyft, and the historical trend of Yellow cabs, suggest that clusters C_{16}, C_{28} and C_{29} can be highly influenced by these factors. This means that in these areas, Yellow taxis do not follow a regular trend but rather an almost random distribution of the demand, making it difficult for the LSTM to build a prediction model. Cluster C_9 has not too poor prediction, having $sMAPE = 23.13\%$, and as we see in Figure 10.14 this is an area which is still slightly dependent on the Yellow cabs operation.

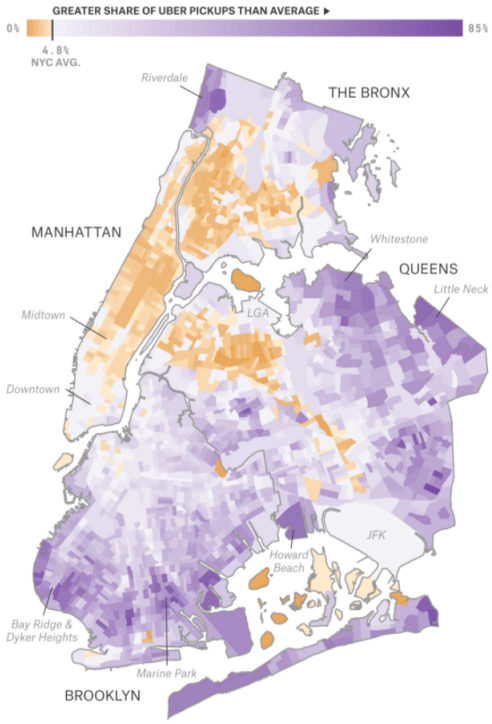


Figure 10.14: Share of all Uber, Yellow cabs and Green cabs pickups from April through September 2014 [14]. As we can see, areas of poor predictability for our dataset are areas where the use of Uber is strongly preferred to the use of Yellow cabs, implying low regularity in the use of the yellow taxis.

On the other hand, clusters C_1 and C_4 are located respectively at the LaGuardia and

JFK Airports. Although these two clusters do not show high predictability, we cannot say that prediction is completely messed up. Even if in this area we have high demand (as Yellow cabs are highly operating in these clusters), prediction is poor probably because influenced by other factors that are not purely geographical. Some of these factors are related to the airport operation and the high competition with the Uber and Lyft companies, that are very active in these areas, because of the high demand. Moreover, airports have other means of transport that NYC is providing.

Given the reasoning on what could be the factors influencing the taxi demand in the different areas of New York City, we decide to confine the use of the LSTM prediction model to the New York Manhattan borough. In fact, the results obtained for the LSTM (and also for the gradient boosting model) show an high level of predictability and we can think of doing separate pickups demand studies depending on the specific city region. Hence, we define a new set of regions $\mathcal{D}_{MAN}$ which will not account for the suburban clusters. This is:

$$\mathcal{D}_{MAN} = \mathcal{D}_{NYC} \setminus C_{BQ} \quad \text{with } C_{BQ} = \{C_1, C_4, C_9, C_{16}, C_{28}, C_{29}\} \tag{10.24}$$

where C_{BQ} is the set of the clusters representing the areas of Brooklyn and Queens while D_{NYC} is the set of clusters for the entire city of New York. If $N_C = 30$ was the number of all the clusters of D_{NYC}, we now define $N_C^* = 24$ as the number of clusters in D_{MAN}. Thus, we can now compute the global sMAPE for D_{MAN} as:

$$sMAPE_{MAN} = \frac{1}{N_C^*} \sum_{C_i \in \mathcal{D}_{MAN}} sMAPE_{C_i} \tag{10.25}$$

For the LSTM model we have $sMAPE_{MAN}^{LSTM} = 5.821\%$. If we decide to include also clusters C_1 (LaGuardia Airport), C_4 (JFK Airport) and C_9, we will have an average $sMAPE_{MAN,air}^{LSTM} = 8.751\%$ which is better than the naïve approach but slightly worse than the gradient boosting algorithm. Indeed, the gradient boosting has $sMAPE_{MAN,air}^{XGBoost} = 8.481$. For XGBoost, not considering also the clusters of the airport regions we will have an average $sMAPE_{MAN}^{XGBoost} = 7.972\%$, which means that in the Manhattan area it performs worse than the LSTM model. Instead, the naïve approach has an average $sMAPE_{MAN,air}^{Naive} = 12.8266\%$, and $sMAPE_{MAN}^{Naive} = 12.827\%$ for the only Manhattan area. Finally, we can conclude that, on top of all the considerations on the predictability and completeness of our dataset about the overall taxi demand of New York City, the XGBoost model can improve the average prediction on the Manhattan and airports areas over the naïve approach of about 34%, while LSTM can improve the average forecast of about 32%. If we focuse only on the Manhattan borough, though, we have $sMAPE_{MAN}^{Naive} = 12.823\%$, $sMAPE_{MAN}^{LSTM} = 5.821\%$ and $sMAPE_{MAN}^{XGBoost} = 7.972\%$ which means an improvement of about 55% (54.6%) for the LSTM and of 38% (37.9%) for XGBoost over the naïve method.
We can also notice, looking at Figure 10.6 and Figure 10.7, that are the representations of the predictions for cluster 2, which is a good performance cluster both for the LSTM ($sMAPE_2^{LSTM} = 4.898\%$) and the gradient boosting models ($sMAPE_2^{XGBoost} = 7.210\%$), that LSTM is able to better follow the dynamics of the pickups demand trend, meaning that it is more prone to learn from the behaviour of the previous time instants and to find relations between them.

Chapter 11

Conclusions

The second part of the thesis has deepened what is the problem regarding the imbalance of the vehicles fleet with respect to the demand of pickups from customers of the transportation service. To this aim, we try to balance the distribution of the vehicles by developing a prediction model in order to forecast the hourly number of pickups requests in different areas of a map. The models are built for the case of New York City after having gathered the most important information from the dataset of the yellow taxi cabs rides for April, May and June of years 2015 and 2016. The dataset has open access and contains the information for each single route performed. With the use of Pandas and Scikit-learn libraries for Python, we have preprocessed the dataset by filtering the noisy data and segmenting it to extract the information about the spatiotemporal distribution of the pickups. Indeed, noisy data (that come mainly from failures in the GPS tracking systems) have to be removed from the dataset in order to avoid afflicting the quality of the model training phase. Filtering has been performed under some metrics regarding the percentile analysis of the vehicle speed, trip distance, trip duration and trip fare.
Furthermore, we have removed all data that have pickup or dropoff outside the New York City area. We obtained a new cleaned dataset that contains about 97% of the data of the original dataset. Segmentation is required to divide the pickups dataset into clusters. We used a k-means algorithm to divide the dataset into 30 clusters, that is the number of clusters which avoids regions centers to be too close to each other. Later, we divided the monthly data into hourly time bins. In this way we will obtain a dataset that has for each cluster a set of time bins. In every time bin we have the number of pickups occurred during that time slot. Weather hourly data has been added to the dataset in order to further characterize the data with the objective of improving the prediction.
After the preprocessing phase, the dataset has been used to train two different machine learning models that are the LSTM recurrent neural network and the gradient boosting model. For the implementation of the LSTM model, we used the Keras API that runs on top of Tensorflow, while for the implementation of the gradient boosting model we relied on the XGBoost algorithm, that is nowadays on of the most performant prediction algorithms. The results of the two models have been compared with the results from a very simple naïve prediction model. To evaluate the performances of the models we used a test dataset and we computed the sMAPE and RMSE of the next hour forecast for each of the 30 clusters. We found out that the LSTM model outperforms the XGBoost algorithm in most of the regions of the map with average sMAPE (in these regions) of 5.8% and 8.0% respectively. Unfortunately, the LSTM performs really bad in some regions of the city with respect to the other models.
Brooklyn boroughs and airports areas seem to be unpredictable for the LSTM, maybe

because of the random behavior of the data due to a low usage of the yellow cabs with respect to the Uber and Lyft services and the green cabs, that were introduced in order to balance the absence of yellow taxis in these areas.
To conclude, the best model for the overall average predictions turns out to be the XGBoost gradient boosting. Indeed, it is capable of very good predictions in most of the clusters with a prediction sMAPE of about 9.6% when including the Brooklyn and airports areas, with an improvement of 30% on average over the naïve approach.
Further improvements can include merging the Uber, Lyft and green cabs pickups data to the yellow cabs rides dataset, in order to cover the complete taxi demand in the whole New York City area and obtain better forecasts. Another improvement that could bring to more realistic applications is to implement an hexagon clustering technique for the pickup locations. This would bring to a more reliable division of the pickups into regions because of the nice properties that have been proved about the use of hexagons.
Moreover, we could think of linking the demand forecasting model to the vehicle assignment algorithm, in such a way to optimize the vehicles distribution and ease the assignment process in order to satisfy a larger number of customers and decrease the travel and waiting times.

Bibliography

[1] Anwar Afian, Amedeo Odoni, and Daniela Rus. Inferring unmet demand from taxi probe data. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 861–868. IEEE, 2015.

[2] Niels Agatz, Alan Erera, Martin Savelsbergh, and Xing Wang. Optimization for dynamic ride-sharing: A review. *European Journal of Operational Research*, 223(2):295–303, 2012.

[3] Maria Albareda Sambola. *Models and algorithms for location-routing and related problems*. Universitat Politècnica de Catalunya, 2003.

[4] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[5] Javier Alonso-Mora, Samitha Samaranayake, Alex Wallar, Emilio Frazzoli, and Daniela Rus. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proceedings of the National Academy of Sciences*, 114(3):462–467, 2017.

[6] Gürdal Arslan, Jason R. Marden, and Jeff S. Shamma. Autonomous vehicle-target assignment: A game-theoretical formulation. 2007.

[7] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.

[8] Said Baadel, Fadi Thabtah, and Joan Lu. Overlapping clustering: A review. In *2016 SAI Computing Conference (SAI)*, pages 233–237. IEEE, 2016.

[9] Mohamed Badreldin, Ahmed Hussein, and Alaa Khamis. A Comparative Study between Optimization and Market-Based Approaches to Multi-Robot Task Allocation. *Advances in Artificial Intelligence (16877470)*, 2013.

[10] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *arXiv preprint arXiv:1203.6402*, 2012.

[11] Javier Béjar Alonso. K-means vs Mini Batch K-means: A comparison. 2013.

[12] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *omega*, 34(3):209–219, 2006.

[13] Gerardo Berbeglia, Jean-François Cordeau, and Gilbert Laporte. Dynamic pickup and delivery problems. *European journal of operational research*, 202(1):8–15, 2010.

[14] Carl Bialik, Andrew Flowers, Reuben Fischer-Baum, and Dhrumil Mehta. Uber is serving new york's outer boroughs more than taxis are. *FiveThirtyEight. com*, 2015.

[15] Norman Biggs, Norman Linstead Biggs, and Biggs Norman. *Algebraic graph theory*, volume 67. Cambridge university press, 1993.

[16] Colin P. D. Birch, Sander P. Oom, and Jonathan A. Beecham. Rectangular and hexagonal grids used for observation, experiment and simulation in ecology. *Ecological modelling*, 206(3-4):347–359, 2007.

[17] Stephen Boyd, Stephen P. Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[18] Paul Bradley, Johannes Gehrke, Raghu Ramakrishnan, and Ramakrishnan Srikant. Scaling mining algorithms to large databases. *Communications of the ACM*, 45(8):38–43, 2002.

[19] Jason Brownlee. Time series prediction with lstm recurrent neural networks in python with keras. *Available at: machinelearningmastery. com*, page 18, 2016.

[20] Suzana J. Camargo, Andrew W. Robertson, Scott J. Gaffney, Padhraic Smyth, and Michael Ghil. Cluster analysis of typhoon tracks. Part I: General properties. *Journal of Climate*, 20(14):3635–3653, 2007.

[21] Pablo Samuel Castro, Daqing Zhang, Chao Chen, Shijian Li, and Gang Pan. From taxi GPS traces to social and community dynamics: A survey. *ACM Computing Surveys (CSUR)*, 46(2):1–34, 2013.

[22] Sanjay Chawla, Yu Zheng, and Jiafeng Hu. Inferring the root cause in road traffic anomalies. In *2012 IEEE 12th International Conference on Data Mining*, pages 141–150. IEEE, 2012.

[23] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[24] Ximing Chen, Fei Miao, George J. Pappas, and Victor Preciado. Hierarchical data-driven vehicle dispatch and ride-sharing. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 4458–4463. IEEE, 2017.

[25] Francois Chollet et al. Keras, 2015.

[26] Brian Coltin and Manuela Veloso. Scheduling for transfers in pickup and delivery problems with very large neighborhood search. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[27] Michele Conforti, Gérard Cornuéjols, Giacomo Zambelli, et al. *Integer programming*, volume 271. Springer, 2014.

[28] Mario Cools, Elke Moons, and Geert Wets. Assessing the impact of weather on traffic intensity. *Weather, Climate, and Society*, 2(1):60–68, 2010.

[29] Jean-François Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54(3):573–586, 2006.

[30] Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem (DARP): Variants, modeling issues and algorithms. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(2):89–101, 2003.

[31] IBM ILOG Cplex. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[32] Emilie Danna, Edward Rothberg, and Claude Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming*, 102(1):71–90, 2005.

[33] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In *Algorithmics of large and complex networks*, pages 117–139. Springer, 2009.

[34] Joseph J. Di Gianni. Exploration of the current state and directions of dynamic ridesharing. 2015.

[35] M. Bernardine Dias, Robert Zlot, Nidhi Kalra, and Anthony Stentz. Market-based multirobot coordination: A survey and analysis. *Proceedings of the IEEE*, 94(7):1257–1270, 2006.

[36] Nasser A. El-Sherbeny. Vehicle routing with time windows: An overview of exact, heuristic and metaheuristic methods. *Journal of King Saud University-Science*, 22(3):123–131, 2010.

[37] Laurene V. Fausett. *Fundamentals of neural networks: architectures, algorithms and applications*. Pearson Education India, 2006.

[38] Jerome H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[39] Masabumi Furuhata, Maged Dessouky, Fernando Ordóñez, Marc-Etienne Brunet, Xiaoqing Wang, and Sven Koenig. Ridesharing: The state-of-the-art and future directions. *Transportation Research Part B: Methodological*, 57:28–46, 2013.

[40] Brian P. Gerkey and Maja J. Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International journal of robotics research*, 23(9):939–954, 2004.

[41] Julio Godoy and Maria Gini. Task allocation for spatially and temporally distributed tasks. In *Intelligent Autonomous Systems 12*, pages 603–612. Springer, 2013.

[42] Bruce L. Golden, Subramanian Raghavan, and Edward A. Wasil. *The vehicle routing problem: latest advances and new challenges*, volume 43. Springer Science & Business Media, 2008.

[43] Y. Gong, Bin Fang, S. Zhang, and J. Zhang. Predict New York city taxi demand. *NYC Data Science Academy*, 2016.

[44] Jae Hyuk Han. Comparing Models for Time Series Analysis. 2018.

[45] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[46] Leif Hockstad and L. Hanel. Inventory of US greenhouse gas emissions and sinks. Technical report, Environmental System Science Data Infrastructure for a Virtual Ecosystem, 2018.

[47] Zihan Hong, Ying Chen, Hani S. Mahmassani, and Shuang Xu. Commuter ride-sharing using topology-based vehicle trajectory clustering: Methodology, application and impact evaluation. *Transportation Research Part C: Emerging Technologies*, 85:573–590, 2017.

[48] Hadi Hosni, Joe Naoum-Sawaya, and Hassan Artail. The shared-taxi problem: Formulation and solution methods. *Transportation Research Part B: Methodological*, 70:303–318, 2014.

[49] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence*, 24(7):881–892, 2002.

[50] Matthew G. Karlaftis and Eleni I. Vlahogianni. Statistical methods versus neural networks in transportation research: Differences, similarities and some insights. *Transportation Research Part C: Emerging Technologies*, 19(3):387–399, 2011.

[51] Kevin Keay and Ian Simmonds. The association of rainfall and other weather variables with road traffic volume in Melbourne, Australia. *Accident analysis & prevention*, 37(1):109–124, 2005.

[52] Ahmed Kharrat, Iulian Sandu Popa, Karine Zeitouni, and Sami Faiz. Clustering algorithm for network constraint trajectories. In *Headway in Spatial Data Handling*, pages 631–647. Springer, 2008.

[53] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[54] Ed Klotz and Alexandra M. Newman. Practical guidelines for solving difficult mixed integer linear programs. *Surveys in Operations Research and Management Science*, 18(1-2):18–32, 2013.

[55] Jacob Kogan, Charles Nicholas, Marc Teboulle, et al. *Grouping multidimensional data*. Springer, 2006.

[56] Krzysztof Koperski, Jiawei Han, and Junas Adhikary. Mining knowledge in geographical data. *Communications of the ACM*, 26(1):65–74, 1998.

[57] G. Ayorkor Korsah, Anthony Stentz, and M. Bernardine Dias. A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research*, 32(12):1495–1512, 2013.

[58] Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[59] Allan Larsen. The dynamic vehicle routing problem. 2000.

[60] Jan Karel Lenstra and A.H.G. Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227, 1981.

[61] Siyu Liao, Liutong Zhou, Xuan Di, Bo Yuan, and Jinjun Xiong. Large-scale short-term urban taxi demand forecasting using deep learning. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 428–433. IEEE, 2018.

[62] Mustafa Lokhandwala and Hua Cai. Dynamic ride sharing using traditional taxis and shared autonomous taxis: A case study of NYC. *Transportation Research Part C: Emerging Technologies*, 97:45–60, 2018.

[63] Yongmei Lu. *Spatial Clustering, Detection and Analysis of*, pages 317–324. 12 2009.

[64] Shuo Ma, Yu Zheng, and Ouri Wolfson. T-share: A large-scale dynamic taxi ridesharing service. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 410–421. IEEE, 2013.

[65] Xiaolei Ma, Zhimin Tao, Yinhai Wang, Haiyang Yu, and Yunpeng Wang. Long short-term memory neural network for traffic speed prediction using remote microwave sensor data. *Transportation Research Part C: Emerging Technologies*, 54:187–197, 2015.

[66] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[67] Harvey J. Miller and Jiawei Han. *Geographic data mining and knowledge discovery*. CRC press, 2009.

[68] Luis Moreira-Matias, Joao Gama, Michel Ferreira, Joao Mendes-Moreira, and Luis Damas. Predicting taxi–passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, 2013.

[69] Naoto Mukai and Naoto Yoden. Taxi demand forecasting based on taxi probe data by neural network. In *Intelligent interactive multimedia: Systems and services*, pages 589–597. Springer, 2012.

[70] Noah Nelson. Why millennials are ditching cars and redefining ownership. *National Public Radio Morning Edition*, 2013.

[71] Lalit Sivanandan Nookala. *Weather impact on traffic conditions and travel time prediction*. PhD thesis, Citeseer, 2006.

[72] Ernesto Nunes, Marie Manner, Hakim Mitiche, and Maria Gini. A taxonomy for task allocation problems with temporal and ordering constraints. *Robotics and Autonomous Systems*, 90:55–70, 2017.

[73] TLC NYC. New york city yellow cabs trip record data, https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[74] Sebastian Osswald, Niklas Brueckel, Carsten Brickwedde, Markus Lienkamp, and Martin Schoell. Taxi Checker: A Mobile Application for Real-Time Taxi Fare Analysis. In *Adjunct Proceedings of the 6th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, pages 1–6, 2014.

[75] Rafail Ostrovsky, Yuval Rabani, Leonard J. Schulman, and Chaitanya Swamy. The effectiveness of Lloyd-type methods for the k-means problem. *Journal of the ACM (JACM)*, 59(6):1–22, 2013.

[76] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[77] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[78] Lasse Korsholm Poulsen, Daan Dekkers, Nicolaas Wagenaar, Wesley Snijders, Ben Lewinsky, Raghava Rao Mukkamala, and Ravi Vatrapu. Green cabs vs. uber in new york city. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 222–229. IEEE, 2016.

[79] Warren B. Powell, Patrick Jaillet, and Amedeo Odoni. Stochastic and dynamic networks and routing. *Handbooks in operations research and management science*, 8:141–295, 1995.

[80] Tatyana Poznyak, Jorge Isaac Chairez Oria, and Alex Poznyak. *Ozonation and Biodegradation in Environmental Engineering: Dynamic Neural Network Approach.* Elsevier, 2018.

[81] Oriana Riva and Cristian Borcea. The urbanet revolution: Sensor power to the people! *IEEE Pervasive Computing*, 6(2):41–49, 2007.

[82] Paolo Santi, Giovanni Resta, Michael Szell, Stanislav Sobolevsky, Steven H. Strogatz, and Carlo Ratti. Quantifying the benefits of vehicle pooling with shareability networks. *Proceedings of the National Academy of Sciences*, 111(37):13290–13294, 2014.

[83] Javier Morales Sarriera, Germán Escovar Álvarez, Kelly Blynn, Andrew Alesbury, Timothy Scully, and Jinhua Zhao. To share or not to share: Investigating the social aspects of dynamic ridesharing. *Transportation Research Record*, 2605(1):109–117, 2017.

[84] Martin W. P. Savelsbergh. Local search in routing problems with time windows. *Annals of Operations research*, 4(1):285–305, 1985.

[85] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, 1998.

[86] David Sculley. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pages 1177–1178, 2010.

[87] Gaurav Sharma. Taxi demand prediction new york city, 2018.

[88] Marius M. Solomon. On the worst-case performance of some heuristics for the vehicle routing and scheduling problem with time window constraints. *Networks*, 16(2):161–174, 1986.

[89] Chaoming Song, Zehui Qu, Nicholas Blumm, and Albert-László Barabási. Limits of predictability in human mobility. *Science*, 327(5968):1018–1021, 2010.

[90] Mitja Stiglic, Niels Agatz, Martin Savelsbergh, and Mirko Gradisar. The benefits of meeting points in ride-sharing systems. *Transportation Research Part B: Methodological*, 82:36–53, 2015.

[91] Shangyu Sun, Huayi Wu, and Longgang Xiang. City-wide traffic flow forecasting using a deep convolutional neural network. *Sensors*, 20(2):421, 2020.

[92] Paolo Toth and Daniele Vigo. An overview of vehicle routing problems. In *The vehicle routing problem*, pages 1–26. SIAM, 2002.

[93] Weather Underground. New york city, ny weather history, https://www.wunderground.com/history/monthly/us/ny/new-york-city/klga/date/2016-4-16.

[94] Ukrish Vanichrujee, Teerayut Horanont, Wasan Pattara-atikom, Thanaruk Theeramunkong, and Takahiro Shinozaki. Taxi Demand Prediction using Ensemble Model Based on RNNs and XGBOOST. In *2018 International Conference on Embedded Systems and Intelligent Technology & International Conference on Information and Communication Technology for Embedded Systems (ICESIT-ICICTES)*, pages 1–6. IEEE, 2018.

[95] Jr. Votaw and A. D. Orden. The peronnel assignment problem. *Symposium on Linear Inequalities and Programming*, 1952.

[96] Chunhui Wang, Qianqian Zhu, Zhenyu Shan, Yingjie Xia, and Yuncai Liu. Fusing heterogeneous traffic data by Kalman filters and Gaussian mixture models. In *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 276–281. IEEE, 2014.

[97] Jun Xu, Rouhollah Rahmatizadeh, Ladislau Bölöni, and Damla Turgut. Real-time prediction of taxi demand using recurrent neural networks. *IEEE Transactions on Intelligent Transportation Systems*, 19(8):2572–2581, 2017.

[98] Daqing Zhang, Bin Guo, and Zhiwen Yu. The emergence of social and community intelligence. *Computer*, 44(7):21–28, 2011.

[99] Kai Zhao, Denis Khryashchev, Juliana Freire, Claudio Silva, and Huy Vo. Predicting taxi demand at high spatial resolution: Approaching the limit of predictability. In *2016 ieee international conference on Big data (big data)*, pages 833–842. IEEE, 2016.

[100] Huiyu Zhou and Kotaro Hirasawa. Spatiotemporal traffic network analysis: technology and applications. *Knowledge and Information Systems*, pages 1–37, 2019.